Applied and Numerical Harmonic Analysis

$$\hat{f}(\gamma) = \int f(x) e^{-2\pi i x \gamma} dx$$

Alexander Paprotny
Michael Thess

Realtime Data Mining

Self-Learning Techniques for Recommendation Engines

Birkhäuser

Applied and Numerical Harmonic Analysis

Series Editor
John J. Benedetto
University of Maryland
College Park, MD, USA

Editorial Advisory Board
Akram Aldroubi
Vanderbilt University
Nashville, TN, USA

Douglas Cochran
Arizona State University
Phoenix, AZ, USA

Hans G. Feichtinger
University of Vienna
Vienna, Austria

Christopher Heil
Georgia Institute of Technology
Atlanta, GA, USA

Stéphane Jaffard
University of Paris XII
Paris, France

Jelena Kovačević
Carnegie Mellon University
Pittsburgh, PA, USA

Gitta Kutyniok
Technische Universität Berlin
Berlin, Germany

Mauro Maggioni
Duke University
Durham, NC, USA

Zuowei Shen
National University of Singapore
Singapore, Singapore

Thomas Strohmer
University of California
Davis, CA, USA

Yang Wang
Michigan State University
East Lansing, MI, USA

For further volumes:
http://www.springer.com/series/4968

Alexander Paprotny • Michael Thess

Realtime Data Mining

Self-Learning Techniques for Recommendation Engines

Alexander Paprotny
Research and Development
prudsys AG
Berlin, Germany

Michael Thess
Research and Development
prudsys AG
Chemnitz, Germany

ISSN 2296-5009 ISSN 2296-5017 (electronic)
ISBN 978-3-319-01320-6 ISBN 978-3-319-01321-3 (eBook)
DOI 10.1007/978-3-319-01321-3
Springer Cham Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013953342

Mathematics Subject Classification (2010): 68T05, 68Q32, 90C40, 65C60, 62-07

© Springer International Publishing Switzerland 2013
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.birkhauser-science.com)

ANHA Series Preface

The Applied and Numerical Harmonic Analysis (ANHA) book series aims to provide the engineering, mathematical, and scientific communities with significant developments in harmonic analysis, ranging from abstract harmonic analysis to basic applications. The title of the series reflects the importance of applications and numerical implementation, but richness and relevance of applications and implementation depend fundamentally on the structure and depth of theoretical underpinnings. Thus, from our point of view, the interleaving of theory and applications and their creative symbiotic evolution is axiomatic.

Harmonic analysis is a wellspring of ideas and applicability that has flourished, developed, and deepened over time within many disciplines and by means of creative cross-fertilization with diverse areas. The intricate and fundamental relationship between harmonic analysis and fields such as signal processing, partial differential equations (PDEs), and image processing is reflected in our state-of-the-art *ANHA* series.

Our vision of modern harmonic analysis includes mathematical areas such as wavelet theory, Banach algebras, classical Fourier analysis, time-frequency analysis, and fractal geometry as well as the diverse topics that impinge on them.

For example, wavelet theory can be considered an appropriate tool to deal with some basic problems in digital signal processing, speech and image processing, geophysics, pattern recognition, biomedical engineering, and turbulence. These areas implement the latest technology from sampling methods on surfaces to fast algorithms and computer vision methods. The underlying mathematics of wavelet theory depends not only on classical Fourier analysis but also on ideas from abstract harmonic analysis, including von Neumann algebras and the affine group. This leads to a study of the Heisenberg group and its relationship to Gabor systems, and of the metaplectic group for a meaningful interaction of signal decomposition methods. The unifying influence of wavelet theory in the aforementioned topics illustrates the justification for providing a means for centralizing and disseminating information from the broader, but still focused, area of harmonic analysis. This will be a key role of *ANHA*. We intend to publish with the scope and interaction that such a host of issues demand.

Along with our commitment to publish mathematically significant works at the frontiers of harmonic analysis, we have a comparably strong commitment to publish major advances in the following applicable topics in which harmonic analysis plays a substantial role:

Antenna theory	*Prediction theory*
Biomedical signal processing	*Radar applications*
Digital signal processing	*Sampling theory*
Fast algorithms	*Spectral estimation*
Gabor theory and applications	*Speech processing*
Image processing	*Time-frequency and time-scale analysis*
Numerical partial differential equations	*Wavelet theory*

The above point of view for the *ANHA* book series is inspired by the history of Fourier analysis itself, whose tentacles reach into so many fields.

In the last two centuries, Fourier analysis has had a major impact on the development of mathematics, on the understanding of many engineering and scientific phenomena, and on the solution of some of the most important problems in mathematics and the sciences. Historically, Fourier series were developed in the analysis of some of the classical PDEs of mathematical physics; these series were used to solve such equations. In order to understand Fourier series and the kinds of solutions they could represent, some of the most basic notions of analysis were defined, for example, the concept of "function." Since the coefficients of Fourier series are integrals, it is no surprise that Riemann integrals were conceived to deal with uniqueness properties of trigonometric series. Cantor's set theory was also developed because of such uniqueness questions.

A basic problem in Fourier analysis is to show how complicated phenomena, such as sound waves, can be described in terms of elementary harmonics. There are two aspects of this problem: first, to find, or even define properly, the harmonics or spectrum of a given phenomenon, for example, the spectroscopy problem in optics; second, to determine which phenomena can be constructed from given classes of harmonics, as done, for example, by the mechanical synthesizers in tidal analysis.

Fourier analysis is also the natural setting for many other problems in engineering, mathematics, and the sciences. For example, Wiener's Tauberian theorem in Fourier analysis not only characterizes the behavior of the prime numbers but also provides the proper notion of spectrum for phenomena such as white light; this latter process leads to the Fourier analysis associated with correlation functions in filtering and prediction problems, and these problems, in turn, deal naturally with Hardy spaces in the theory of complex variables.

Nowadays, some of the theory of PDEs has given way to the study of Fourier integral operators. Problems in antenna theory are studied in terms of unimodular trigonometric polynomials. Applications of Fourier analysis abound in signal processing, whether with the fast Fourier transform (FFT) or filter design or the

adaptive modeling inherent in time-frequency-scale methods such as wavelet theory. The coherent states of mathematical physics are translated and modulated Fourier transforms, and these are used, in conjunction with the uncertainty principle, for dealing with signal reconstruction in communications theory. We are back to the raison d'être of the *ANHA* series!

University of Maryland *John J. Benedetto*
College Park Series Editor

Reprinted with the permission of Frank Nathan

Preface

The area of realtime data mining is currently developing at an exceptionally dynamic pace. Realtime data mining systems are the counterpart of today's "classic" data mining systems. Whereas the latter learn from historical data and then use it to deduce necessary actions, realtime analytics systems learn and act continuously and autonomously. In the vanguard of these new analytics systems are *recommendation engines* (REs). They are principally found on the Internet, where all information is available in real time and an immediate feedback is guaranteed.

In this book, we describe novel mathematical concepts for recommendation engines based on realtime learning. These feature a sound mathematical framework which unifies approaches based on control and learning theories, tensor factorization, and hierarchical methods. Furthermore, they present promising results of numerous experiments on real-world data. Thus, the book introduces and demystifies this concept of "realtime thinking" for a specific application—recommendation engines. Additionally, the book provides useful knowledge about recommendation engines such as verification of results in A/B tests including calculation of confidence intervals, coding examples, and further research directions.

The main goal of the research presented in the book consists of devising a sound and effective mathematical and computational framework for automatic adaptive recommendation engines. Most importantly, we introduce an altogether novel control-theoretic approach to recommendation based on considering the customer of an (online) shop as a dynamic system upon which the recommendation engine acts as a closed-loop control system, the objective of which is maximizing the incurred reward (e.g., revenue). Besides that, we also cover classical data mining-based approaches and develop efficient numerical procedures for computing and, especially, updating the underlying matrix and tensor decompositions. Furthermore, we take a step toward a framework that unifies the two approaches, that is, the classical and the control-theoretic one. In summary, the book proposes a very modern approach to realtime analytics and includes a lot of new material.

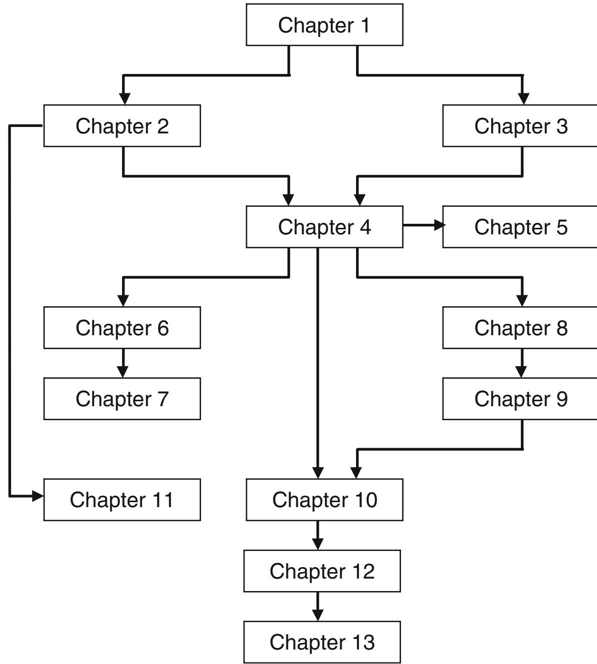
Currently, most books about recommendation engines focus on traditional techniques, such as collaborative filtering, basket analysis, and content-based recommendations. Recommendations are considered from a prediction point of view only, that is, the recommendation task is reduced to the prediction of content that the user is going to select with highest probability anyway. In contrast, in our book we consider recommendations as a control-theoretic problem by investigating the interaction of analysis and action. At this, an optimization problem with respect to maximum reward is considered.

Another important frequently recurring theme in our train of thought is that of hierarchical approaches. In recent decades, methods that capture and take into account effects at different scales have turned out to be a key ingredient to successfully tackling complex problems in signal processing and numerical solution of partial differential equations. Supported by the evidence that we shall present in this book, we strongly conjecture that this paradigm may give rise to major improvements in the efficiency of computational procedures deployed in the framework of realtime recommendation engines. We therefore would like to stress that this book is also a step toward introducing harmonic thinking in the theory and practice of recommendation engines.

The book targets, on one hand, computer scientists and specialists in machine learning, especially from the area of recommendation systems, because it conveys a new way of realtime thinking especially by considering recommendation tasks as control-theoretic problems. On the other hand, the book may be of considerable interest to application-oriented mathematicians, because it consistently combines some of the most promising mathematical areas, namely, control theory, multilevel approximation, and tensor factorization.

Owing to the complexity of the subject, the book cannot go into all the details of the mathematical theory, let alone its implementation. Nevertheless, it sets out the basic assumptions and tools that are needed for an understanding of the theory. In some areas of fundamental importance, we also offer more detailed mathematical examples. Overall, however, we have tried to keep the mathematical illustrations short and to the point.

The document structure is as follows. Chapter 1 offers a general introduction to methods of realtime analytics and sets out their advantages and disadvantages as compared with conventional analytics methods, which learn only from historical data. Chapter 2 describes conventional approaches for recommendation engines and shows how their inherently static methodology is their main weak point. The use of realtime analytics methods is suggested as a way of overcoming precisely this problem and, specifically, reinforcement learning (RL), one of the very newest disciplines, which models the interplay of analysis and action. Chapter 3 provides a brief introduction to RL, while Chap. 4 applies this knowledge to recommendation engines. There are still a number of fundamental problems to resolve, however, requiring the introduction of some additional empirical assumptions. This is done in Chap. 5, resulting in a complete RL-based approach for recommendation engines.



The next chapters are devoted to improve our solution, especially concerning stability and speed of convergence. Thus, in Chap. 6 we study hierarchical methods and add a hierarchical convergence accelerator to further boost the learning speed. Chapter 7 represents an extension to the topic of hierarchical methods where a powerful adaptive scoring technique is described—sparse grids. For a better exploitation of the data in the calculation of recommendations, in Chap. 8 we introduce matrix factorization techniques along with some adaptive implementation. Using the tensor concept, Chap. 9 extends the factorization to the high-dimensional case. This enables us to combine hierarchical RL with adaptive tensor factorization in order to include additional dimensions into the realtime calculation of recommendations. This "big picture," which is still in the very beginning, is described in Chap. 10 and concludes the technical description of our new recommendation approach.

In Chap. 11, we discuss statistically rigorous methods for measuring the success of recommendation engines. Chapter 12 is devoted to the prudsys XELOPES library which implements most of the algorithms described in this book and provides a powerful infrastructure for realtime learning. Finally, in Chap. 13 we summarize the main elements covered in the book.

Parts of the book provide an easily understandable introduction to realtime recommendations and do not require deep mathematical knowledge. Especially, this applies to Chaps. 1 and 2 as well as Chaps. 11, 12, and 13. Chapters 3, 4, and 5 are devoted to reinforcement learning and assume basic knowledge of algebra and statistics. In contrast, Chaps. 6, 7, 8, 9, and 10 address mathematically more experienced readers and require solid knowledge of linear algebra and analysis.

Acknowledgements

We would like to thank André Müller and Sven Gehre for their assistance in the tensor factorization chapter and Jochen Garcke for his critical review of the manuscript. We would also like to thank Holm Sieber for his help in the mathematical treatment of multiple recommendations and, additionally, Toni Volkmer for deriving the confidence intervals of revenue increase. Further we would like to thank the many reviewers who provided us with critical comments and suggestions. In particular, we would like to mention Jens Scholz, Tina Stopp, Gerard Zenker, and Brian Craig, as well as the anonymous reviewers allocated by the publisher.

Berlin, Germany Alexander Paprotny

Contents

1 Brave New Realtime World: Introduction 1
1.1 Historical Perspective 1
1.2 Realtime Analytics Systems 2
1.3 Advantages of Realtime Analytics Systems 3
1.4 Disadvantages of Realtime Analytics Systems 4
1.5 Combining Offline and Online Analysis 6
1.6 Methodical Remarks 6

2 Strange Recommendations? On the Weaknesses of Current Recommendation Engines 11
2.1 Introduction to Recommendation Engines 11
2.2 Weaknesses of Current Recommendation Engines and How to Overcome Them 12

3 Changing Not Just Analyzing: Control Theory and Reinforcement Learning 15
3.1 Modeling 16
3.2 Markov Property 17
3.3 Implementing the Policy: Selecting the Actions 18
3.4 Model of the Environment 19
3.5 The Bellman Equation 20
3.6 Determining an Optimal Solution 24
3.7 The Adaptive Case 26
3.8 The Model-Free Approach 28
3.9 Remarks on the Model 31
3.9.1 Infinite-Horizon Problems 31
3.9.2 Properties of Graphs and Matrices 32
3.9.3 The Steady-State Distribution 33
3.9.4 On the Convergence and Implementation of RL Methods 35
3.10 Summary 40

4 Recommendations as a Game: Reinforcement Learning for Recommendation Engines 41
4.1 Basic Approach 43
4.2 Multiple Recommendations 45
4.2.1 Linear Approach 45
4.2.2 Nonlinear Approach 46
4.3 Remarks on the Modeling 53
4.4 Verification Methods 54
4.5 Summary 56

5 How Engines Learn to Generate Recommendations: Adaptive Learning Algorithms 57
5.1 Unconditional Approach 58
5.2 Conditional Approach 61
5.2.1 Discussion 63
5.2.2 Special Cases 66
5.2.3 Estimation of Transition Probabilities 67
5.3 Combination of Conditional and Unconditional Approaches 76
5.4 Experimental Results 79
5.4.1 Verification of the Environment Model 80
5.4.2 Extension of the Simulation 84
5.4.3 Experimental Results 86
5.5 Summary 90

6 Up the Down Staircase: Hierarchical Reinforcement Learning 91
6.1 Introduction 92
6.1.1 Analytical Approach 92
6.1.2 Algebraic Approach 99
6.2 Multilevel Methods for Reinforcement Learning 103
6.2.1 Interpolation and Restriction Based on State Aggregation 104
6.2.2 The Model-Based Case: AMG 106
6.2.3 Model-Free Case: TD with Additive Preconditioner 112
6.3 Learning on Category Level 116
6.4 Summary 118

7 Breaking Dimensions: Adaptive Scoring with Sparse Grids 119
7.1 Introduction 119
7.2 The Sparse Grid Approach 122
7.2.1 Discretization 124
7.2.2 Grid-Based Discrete Approximation 125
7.2.3 Sparse Grid Space 126
7.2.4 The Sparse Grid Combination Technique 131
7.2.5 Adaptive Sparse Grids 134
7.2.6 Further Sparse Grid Versions 136

7.3 Experimental Results 138
7.3.1 Two-Dimensional Problems 138
7.3.2 High-Dimensional Problems 140
7.4 Summary 142

8 Decomposition in Transition: Adaptive Matrix Factorization 143
8.1 Matrix Factorizations in Data Mining and Beyond 144
8.2 Collaborative Filtering 147
8.3 PCA-Based Collaborative Filtering 149
8.3.1 The Problem and Its Statistical Rationale 149
8.3.2 Incremental Computation of the Singular Value Decomposition 155
8.3.3 Computing Recommendations 159
8.4 More Matrix Factorizations 163
8.4.1 Lanczos Methods 163
8.4.2 RE-Specific Requirements 166
8.4.3 Nonnegative Matrix Factorizations 167
8.4.4 Experimental Results 169
8.5 Back to Netflix: Matrix Completion 171
8.6 A Note on Efficient Computation of Large Elements of Low-Rank Matrices 176
8.7 Summary 180

9 Decomposition in Transition II: Adaptive Tensor Factorization . . . 183
9.1 Beyond Behaviorism: Tensor-PCA-Based CF 183
9.1.1 What Is a Tensor? 183
9.1.2 And Why We Should Care 186
9.1.3 PCA for Tensorial Data: Tucker Tensor and Higher-Order SVD 187
9.1.4 . . . And How to Compute It Adaptively 191
9.1.5 Computing Recommendations 193
9.2 More Tensor Factorizations 198
9.2.1 CANDECOMP/PARAFAC 198
9.2.2 RE-Specific Factorizations 200
9.2.3 Problems of Tensor Factorizations 202
9.3 Hierarchical Tensor Factorization 203
9.3.1 Hierarchical Singular Value Decomposition 203
9.3.2 Tensor-Train Decomposition 204
9.4 Summary 207

10 The Big Picture: Toward a Synthesis of RL and Adaptive Tensor Factorization 209
10.1 Markov-k-Processes and Augmented State Spaces 210
10.2 Breaking the Curse of Dimensionality: A Tensor View on Augmented State Spaces 212
10.3 Estimation of Factorized Transition Probabilities 216

10.4 Factored Representation and Computation of the State Values 217
10.4.1 A Model-Based Approach 217
10.4.2 Model-Free Computation in Virtue of TD (λ) with Function Approximation 218
10.5 Clustering Sequences of Products 221
10.5.1 An Adaptive Approach 221
10.5.2 Switching Between Aggregation Bases 222
10.6 How It All Fits Together 223
10.7 Summary 224

11 What Cannot Be Measured Cannot Be Controlled: Gauging Success with A/B Tests 227
11.1 Same Environments in Both Groups 228
11.2 No Loss of Performance Through Recommendations 229
11.3 Assessing the Statistical Stability of the Results 229
11.4 Observing Simpson's Paradox 233
11.5 Summary 234

12 Building a Recommendation Engine: The XELOPES Library 235
12.1 The XELOPES Library 236
12.1.1 The Main Design Principles 236
12.1.2 The Building Blocks of the Library 243
12.1.3 The Data Mining Framework 254
12.1.4 The Mathematics Package 260
12.2 The Realtime Analytics Framework of XELOPES 269
12.2.1 The Agent Framework 269
12.2.2 The Reinforcement Learning Package 274
12.2.3 The RL-Based Recommendation Package 284
12.3 Application Example of XELOPES: The prudsys RDE 298
12.4 Summary 299

13 Last Words: Conclusion 301

References 305

Summary of Notation

Number Sets

$\mathbf{N}$	The set of natural numbers
$\mathbf{N}_0$	The set of natural numbers with 0
$\Re$	The set of real numbers

Set Operators and Relations

$\cup$	Union
$\cap$	Intersection
$\backslash$	Exclusion
$\prod_{i \in \mathfrak{I}} X_i$	Cartesian product of the sets X_i
$\subset$	Subset or equal

Discrete Sets and Graphs

$\underline{n}$	The set $\{1, \ldots, n\}$
$\lvert S \rvert$	Cardinality of the set S
G	A partition of an index set
$\Gamma = (V, E)$	Directed graph with vertex set V and edge set E
Γ_G	Aggregation of the graph Γ w.r.t. the partition G
$\Gamma\vert_S$	Restriction of the graph $\Gamma = (V, E)$ to the set $S \subset V$

Spaces, Subsets of Spaces

$\Re^n$	Vector space of dimension n over $\Re$
$\Re^{n \times m}$	Matrix space of dimension $n \times m$ over $\Re$

$\mathfrak{R}^n_{\geq 0}, \mathfrak{R}^n_{>0}$	The sets of nonnegative and positive vectors of length n, respectively
$\mathfrak{R}^n_{>O}\left(\mathfrak{R}^n_{\geq O}\right)$	The sets of positive (nonnegative) $n \times n$ matrices
$\mathfrak{R}^n_{\succ O}\left(\mathfrak{R}^n_{\succeq O}\right)$	The sets of symmetric positive (semi-) definite $n \times n$ matrices
$\mathfrak{R}^S$	Space of functions $S \to \mathfrak{R}$, isomorphic to $\mathfrak{R}^{\lvert S \rvert}$
$V^{\perp}$ $<\cdot,\cdot>$	Orthogonal space of V w.r.t. the inner product $<\cdot,\cdot>$

Subspaces Related to Matrices

ran A	Range space of A
ker A	Kernel (null space) of A

Components of Matrices and Vectors

v_i	i-th component of the vector v
a_{ij}, $(A)_{ij}$	Component in ith row and jth column of matrix A
$a^{(j)}$	j-th column of A

Operations on Matrices

A^T	Transpose of A
$\oplus$	Direct sum
$\otimes$	Kronecker product
$\sigma(A)$	Spectrum of A
$\rho(A)$	Spectral radius of A
$sub(A)$	Subdominant radius of A
$rank\ A$	Dimension of the range space of A

Inner Products and Norms

$<\cdot,\cdot>$	Generic inner product, also canonical inner product $x^T y$
$<\cdot,\cdot>_S$	Inner product induced by the matrix S, $<x, y>_S = x^T S y$
$\lVert\cdot\rVert$	Generic norm
$\lVert\cdot\rVert^*$	Operator norm induced by $\lVert\cdot\rVert$
$\lVert\cdot\rVert_1$	l_1-norm
$\lVert\cdot\rVert_2$	l_2-norm
$\lVert\cdot\rVert_\infty$	l_∞-norm (max-norm)
$\lVert\cdot\rVert_w$	w-weighted max-norm
$\lVert\cdot\rVert_*$	Nuclear norm

Matrix Inverses

A^{-1} Algebraic inverse of A
A^{+} Moore-Penrose inverse of A
A^{+S} Moore-Penrose inverse of A w.r.t. $<\cdot,\cdot>_S$
A^{+_w} Moore-Penrose inverse of A w.r.t. $<\cdot,\cdot>_{diag(w)}$

Important Vectors and Matrices

I_n $n \times n$ identity matrix
I Identity matrix (follows from context)
$O_{m,n}$ $m \times n$ matrix of all zeros
O_n $n \times n$ matrix of all zeros
O Matrix of all zeros (dimension follows from context)
$\mathbf{1}_n$ Vector of length n of all ones
$\mathbf{1}$ Vector of all ones (dimension follows from context)
$e_n^{(i)}$ The vector $\left(e_n{}^{(i)}\right)_j = \sigma_{ij},\ i,\ j \in \underline{n}$
$e^{(i)}$ The vector $e_n^{(i)}$ (n follows from context)
$\prod$ Projector
b Right-hand side of a system of linear equations
A Coefficient matrix of a system of linear equations
x^* Solution of a system of linear equations

Dynamic Programming

M Markov decision process (MDP)
π Policy
$\prod_M$ Set of all policies for the MDP M
M_π Markov chain induced by policy $\pi \in \prod_M$
S State space
A Action space
$A(s)$ Action set in state s
p_{ij}^a Probability of state transition from i to j given action a
r_{ij}^a Reward of state transition from i to j given action a
r_{ij} Reward of state transition from i to j (action follows from context)
P Transition probability tensor
R Transition reward tensor
P^π Transition probability matrix of Markov chain M_π
R^π Transition reward matrix of Markov chain M_π
r^π Transition reward of Markov chain M_π
v^π State-value function corresponding to the policy π

q^{π}	Action-value function corresponding to the policy π
$d_t(\cdot)$, $d_t^k(\cdot)$	Temporal-difference error at t (and iteration step k)
$z_t(\cdot)$, $z_t^k(\cdot)$	Eligibility trace at t (and iteration step k)
γ	Discount rate
ε	Probability of random action in ε-greedy policy
α, β	Step-size parameters
λ	Decay-rate parameter for eligibility traces

DP for Recommendations

s_a	State associated with recommendation a
a_s	Recommendation associated with state s
$S_{A(s)}$	State set associated with all m possible recommendations
$\overline{a}$	Composite recommendation of k recommendations: $\overline{a} = (a_1, \ldots a_k)$
$S_{\overline{a}}$	State set associated with all recommendations: $S_{\overline{a}} = \{s_1\} \cup \ldots \cup \{s_k\}$
$\prod_{\overline{a}}$	Transition probabilities of recommendation states: $\prod_{\overline{a}} = \{p_{ss'}\}_{s' \in S\overline{a}}$

Multilevel

I_l^m	Level l-to-m interpolation/restriction operator
I_{l+1}^l	Interpolation (prolongation) operator from level $l+1$ to level l
I_l^{l+1}	Restriction operator from level l to level $l+1$
L	Interpolation (prolongation) operator I_1^0
R	Restriction operator I_0^1

Tensors

$\mathbf{i}$	Multi-index
$\underline{\mathbf{n}}$	Set of multi-indexes
$(\mathbf{m},\mathbf{n})$	Concatenation of multi-indexes
$\circ$	Outer vector product
$\otimes$	(Outer) tensor product
$\otimes_\delta$	Multilinear (concatenated) product over δ indexes
$\times_p$	Multilinear p-mode product with matrix (multilinear product with $\delta = 1$)
$\mathbf{i}^{(k)}$	Multi-index without k-th coordinate
$\underline{\mathbf{n}}^{(k)}$	Set of multi-indexes without k-th coordinate
$A^{(k)}$	k-Mode matricization of A

Miscellaneous

δ_{ij}	Kronecker delta
diag(v)	Diagonal matrix with components of v on the diagonal
~, $\propto$	Proportional
$f\|_X$	Restriction of the function f to the set S
$\text{argmin}_{x \in X} f(x)$	The set of minimizers of the function $f\|_X$
$\text{argmax}_{x \in X} f(x)$	The set of maximizers of the function $f\|_X$
$V \perp_{<\cdot,\cdot>} W$	V is orthogonal to W in terms of the inner product $<\cdot,\cdot>$

Chapter 1
Brave New Realtime World: Introduction

Abstract The chapter offers a general introduction to methods of realtime analytics and sets out their advantages and disadvantages as compared with conventional analytics methods, which learn only from historical data. In particular, we stress the difficulties in the development of theoretically sound realtime analytics methods. We emphasize that such online learning does not conflict with conventional offline learning but, on the opposite, both complement each other. Finally, we give some methodical remarks.

1.1 Historical Perspective

"Study cybernetics!" the Soviet author Viktor Pekelis urged his young readers, of whom I was one, in 1977 [Pek77]. But I didn't, not least because by the time I could have done so, cybernetics was no longer available as a study option. By the end of the 1970s, after more than 25 years, the wave of enthusiasm for cybernetics had finally ebbed [Pia04]. So what had gone wrong?

Cybernetics was established in the late 1940s by the American mathematician Norbert Wiener as a scientific field of study exploring the open- and closed-loop control of machines, living organisms and even entire social organizations [Wien48]. Cybernetics was also defined as the "art of control," and feedback in particular played a central role here. Its purpose was to ensure that systems do not get out of hand but instead adapt successfully to their environment. The thermostat is a classic example of a cybernetic control.

In fact the scientific benefits were immense: cybernetics brought together such diverse disciplines as control theory, neurology and information theory, and leading scientists such as John von Neumann, Warren McCulloch and Claude Shannon were involved in its development. It caused a sensation in the media. The possibilities offered by this new discipline seemed infinite: robots would take on day-to-day chores, factories would manage themselves, and computers would write poetry and compose music. More ambitiously still, from 1971 onwards the Cybersyn project in Chile headed up by the Englishman Stafford Beer sought to establish a centralized system of cybernetic economic control [Beer59]. And in the Soviet Union the OGAS project [GV81] led by pioneering cyberneticists Viktor Glushkov and Anatoly Kitov aimed to bring the entire Soviet planned economy under automated control.

A. Paprotny and M. Thess, *Realtime Data Mining: Self-Learning Techniques for Recommendation Engines*, Applied and Numerical Harmonic Analysis,
DOI 10.1007/978-3-319-01321-3_1, © Springer International Publishing Switzerland 2013

Ultimately, however, neither was successful. The Cybersyn project was brought to an abrupt end by Pinochet's coup d'etat, while OGAS was successfully blocked by Soviet bureaucrats who feared the loss of their sinecures. Yet even if these projects had been fully implemented, their immense complexity would inevitably have led to their ultimate failure. When later we see how complicated it is to properly control even far simpler systems (like our recommendation engines), we will appreciate the boldness – but also the foolhardiness – of these endeavors. For in reality, even for much more straight-forward tasks like computer chess or machine translation, cybernetics was for the moment unable to live up to its expectations.

For that reason specific elements of cybernetics began to emerge as separate research fields. Probably the most widely known of these is Artificial Intelligence (AI), which initially was hyped in much the same way as cybernetics (although lacking its scientific merit) but became discredited over time in public opinion. Like most mathematicians, I was suspicious of AI: I associated it mainly with long-haired gurus who spoke in incomprehensible sentences, always ending with the threat that robots would take over the world. In a word: cranks!

But I changed my mind after reading the classic *Artificial Intelligence: A Modern Approach* by Stuart Russell and Peter Norvig [RN02]. This book centers on the concept of an *agent* communicating with its *environment*. The authors then systematically introduce different types of agent: planning and non-planning, learning and non-learning, deterministic and stochastic, etc. An AI system encompassing a wide array of diverse fields emerges. What's more, the practical successes of AI can no longer be ignored: computer programs play better chess than grand masters, call centers work with voice control, and IBM's Watson computer recently dealt mercilessly with past champions on the American quiz show *Jeopardy*. There is still a long way to go of course: modern robots still tend to move like Martians; you have to repeat everything ten times to make voice control work, and automated Google translation is a source of constant amusement. Yet the advances are undeniable.

Michael Thess

1.2 Realtime Analytics Systems

The area of realtime data mining (*realtime analytics*, or *online* methods for short) is currently developing at an exceptionally dynamic pace. Realtime data mining systems are the counterpart of today's "classical" data mining systems (known as *offline* methods). Whereas the latter learn from historical data and then use it to deduce necessary actions (i.e., decisions), realtime analytics systems learn and act continuously and autonomously; see Fig. 1.1. (Strictly speaking, they should therefore be called realtime analytics action systems, but we will stick to the established terms.) In the vanguard of these new analytics systems are *recommendation engines* (REs). They are principally found on the Internet, where all information is available in real time and an immediate feedback is guaranteed.

Realtime analytics systems mostly use adaptive analytics methods, which means that they work incrementally: as soon as a new data set has been learned, it can be deleted. Apart from anything else, the adaptive operating principle is a practical necessity: if classic analytics methods were used, each learning step would require an analysis of all historical data. As realtime systems learn in (almost) every interaction step, the computing time would be unacceptably high.

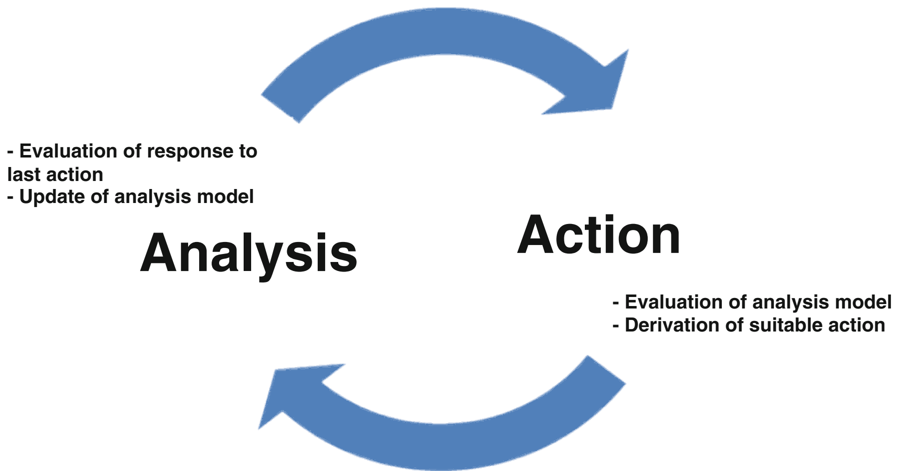


Fig. 1.1 Realtime analytics as interplay of analysis and action

Before we look at the new approach in more detail, it is worth mentioning that adaptive behavior is a mega-trend at present, not just in data mining but in many scientific disciplines. Examples include adaptive finite element methods to solve partial differential equations, adaptive control systems in production, and adaptive e-learning in education.

1.3 Advantages of Realtime Analytics Systems

Let's begin by discussing the general advantages and disadvantages of (adaptive) realtime analytics systems. The advantages are **higher quality, fewer statistical conditions required, immediate adaptation to a changed environment,** and **no storage of historical data necessary**.

The first of these, **higher quality**, is the most important. Whereas classical data mining is based exclusively on the analysis of historical data, the realtime analytics paradigm is aimed at the *interplay* of analysis and action. This requires an entirely new way of thinking and a new theoretical foundation. This foundation is reminiscent of the cybernetic approach and is based on *control theory*.

The common modeling of analysis and action is more than merely the sum of its parts. Electromagnetic waves are a graphic example of this (Fig. 1.2). These waves are based on the interplay of an electrical and a magnetic field, as defined by Maxwell's equations: the change in the electrical field over time is always associated with a spatial change in the magnetic field. Likewise, the change in the magnetic field over time is associated in turn with a spatial change in the electrical field. The result is a continuous wave of unsurpassed speed – that of light.

The same is true of the interplay of analysis and action: the results of the analysis lead to improved actions (e.g., recommendations), which are instantly applied and

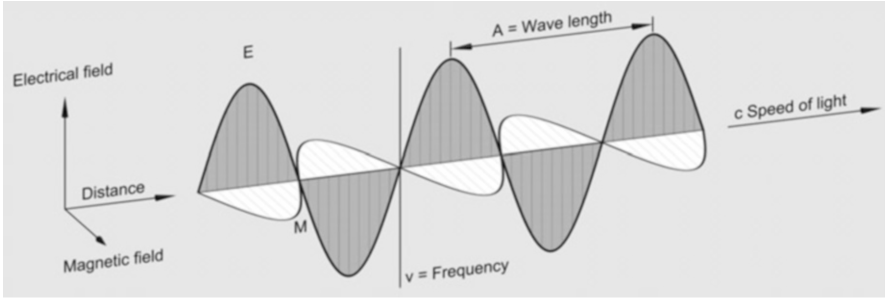


Fig. 1.2 An electromagnetic wave as interplay of an electrical and a magnetic field

thus allow an immediate refinement of the analysis. So instead of using the same model for actions for a constant period (e.g., a day or a campaign) – as in existing data mining systems – and then analyzing the results subsequently, the continuous interplay of analysis and action brings about a new *quality* of analytics systems.

The second advantage lies in the fact that adaptive analytics systems **require fewer statistical conditions**, as they explore their environment independently and can adapt to local circumstances. Adaptive finite element methods (FEMs) for solving differential equations offer a nice analogy here: whereas conventional FEM methods impose a number of regularity conditions on differential equations (shape and smoothness of the boundaries, load function space, etc.), which in practice are often difficult to verify, adaptive error estimators look independently for potential error locations such as singularities and refine the solution function grids locally. As a consequence, adaptive FEM methods, both in theory and in practice, are far more flexible and robust than conventional methods.

The third advantage, **immediate adaptation to a changed environment**, follows from the realtime concept. In classical data mining, models are first laboriously constructed from historical data and then used productively for actions for some considerable time. As a result, they are often out of date by the time they come to be used, because environmental conditions such as availability or price have changed or competitors have taken action. Realtime analytics models, on the other hand, are always up to date and adapt constantly to their changed environment.

The fourth advantage has already been described: **no storage of historical data is necessary**. Expensive data warehouses or data marts are no longer a condition for realtime analytics, making it much leaner and more flexible.

1.4 Disadvantages of Realtime Analytics Systems

Of course, adaptive analytics systems have disadvantages as compared with conventional systems too. They are **much more complex theory, restricted method classes**, and **direct feedback required**.

The significance of the first disadvantage, **more complex theory**, is often completely underestimated. Developing an adaptive algorithm for an existing task is not usually a problem. People tend to assume that the feedback loop will fix any glitches, but they are wrong. Successfully developing adaptive methods, in theory *and* in practice, is an art. For anyone who has ever come into contact with the theory of adaptive error estimators of differential equations, most conventional FEM solvers seem almost like light relief in comparison! The same is true of realtime analytics methods. As we will see in this book, their whole philosophy is far more complicated than that of conventional data mining approaches.

Incidentally, the example of light waves we looked at earlier can also be used quite effectively to illustrate the problem of developing powerful adaptive systems. Simply getting a sign wrong (even just for a moment) in the third or fourth Maxwell's equation would cause the entire electromagnetic wave literally to collapse. It is not for nothing that physicists are constantly delighted by the "beauty" of Maxwell's equations.

Philosophically, one could argue that the greater capability and robustness of adaptive behavior comes at the cost of a significantly increased workload in terms of theoretical and practical preparation. And yet it is worth it: once their development is complete, the practical advantages of adaptive realtime systems become abundantly clear.

The second disadvantage, **restricted method classes**, is related to the first. It is not merely difficult to design conventional data mining methods adaptively; in some cases, it is downright impossible. It is a fuzzy boundary: any data mining method can be made adaptive one way or another, but fundamental features of the method, such as convergence or scalability, may be lost. These losses have to be weighed up and checked in each individual case.

The third disadvantage appears self-evident: realtime analytics systems need a **direct feedback loop**; otherwise they cannot be used. In many areas, such as product placement in supermarkets for cross-selling or the mailing of brochures in optimized direct mailings, no such loop exists. There is nothing to be done about this – other than wait. And waiting helps: the introduction of new technologies is constantly extending the potential applications for technologies with realtime capability. In supermarkets, these include in-store devices such as customer terminals, voucher dispensers, or electronic price tags, which are currently revolutionizing high street retailing. But online and mobile sales channels too offer excellent feedback possibilities. The trend is being reinforced by a general move within business IT infrastructure toward service orientation (SOA, Web 2.0, etc.).

If we look at classic and adaptive analytics methods, we can see a general shift in the understanding of analytics methods. Until recently,

Rule I: The larger the available data set, the better the analysis results.

In statistical terms, that is still true of course. But increasingly, it is also the case that

Rule II: Learning by direct interaction is more important than analyzing purely historical data.

Clearly, knowing whether a customer bought milk 5 years ago, and in what combination, is less important than the information about his/her response to the milk offer in the current session. And knowing what moves a chess player made 2 years ago is much less important than understanding what tactics he/she is using in the present game.

1.5 Combining Offline and Online Analysis

Despite a trend toward realtime analytics, we have seen that both classic and adaptive analytics methods have their pros and cons. Ultimately, it is futile to try to trade one off against the other – both are necessary. Fortunately, they complement each other perfectly: historical data can be used with offline methods to create the initial analysis model so that the online system is not starting from a blank slate. Once the online system is operational, the analysis model is modified adaptively in real time. Offline analytics can still be useful when the online system is running, for integrating external transactions which cannot be communicated to the system online.

Once again, chess can offer us a useful example here: an offline chess player only learns by replaying games from chess books. By contrast, an online chess player only ever plays against living opponents. A combination of the two is ideal: replaying and learning from other people's games and at the same time keeping up with the practice.

To summarize,

Rule III: Offline and online learning complement each other organically.

For example, the recommendation engine of Sect. 12.3, the prudsys RDE, always combines both types of analytics.

1.6 Methodical Remarks

Before embarking on the actual subject of recommendation engines, we'll begin with a few preliminary remarks on methodology.

Our principle, wherever possible, is to reduce a complex problem to simple basic assumptions and then to address it in mathematical terms as fully as possible. In other words, rather than tackling a problem in its most complex form and reaching only vague conclusions, it is better to solve the simplified problem rigorously. After that, it may be possible to use the knowledge obtained to solve the problem for more

complex assumptions, etc. For example, initially we will only calculate recommendations based on the current product and only optimize them in a single step. Later on we can then discard the second and ultimately also the first requirement, by extending the method accordingly.

A good illustration of this is the discussion about infinity. Philosophers blustered about the meaning of infinity for centuries, but it was scientists in the eighteenth century working on the specific task of infinitesimal calculus who reduced the concept of infinity to epsilon estimations. Suddenly infinity was easy to understand and merely an abstraction. This viewpoint had become generally established when at the end of the nineteenth century, while working on his continuum theory, the German mathematician Georg Cantor dropped the bombshell that infinity does in fact exist and can even be used in calculations. After much debate, this ultimately led to a greater understanding of the concept of infinity, which then found expression in philosophy too.

Conversely, however, it is often argued that complex data mining algorithms are not worthwhile because they are difficult to master. It is better, so the argument goes, to use a simple algorithm and to provide large data sets. A classic example of this is Google, which successfully uses a relatively simple search algorithm on vast data sets. There is also an example of this in the area of recommendation engines: Amazon's item-to-item collaborative filtering (ITI CF). Quite simple in mathematical terms, it has displaced the previously used collaborative filtering, which was very complex and poorly scaled.

Although this view seems perfectly pragmatic, and in the cases described here has been successful too, it is nevertheless shortsighted. Generally speaking, one could argue that people would still be living in caves if they had followed this way of thinking. But there are also some very specific reasons for not adopting this approach: most companies simply do not have enough data to generate meaningful recommendations in this way. Nowadays even a small bookseller can in principle offer the same millions of books as Amazon – so ITI CF would only generate recommendations for a small fraction of its books. More sophisticated methods, like content-based recommendations or, better still, the hierarchical approach described in Chap. 6, are needed to resolve this problem. Moreover, the rapidly accelerating pace of the Internet world, with its constantly changing products, prices, ratings, competitors, and business models, is making realtime-capable recommendation systems indispensable.

So the transition to more complex recommendation methods is unavoidable. That does not mean, however, that all steps have to be perfect and mathematically proven; practice has every right to rush on ahead of theory. This may seem like a contradiction of the methodology we described earlier, but it isn't. If we look at shell theory in mechanics, for example, it is still not always capable of the rigorous numerical calculation of the deformation of even simple bodies like a cylinder. Yet supercomputers can successfully simulate the deformation of an entire car in crash situations. Even if theoretically it is not entirely rigorous, should scientists wait for another 100 years until shell theory is sufficiently mature before performing crash simulations? Should thousands more people be allowed to lose

their lives in the meantime before the go-ahead is given for "theoretically rigorous" simulation? Of course not.

In the case of realtime recommendation engines too, there are still many questions left unanswered. We will address these head on. We will also always clearly emphasize empirical assumptions such as the Markov property or probability assumptions. For one thing, it would be naive and wrong to seek to derive everything in science purely in mathematical terms and to eliminate the necessary empirical component (expressions such as "scientifically deduced" should always sound alarm bells). And for another, it is important to understand about assumptions so that in individual cases, the applicability of the recommendation method can be verified in practice. That is why a methodologically rigorous procedure as described in the introduction is essential: a **stepwise approach** to a self-learning recommendation engine.

It is also clear that new ideas and methods, such as reinforcement learning for recommendation engines as described here, usually need to mature for years before they are suitable for practical application. The initial euphoria, especially when everything seems to be "mathematically sound" and proven, is usually followed by disillusionment in practice, with countless setbacks. But practical problems should also be regarded as an opportunity, because tackling them often leads to the most exciting theoretical advances. And when the method is finally ready for commercial application, this is often followed by a dramatic breakthrough.

Finally, let's pick up once more on some critical points regarding the general use of recommendation engines (and of realtime analytics). This brings us back first of all to the "cybernetic control" of the Soviet planned economy envisaged by the OGAS project. Soviet economists blamed its failure on its inconsistent and piecemeal implementation, and this has been a constant source of regret. Even now the legend still lingers on in Russia that the Soviet economy would have developed differently if only OGAS had been implemented consistently. As a consequence, the "theory of economic control" – now opportunistically extended to include a synthesis of market and planned economy – is undergoing a real revival in the search for a "third way." Ultimately, however, this is more about reinvigorating the failed concept of the planned economy. The growing importance of cybernetics in modern Russian economics is clearly a retrograde step (which does not mean to say that the use of cybernetic approaches in economics is inherently wrong).

As we mentioned earlier, it is true that OGAS was not implemented correctly. But it is also true that the entire concept was misguided. For one thing, predicting key indicators in economics is difficult over the long term, and predicting an entire economic system is impossible. The idea of controlling it completely is even more absurd. Not to mention the fact that in a (market) economy, the state can never set out to exercise control over the economy.

For that reason, the "father of cybernetics" Norbert Wiener excluded economics and sociology entirely from the remit of cybernetics as a highly mathematized science [Wien64]:

> The success of mathematical physics led the social scientist to be jealous of its power without quite understanding the intellectual attitudes that had contributed to this power. The use of mathematical formulae had accompanied the development of the natural sciences and become the mode in the social sciences. Just as primitive peoples adopt the Western modes of denationalized clothing and of parliamentarism out of a vague feeling that these magic rites and vestments will at once put them abreast of modern culture and technique, so the economists have developed the habit of dressing up their rather imprecise ideas in the language of the infinitesimal calculus...
>
> Difficult as it is to collect good physical data, it is far more difficult to collect long runs of economic or social data so that the whole of the run shall have a uniform significance...
>
> Under the circumstances, it is hopeless to give too precise a measurement to the quantities occurring in it. To assign what purports to be precise values to such essentially vague quantities is neither useful nor honest, and any pretense of applying precise formulae to these loosely defined quantities is a sham and a waste of time.

From a modern perspective, Norbert Wiener's assessment now seems too pessimistic. Yet it highlights the difficulties inherent in these disciplines, and the role of mathematical theories in economics in particular is still a subject for debates today, usually each time after the Nobel Prize for economics is announced.

Recommendation engines are used primarily in retail, which also has a complex environment. Where data mining is used in industrial quality assurance, for example, environmental conditions are relatively constant (temperature and lighting conditions in the factory, output speed, etc.), whereas in retail they are changing all the time. We have already touched on this as an argument in support of *realtime* analytics. Control is even more difficult. Empirical evidence shows that recommendation engines change user behavior significantly. The skill, however, is to convert this into increased sales. In many cases, the use of REs simply leads to the purchase of alternative products, and this can even result in down-selling and a loss of sales. We will look in detail at the subject of down-selling in mathematical terms in Chap. 5.

Fortunately, user behavior in the areas in which REs are used can generally be predicted fairly reliably, albeit within strict limits in terms of time and content. And, unlike the case with economics as described above, the primary and realistic aim of REs is to control and direct user behavior. As such, the use of realtime methods makes absolute sense. Nevertheless, to avoid unrealistic expectations, it is important to stress the complexity of the retail environment (unlike the earlier example of the electromagnetic wave). For that reason, having rigorous methods of gauging success is of paramount importance, so we have devoted an entire chapter – Chap. 11 – to this subject (although this does not relate solely to realtime analytics systems).

Finally, we mention that the book covers different mathematical disciplines that sometimes require complex notations. To make the notation more understandable and to reduce possible confusion, we included a summary of notation at the beginning of the book. Nevertheless, the authors could not avoid that some symbols are used for different representations. In these cases, the meaning should be clear from the context.

Chapter 2
Strange Recommendations? On the Weaknesses of Current Recommendation Engines

Abstract Currently, most approaches to recommendation engines focus on traditional techniques such as collaborative filtering, basket analysis, and content-based recommendations. Recommendations are considered from a prediction point of view only, i.e., the recommendation task is reduced to the prediction of content that the user is going to select with highest probability anyway. In contrast, in this chapter we propose to view recommendations as control-theoretic problem by investigating the interaction of analysis and action. The corresponding mathematical framework is developed in the next chapters of the book.

2.1 Introduction to Recommendation Engines

Recommendation engines (REs) for customized recommendations have become indispensable components of modern web shops. REs offer the users additional content so as to better satisfy their demands and provide additional buying appeals.

There are different kinds of recommendations that can be placed in different areas of the web shop. "Classical" recommendations typically appear on product pages. Visiting an instance of the latter, one is offered additional products that are suited to the current one, mostly appearing below captions like "Customers who bought this item also bought" or "You might also like." Since it mainly respects the currently viewed product, we shall refer to this kind of recommendation, made popular by Amazon, as product recommendation. Other types of recommendations are those that are adapted to the user's buying behavior and are presented in a separate area as, e.g., "My Shop," or on the start page after the user has been recognized. These provide the user with general but personalized suggestions with respect to the shop's product range. Hence, we call them personalized recommendations.

Further recommendations may, e.g., appear on category pages (best recommendations for the category), be displayed for search queries (search recommendations), and so on. Not only products but also categories, banners, catalogs,

A. Paprotny and M. Thess, *Realtime Data Mining: Self-Learning Techniques for Recommendation Engines*, Applied and Numerical Harmonic Analysis,
DOI 10.1007/978-3-319-01321-3_2, © Springer International Publishing Switzerland 2013

authors (in book shops), etc., may be recommended. Even more, as an ultimate goal, recommendation engineering aims at a total personalization of the online shop, which includes personalized navigation, advertisements, prices, mails, and text messages. The amount of prospects is seemingly inexhaustible. For the sake of simplicity, however, this book will be restricted to mere product recommendations – we shall see how complex even this task is.

Recommendation engineering is a vivid field of ongoing research. Hundreds of researchers, predominantly from the USA, are tirelessly devising new theories and methods for the development of improved recommendation algorithms. Why, after all?

Of course, generating intuitively sensible recommendations is not much of a challenge. To this end, it suffices to recommend top sellers of the category of the currently viewed product. The main goal of a recommendation engine, however, is an increase of the web shop's revenue (or profit, sales numbers, etc.). Thus, the actual challenge consists in recommending products that the user actually visits *and* buys, while, at the same time, preventing down-selling effects, so that the recommendations do not simply stimulate buying substitute products and, therefore, in the worst case, even lower the shop's revenue.

This brief outline already gives a glimpse at the complexity of the task. It is even worse: many web shops, especially those of mail-order companies (let alone bookshops), by now have hundreds of thousands, even millions, of different products on offer. From this giant amount, we then need to pick the right ones to recommend! Furthermore, through frequent special offers, changes of the assortment as well as – especially in the area of fashion – prices are becoming more and more frequent. This gives rise to the situation that good recommendations become outdated soon after they have been learned. A good recommendation engine should hence be in a position to learn in a highly dynamic fashion. We have thus reached the main topic of the book – adaptive behavior.

We abstain from providing a comprehensive exposition of the various approaches to and types of methods for recommendation engines here and refer to the corresponding literature, e.g., [BS10, JZFF10, RRSK11]. Instead, we shall focus on the crucial weakness of almost all hitherto existing approaches, namely, the lack of a control theoretical foundation, and devise a way to surmount it.

2.2 Weaknesses of Current Recommendation Engines and How to Overcome Them

Recommendation engines are often still wrongly seen as belonging to the area of classical data mining. In particular, lacking recommendation engines of their own, many data mining providers suggest the use of basket analysis or clustering techniques to generate recommendations. Recommendation engines are currently one of the most popular research fields, and the number of new approaches is

also on the rise. But even today, virtually all developers rely on the following assumption:

> If the products (or other content) proposed to a user are those which other users with a comparable profile in a comparable state have chosen, then those are the best recommendations.

Or in other words:

Approach I: What is recommended is statistically what a user would very probably have chosen in any case, even without recommendations.

This reduces the subject of recommendations to a statistical analysis and modeling of user behavior. We know from classic cross-selling techniques that this approach works well in practice.

Yet it merits a more critical examination. In reality, a pure analysis of user behavior does not cover all angles:

1. **The effect of the recommendations is not taken into account:** If the user would probably go to a new product anyway, why should it be recommended at all? Wouldn't it make more sense to recommend products whose recommendation is most likely to change user behavior?
2. **Recommendations are self-reinforcing:** If only the previously "best" recommendations are ever displayed, they can become self-reinforcing, even if better alternatives may now exist. Shouldn't new recommendations be tried out as well?
3. **User behavior changes:** Even if previous user behavior has been perfectly modeled, the question remains as to what will happen if user behavior suddenly changes. This is by no means unusual. In web shops, data often changes on a daily basis: product assortments are changed, heavily discounted special offers are introduced, etc. Would it not be better if the recommendation engine were to learn continually and adapt flexibly to the new user behavior?

There are other issues too. The above approach does not take the sequence of all of the subsequent steps into account:

4. **Optimization across all subsequent steps:** Rather than only offering the user what the recommendation engine considers to be the most profitable product in the next step, would it not be better to choose recommendations with a view to optimizing sales across the most probable sequence of all subsequent transactions? In other words, even to recommend a less profitable product in some cases, if that is the starting point for more profitable subsequent products? To take the long-term rather than the short-term view?

These points all lead us to the following conclusion, which we mentioned right at the start – while the conventional approach (Approach I) is based solely on the

analysis of historical data, good recommendation engines should model the interplay of analysis and action:

Approach II: Recommendations should be based on the interplay of analysis and action.

In the next chapter, we will look at one such approach of control theory – reinforcement learning. First though we should return to the question of why the first approach still dominates current research.

Part of the problem is the limited number of test options and data sets. Adopting the second approach requires the algorithms to be integrated into realtime applications. This is because the effectiveness of recommendation algorithms cannot be fully analyzed on the basis of historical data, because the effect of the recommendations is largely unknown. In addition, even in public data sets, the recommendations that were actually made are not recorded (assuming recommendations were made at all). And even if recommendations had been recorded, they would mostly be the same for existing products because the recommendations would have been generated manually or using algorithms based on the first approach.

This trend was further reinforced by the Netflix competition [Net06]. The company Netflix offered a prize of 1 million dollars to any research team which could increase the prediction accuracy of the Netflix algorithm by 10 % using a given set of film ratings. The Netflix competition was undoubtedly a milestone in the development of recommendation systems, and its importance as a benchmark cannot be overstated. But it pushed the development of recommendation algorithms firmly in the direction of pure analytics methods based on the first approach.

So we can see that on practical grounds alone, the development of viable recommendation algorithms is very difficult for most researchers. However, the number of publications in the professional literature treating recommendations as a control problem and adopting the second approach has been on the increase for some time.

As a further boost to this way of thinking, prudsys AG chose the theme of recommendation algorithms for its 2011 Data Mining Cup, one of the world's largest data mining competitions [DMC11]. The first task related to the classical problem of pure analysis, based however on transaction data for a web shop. But the second task looked at realtime analytics, asking participants to design a recommendation program capable of learning and acting in realtime via a defined interface. The fact that over 100 teams from 25 countries took part in the competition shows the level of interest in this area.

A further example of new realtime thinking is the RECLAB project of RichRelevance, another vendor of recommendation engines. Under the slogan "If you can't bring the data to the code, bring the code to the data," it offers researchers to submit their recommendation code to the lab. There, new algorithms can be tested in personalization applications on live retail sites.

Chapter 3
Changing Not Just Analyzing: Control Theory and Reinforcement Learning

Abstract We give a short introduction to reinforcement learning. This includes basic concepts like Markov decision processes, policies, state-value and action-value factions, and the Bellman equation. We discuss solution methods like policy and value iteration methods, online methods like temporal-difference learning, and state fundamental convergence results.

It turns out that RL addresses the problems from Chap. 2. This shows that, in principle, RL is a suitable instrument for solving all of these problems.

We have described how a good recommendation engine should learn step by step by interaction with its environment. It is precisely this task that reinforcement learning (RL), one of the most fascinating disciplines of machine learning, addresses. RL is used among other things to control autonomous systems such as robots and also for self-learning games like backgammon or chess. And as we will see later, despite all problems, RL turns out to be an excellent framework for recommendation engines.

In this chapter, we present a brief introduction to reinforcement learning before in the subsequent chapter we consider its application to REs. For a detailed introduction, we refer you to the standard work "Reinforcement Learning – An Introduction" by Richard Sutton and Andrew Barton [SB98], from which some of the figures in this chapter have been taken. Especially, following [SB98] for reasons of a unified treatment, we will unify the model-based approach, the dynamic programming, as well as the model-free approach, the actual reinforcement learning, under the term "reinforcement learning."

A. Paprotny and M. Thess, *Realtime Data Mining: Self-Learning Techniques for Recommendation Engines*, Applied and Numerical Harmonic Analysis,
DOI 10.1007/978-3-319-01321-3_3, © Springer International Publishing Switzerland 2013

3.1 Modeling

RL was based originally on methods of dynamic programming (DP, the mathematical theory of optimal control), albeit that in machine learning, the theories and terminology have since been developed beyond DP. Central to this – as is usual in AI – is the term *agent*. Figure 3.1 shows the interaction between agent and environment in reinforcement learning.

The agent passes into a new *state* (s), for which it receives a *reward* (r) from the environment, whereupon it decides on a new *action* (a) from the admissible *action set* for s ($A(s)$), by which in most cases it learns, and the environment responds in turn to this action, etc. In such cases, we differentiate between *episodic tasks*, which come to an end (as in a game), and *continuing tasks* without any end state (such as a service robot which moves around indefinitely). The goal of the agent consists in selecting the actions in each state so as to maximize the sum of all rewards over the entire episode. The selection of the actions by the agent is referred to as its *policy* π, and that policy which results in maximizing the sum of all rewards is referred to as the *optimal policy*.

Example 3.1 As the first example for RL, we can consider a robot, which is required to reach a destination as quickly as possible. The states are its coordinates, the actions are the selection of the direction of travel, and the reward at every step is -1. In order to maximize the sum of rewards over the entire episode, the robot must achieve its goal in the fewest possible steps. ■

Example 3.2 A further example is chess once again, where the positions of the pieces are the states, the moves are the actions, and the reward is always 0 except in the final position, at which it is 1 for a win, 0 for a draw, and -1 for a loss (this is what we call a *delayed reward*). ■

Example 3.3 A final example, to which we will dedicate more intensive study, is recommendation engines. Here, for instance, the product detail views are the states, the recommended products are the actions, and the purchases of the products are the rewards. ■

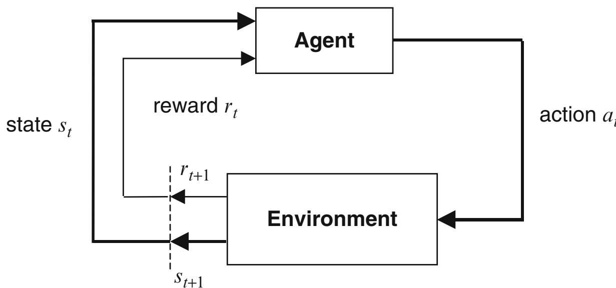


Fig. 3.1 The interaction between agent and environment in RL

3.2 Markov Property

In order to keep the complexity of determining a good (most nearly optimal) policy within bounds, in most cases, it is assumed that the RL problem satisfies what is called the *Markov* property:

> **Assumption 3.1 (Markov property):** In every state, the selection of the best action depends only on this current state and not on transactions preceding it.

A good example of a problem which satisfies the Markov property is once again the game of chess. In order to make the best move in any position, from a mathematical point of view, it is totally irrelevant how the position on the board was reached (though when playing the game in practice, it is generally helpful). On the other hand, it is important to think through all possible subsequent transactions for every move (which of course in practice can be performed only to a certain depth of analysis) in order to find the optimal move.

Put simply, we have to work out the future from where we are, irrespective of how we got here. This allows us to reduce drastically the complexity of the calculations. At the same time, we must of course check each model to determine whether the Markov property is adequately satisfied. Where this is not the case, a possible remedy is to record a certain limited number of preceding transactions (generalized Markov property; see Chap. 10) and to extend the definition of the states in a general sense.

Provided the Markov property is now satisfied (*Markov decision process – MDP*), the policy π depends solely on the current state, that is, $a = \pi(s)$. RL is now based directly on the DP methods for solution of the *Bellman* equation. This involves assigning to each policy π an *action-value function* $q^{\pi}(s,a)$ which assigns for each state s and for all the permissible actions a for that state the expected value of the cumulative rewards throughout the remainder of the episode. We shall refer to this magnitude as the *expected return* R:

$$R_t := r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{3.1}$$

where t denotes the current time step and $\gamma \in [0,1]$ the rate for discounting future rewards. For $\gamma = 0$, the agent acts myopically in that it seeks to maximize only the immediate reward r_{t+1}. With γ increasing, the agent acts in a more and more long-term oriented fashion in that future rewards make a larger contribution. If $\gamma < 1$, the infinite series always converges to finite value, provided that the sequence $\{r_k\}$ is bounded.

If then for any two actions $a, b \in A(s): \; q^{\pi}(s,a) < q^{\pi}(s,b)$, then b ensures a higher return than a. Therefore, the policy $\pi(s)$ should prefer the action b to the action a, but we will come to that in a minute.

By analogy with Fig. 1.1, we see that while the update of the action-value function $q^\pi(s, a)$ corresponds to the analysis, the policy $\pi(s)$ determines the selection of the correct action. The action-value function thus constitutes our analysis model, and updating it is what we call learning.

We describe firstly the implementation of the policy for an action-value function, that is, the selection of the action.

3.3 Implementing the Policy: Selecting the Actions

The basic policy for an action-value function is what we call the *greedy policy*, which in every state s selects the action a which is the one that maximizes $q^\pi(s, a)$:

$$\pi(s) = \underset{a \in A(s)}{\arg\max}\, q^\pi(s, a).$$

So in every state, the action with the largest action value is selected, or if there are several having that value, then one of them. This is the most obvious selection.

In order, however, to avoid always restricting ourselves to *exploiting* existing knowledge but rather to allow new actions to be *explored*, in addition to the *deterministic* greedy policy, *stochastic* policies are also used. A stochastic policy $\pi(s, a)$ specifies for every state s and every action $a \in A(s)$ the probability of selection of a. So while in every state the deterministic policy always makes a unique selection of the action, the stochastic policy permits the selection of different actions with specified probabilities.

In the simplest case of a stochastic policy, at most steps, we select the best action (greedy policy), but from time to time – that is, with the probability ε – we select an action $a \in A(s)$ at random. We call the resulting policy the ε *-greedy policy*.

The combination of exploitation and exploration can also be performed on a sliding basis, that is, the frequency of selection of an action increases with its action value. This is done by means of the *softmax policy*. For this, the recommendations are calculated at every step in accordance with a probability distribution such as the Boltzmann distribution:

$$\pi(s, a) = \frac{e^{\frac{q(s,a)}{\tau}}}{\sum\limits_{b \in A(s)} e^{\frac{q(s,b)}{\tau}}}$$

where τ is the "temperature parameter." A high value for τ asymptotically leads to an even distribution of all actions (exploration); a low value for τ leads to selection of the best actions (exploitation). In general, the softmax policy leads to better results than the ε -greedy policy, but both in theory and in practice, it is more difficult to handle.

The correct interplay between exploitation and exploration is one of the central issues in RL. Here again, chess provides a useful example: in most positions, we

make what we think are the best moves. From time to time, however, we try out a new move in a known position – even Kasparov does that. In doing so, we also solve the problem of self-reinforcing recommendations (Chap. 2, Problem 2) suffered by conventional recommendation engines.

3.4 Model of the Environment

Before finally addressing the Bellman equation, we still need a model of the environment, which is given by the transition probabilities and rewards. Let $p^a_{ss'}$ be the transition probabilities from the state s into state s' as a result of the action a and $r^a_{ss'}$ the corresponding transition rewards. Put another way, if in the state s the action a is performed, $p^a_{ss'}$ gives the probability of passing into state s' and $r^a_{ss'}$ the reward obtained as a result of the transition to s'. In particular,

$$\sum_{s'} p^a_{ss'} = 1, \tag{3.2}$$

that is, when performing the action a in state s, the sum of the transition probabilities over all possible subsequent states s' equals 1, because we must of necessity pass into one of those states.

Example 3.4 Let us consider a fictional car which is being driven along a mountainous road, mostly uphill. Let the states be the speeds $s_1 = 80\ km/h$, $s_2 = 90\ km/h$, $s_3 = 100\ km/h$, the actions $a_0 = \text{noaccelerator}$, $a_1 = \text{accelerator}$, and the rewards always the speed values in the subsequent state, that is, we want to get to our goal as quickly as possible (Fig. 3.2).

This gives us for the rewards:

$$r^a_{ss_i} = v_i,$$

that is, the value of the speed v_i in the subsequent state s_i independent of the state s and the action a. If, for instance, the driver in the state s_2 presses the accelerator, that is, action a_1, and passes into the state s_3, then the reward is $r^{a_1}_{s_2 s_3} = v_3 = 100$.

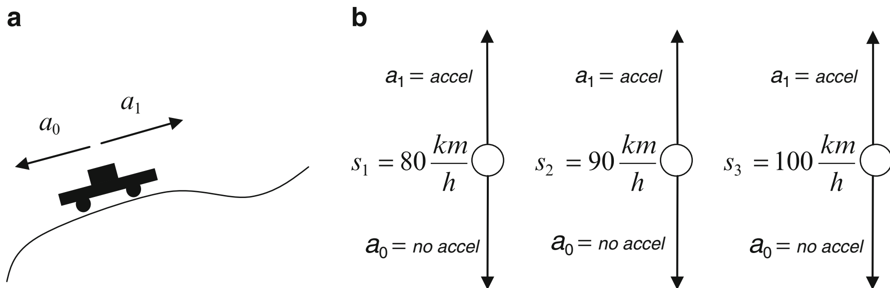


Fig. 3.2 A car with three speed states and two control actions

The action “no acceleration” generally leads to reduced speed; however, on level or downhill stretches, it can lead to constant or even increased speed. For instance, for s_2, we can specify

$$p_{s_2 s_1}^{a_0} = 0.75, \; p_{s_2 s_2}^{a_0} = 0.2, \; p_{s_2 s_3}^{a_0} = 0.05.$$

So if we drive at 90 km/h and do not accelerate, the probability that the speed will reduce to 80 km/h is 75 %, that it will remain at 90 km/h is 20 %, and that it will increase to 100 km/h is 5 %. Remember that in accordance with (3.2), the probabilities must add up to 100 %. Similarly, for the remaining states s_1 and s_3, we can define

$$\begin{aligned} p_{s_1 s_1}^{a_0} &= 0.7, \; p_{s_1 s_2}^{a_0} = 0.3, \\ p_{s_3 s_2}^{a_0} &= 0.9, \; p_{s_3 s_3}^{a_0} = 0.1. \end{aligned}$$

The action “acceleration” of course has precisely the inverse effect. We start once again with the specification for s_2:

$$p_{s_2 s_1}^{a_1} = 0.1, \; p_{s_2 s_2}^{a_1} = 0.2, \; p_{s_2 s_3}^{a_1} = 0.7.$$

So if we drive at 90 km/h and accelerate, the probability that the speed will increase to 100 km/h is 70 %, that it will remain at 90 km/h is 20 %, and that it will decrease to 80 km/h is 10 %. Similarly, for the remaining states s_1 and s_3, we can define

$$\begin{aligned} p_{s_1 s_1}^{a_1} &= 0.3, \; p_{s_1 s_2}^{a_1} = 0.7, \\ p_{s_3 s_2}^{a_1} &= 0.1, \; p_{s_3 s_3}^{a_1} = 0.9. \end{aligned}$$

In so doing, we have adequately described our environment. ■

3.5 The Bellman Equation

We first define an MDP as a quadruplet $M := (S, A, P, R)$ of the state and action spaces S and A, the transition probabilities P, and rewards R. Please note that the Markov property need not be explicitly stipulated to hold, since it implicitly follows from the given representations of P and R.

Each policy $\pi(s, a)$ induces a *Markov chain* (MC), which is characterized by the tuple $M_\pi := (S, P^\pi)$, where $P^\pi = (p_{s,s'}^\pi)_{s,s' \in S}$ denote the transition probabilities that result from following the policy $\pi(s, a)$:

$$p_{ss'}^{\pi} = \sum_{a \in A(s)} \pi(s,a) p_{ss'}^{a}. \tag{3.3}$$

In other words, by following a policy, we obtain a sequence of states which is generated by a Markov chain. The latter follows from the simple fact that the probability of a transition from a state to another state under a given policy $\pi(s, a)$ depends exclusively on the current state s and not on its predecessors.

We thus arrive at the *Bellman equation*. For the discrete case, the action-value function for each state s and each action $a \in A(s)$ satisfies

$$q^{\pi}(s,a) = \sum_{s'} p_{ss'}^{a} \left[r_{ss'}^{a} + \gamma v^{\pi}\left(s'\right) \right], v^{\pi}(s) = \sum_{a} \pi(s,a) q^{\pi}(s,a). \tag{3.4}$$

$v^{\pi}(s)$ is what we call the *state-value function*, which assigns to each state s the expected cumulative reward, that is, the expected return. The state-value function and action-value function are thus related and can be converted from one into the other (provided the model of the environment is known).

At this point, we should further mention that the Bellman equation (3.4) represents the discrete counterpart of the Hamilton-Jacobi-Bellman (HJB) differential equation, a fact which will become significant in Chap. 6 of the hierarchical methods. For a detailed discussion of the HJB equation and the relation to other formulations, we refer to [Mun00].

At the first glance, the Bellman equation appears rather complex, but it is not so difficult to understand. Let us first consider the case $\gamma = 0$, that is, taking into account only the immediate reward. The Bellman equation (3.4) then takes the following simplified form:

$$q^{\pi}(s,a) = \sum_{s'} p_{ss'}^{a} r_{ss'}^{a}. \tag{3.5}$$

The expected return in the state s on taking the action a therefore equals the sum (over all possible subsequent states s') for all products of the probability $p_{ss'}^{a}$ of passing into the subsequent state s' and the reward $r_{ss'}^{a}$ obtained by doing so.

For $\gamma > 0$, in addition to the immediate reward $r_{ss'}^{a}$, the expected additional return over all subsequent transactions, which is $\gamma v^{\pi}(s')$, must now be added for the transition to the subsequent state s' ("chain optimization"); see Fig. 3.3a. In general, there are always two possibilities of reward in RL: the immediate reward or the indirect reward via the fact that it leads to an attractive subsequent state (or both).

The state-value function can in turn be determined using (3.4) from the action-value function, namely, as the sum (over all actions a permissible in s) of the product of the probability of the selection of the action a by the existing policy and its expected action value (Fig. 3.3b). By substituting the state-value function v^{π} into (3.4), we can write it similarly to the Bellman equation (Fig. 3.4a):

$$q^{\pi}(s,a) = \sum_{s'} p_{ss'}^{a} \left[r_{ss'}^{a} + \gamma \sum_{a'} \pi\left(s', a'\right) q^{\pi}\left(s', a'\right) \right]. \tag{3.6}$$

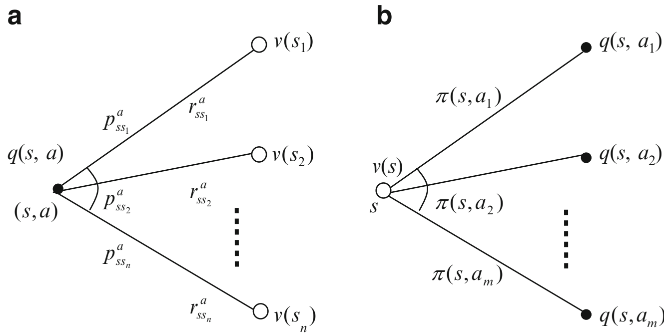


Fig. 3.3 Relationship between the state-value and action-value functions v and q

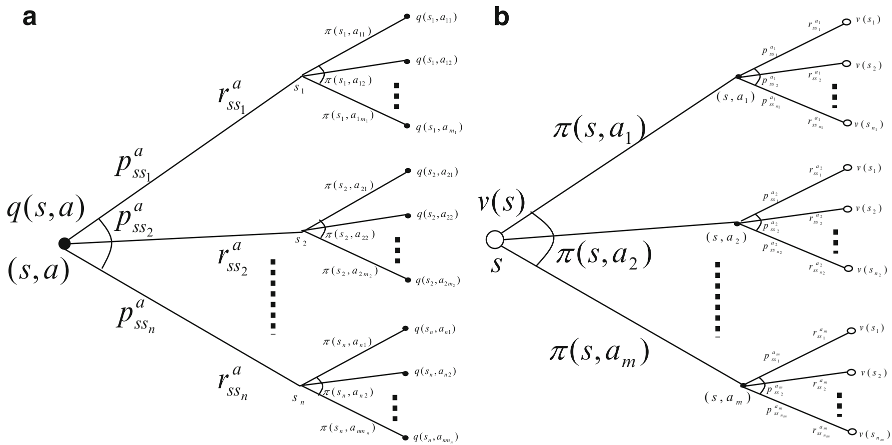


Fig. 3.4 Bellman equation for both the action-value function q and the state-value function v

Since to $q^\pi(s', a')$, too, a Bellman equation in accordance with (3.6) applies, with new subsequent states s'' and actions a$''$ (which in part can contain the original s and a!), the solution of (3.6) – unlike the 1-step special case (3.5) – is usually a more complex undertaking. This reflects the fact that we are taking into account the entire chain of subsequent transactions by which we address the Problem 4 in Chap. 2. Since our transition probabilities $p^a_{ss'}$ in fact depend on the action a, we learn directly from this and thus also solve Problem 1 in Chap. 2.

For the sake of completeness, we should also mention that in (3.4), we can conversely eliminate the action-value function q^π. We then obtain the Bellman equation for the state-value function (Fig. 3.4b):

$$v^\pi(s) = \sum_a \pi(s,a) \sum_{s'} p^a_{ss'} \left[r^a_{ss'} + \gamma v^\pi\left(s'\right) \right]. \tag{3.7}$$

This is an even clearer form, since the state-value function $v^\pi(s)$ depends only on the state s, unlike the action-value function $q^\pi(s, a)$, which additionally depends on the action a. We will, however, mainly work with the action-value function, since we need it for the model-free case, which is of practical importance (and to which we have yet to come), where it cannot be converted directly into the state-value function (since in the model-free case $p_{ss'}^a$ and $r_{ss'}^a$ are not explicitly known).

After so many abstract explanations, we shall seek to illustrate the Bellman equation using our simple example of a car.

Example 3.5 Let us now return to our example of a car, and calculate it exemplarily for the Bellman equation with the discount parameter $\gamma = 0.5$ for the policy which in each of the three states performs the action a_0, that is, the one where the accelerator is never pressed.

From (3.6), we then obtain for the first state s_1:

$$\begin{aligned} q^\pi(s_1, a_0) &= p_{s_1 s_1}^{a_0}\left[r_{s_1 s_1}^{a_0} + 0.5 q^\pi(s_1, a_0)\right] + p_{s_1 s_2}^{a_0}\left[r_{s_1 s_2}^{a_0} + 0.5 q^\pi(s_2, a_0)\right] \\ &= 0.7 \cdot [80 + 0.5 q^\pi(s_1, a_0)] + 0.3 \cdot [90 + 0.5 q^\pi(s_2, a_0)]. \end{aligned}$$

Similarly, we obtain for the second state s_2:

$$\begin{aligned} q^\pi(s_2, a_0) &= p_{s_2 s_1}^{a_0}\left[r_{s_2 s_1}^{a_0} + 0.5 q^\pi(s_1, a_0)\right] + p_{s_2 s_2}^{a_0}\left[r_{s_2 s_2}^{a_0} + 0.5 q^\pi(s_2, a_0)\right] \\ &\quad + p_{s_2 s_3}^{a_0}\left[r_{s_2 s_3}^{a_0} + 0.5 q^\pi(s_3, a_0)\right] \\ &= 0.75 \cdot [80 + 0.5 q^\pi(s_1, a_0)] + 0.2 \cdot [90 + 0.5 q^\pi(s_2, a_0)] \\ &\quad + 0.05 \cdot [100 + 0.5 q^\pi(s_3, a_0)] \end{aligned}$$

and for s_3:

$$\begin{aligned} q^\pi(s_3, a_0) &= p_{s_3 s_2}^{a_0}\left[r_{s_3 s_2}^{a_0} + 0.5 q^\pi(s_3, a_0)\right] + p_{s_3 s_3}^{a_0}\left[r_{s_3 s_3}^{a_0} + 0.5 q^\pi(s_3, a_0)\right] \\ &= 0.9 \cdot [90 + 0.5 q^\pi(s_2, a_0)] + 0.1 \cdot [100 + 0.5 q^\pi(s_3, a_0)]. \end{aligned}$$

We thus have a system of three equations with three unknowns, the action values. Its solution yields

$$q^\pi(s_1, a_0) \approx 166, \quad q^\pi(s_2, a_0) \approx 167, \quad q^\pi(s_3, a_0) \approx 174.$$

So far, this is sensible: since, with no acceleration, the states s_1 and s_2 almost always lead to the state s_1, they also obtain largely the same expected return. Without acceleration, the state s_3 almost always leads to the state s_2 and therefore has a higher expected return. The fact that $q^\pi(s_2, a_0)$ is somewhat higher than $q^\pi(s_1, a_0)$ is due to

that fact that with no acceleration, we still remain in the same state or even pass into the next highest state in rare cases. The result is therefore plausible. ■

3.6 Determining an Optimal Solution

The question remains of how to determine an optimal solution to the Bellman equation, since we neither know its action-value function $q^{\pi}(s, a)$ nor its policy $\pi(s, a)$. A solution to this is provided by the *policy iteration* method from dynamic programming, which in a generalized form can be used as a central tool in RL and generally for REs.

Policy iteration is based on the following approach: starting with an arbitrary initial policy π_0, the action-value function q^i corresponding to the current policy π_i is computed by solving the Bellman equation (3.6) in every step $i = 0,\ldots, n$ (the solution method will be described in Sect. 3.9.4). After this, we determine a greedy policy corresponding to the action-value function q^i, that is,

$$\pi_{i+1}(s) = \arg\max_{a \in A(s)} q^i(s, a).$$

In plain English, π_{i+1} is taken to be a policy which in every state s selects one of the actions a, such as to maximize $q^i(s, a)$. For π_{i+1}, the action-value function q^{i+1} is then calculated in turn, and so on. This then yields a sequence of policies and action-value functions:

$$\pi_0 \to q^0 \to \pi_1 \to q^1 \to \pi_2 \to q^2 \to \ldots.$$

It can be shown that after a finite number of iterations, this process terminates with the optimal policy π^* and corresponding action-value function q^*, which satisfy

$$q^*(s, a) = \max_{\pi} \quad q^{\pi}(s, a), \quad \forall s \in S, \forall a \in A(s).$$

Example 3.6 For our example of the car with $\gamma = 0.5$ as above, policy iteration yields – not surprisingly – the optimal policy π^*, which stipulates that in each of the three states, the action a_1 be performed, that is, to always accelerate. The associated action values are

$$q^*(s_1, a_1) \approx 182, \quad q^*(s_2, a_1) \approx 194, \quad q^*(s_3, a_1) \approx 198,$$

and are thus all greater than those of the non-acceleration policy considered in the last example, which is incidentally the least successful among all policies.

What will happen if we decrease the reward of the third state? We might, for instance, be pulled over by the police for exceeding a speed limit. The result for different $r^a_{ss_3}$ values is shown in Fig. 3.5, where a) shows the case $r^a_{ss_3} = 100$ under consideration. The lower the value now assigned to this reward, the more unattractive

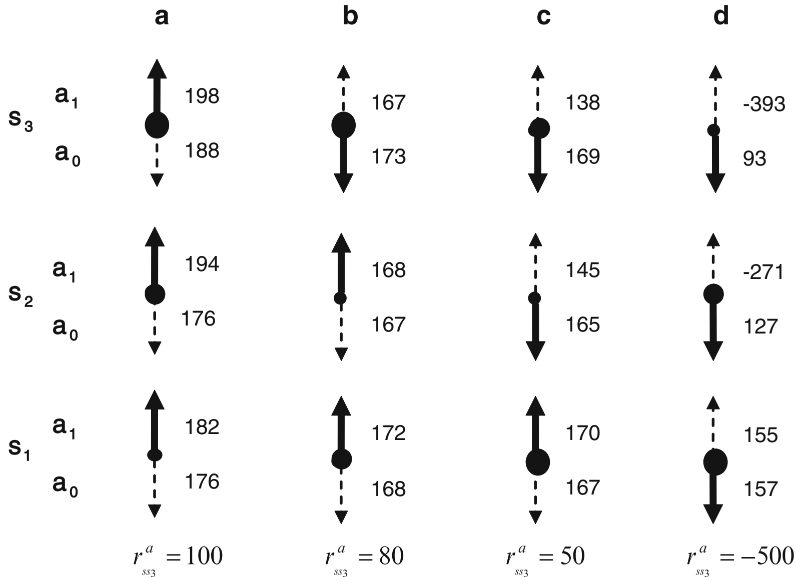


Fig. 3.5 Example of the car. Action values and optimal policies (*bold arrows*) for different $r^a_{ss_3}$ values of the third state. The larger the dot for a state, the higher its state value

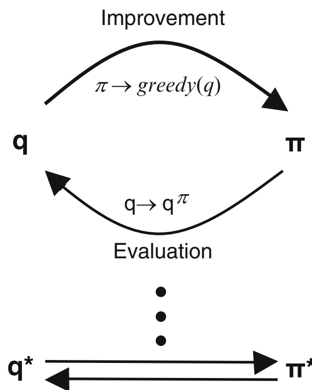


Fig. 3.6 GPI. Policy and action-value functions interact until they are mutually consistent and thus optimal

the transition into the last state. As $r^a_{ss_3}$ reduces, the number of states with non-acceleration actions increases. In the last case d), $r^a_{ss_3} = -500$ finally becomes so small that even in the first state, the optimal action is not to accelerate, although the reward of the second state is higher than that of the first one. However, the transition into the unattractive last state is so dangerous that, although we did not accelerate in the second state, even the tiny probability of a transition into the last state is still too high! Thus, our example of the car is a very good illustration of chain optimization. ■

For reinforcement learning, which encompasses more than just dynamic programming, the idea of policy iteration has been generalized to *general policy iteration (GPI)*, which is illustrated in Fig. 3.6.

The GPI approach is thus the following: the policy is constantly being improved with respect to the action-value function (policy improvement), and the action-value function is constantly being driven toward the action-value function of the policy (policy evaluation). If we compare Fig. 3.6 with the general case of the adaptive analytics method shown in Fig. 1.1, the policy evaluation represents the analysis step, while the policy improvement corresponds to the action. Virtually all reinforcement learning methods can be described as GPI.

But that's not all. The classic cross-selling approach, that is, Approach I from Chap. 2, can also be interpreted as an instance of GPI: the user behavior is analyzed (data mining model is evaluated) and its most promising products recommended at every process step. This represents policy improvement. After this, the user behavior, as changed by these recommendations, is analyzed afresh (a data mining model is created), which corresponds to policy evaluation. Once again, the most promising recommendations are derived from this, and so on. Of course, this is not a mathematically rigorous reasoning, but it reveals the power of the GPI approach.

3.7 The Adaptive Case

Thus far, we have been solving the Bellman equation (3.4) merely by means of static methods. Specifically, we have been assuming that a model of the environment be available beforehand in the form of its transition probabilities $p_{ss'}^{a}$ and rewards $r_{ss'}^{a}$, which then enables us to compute an optimal policy by means of complex policy iteration once and for all. In this respect, we have been acting no differently from the classical data mining approach (although, of course, the transition to the control problem adds a new level of quality).

An interesting observation is that the policy iteration used here can be interpreted throughout as a realtime analytics method in accordance with Fig. 1.1. This raises the question of whether the problem (3.4) may also be solved adaptively. The answer is yes! There are two main possibilities here:

1. Calculating the transition probabilities $p_{ss'}^{a}$ and rewards $r_{ss'}^{a}$ incrementally, then seeking after an adaptive solution to (3.4)
2. Abandoning the model of the environment completely, that is, solve the Bellman equation (3.4) indirectly, without explicit representations of $p_{ss'}^{a}$ and $r_{ss'}^{a}$

The starting point is the fundamental update equation

$$X_{k+1} = X_k + \alpha_{k+1}(x_{k+1} - X_k), \tag{3.8}$$

where X_k is the estimation of the target variable x_k in the kth update step. Here $(x_{k+1} - X_k)$ is the current estimation error. It is reduced by taking a step toward

the target variable. The coefficient α_k is the step size and defines the speed of the update.

Example 3.7 For $\alpha_k = \frac{1}{k}$, (3.8) is the step-size-like form of the average calculation, that is,

$$\frac{x_1 + x_2 + \ldots x_n}{n} = X_n = X_{n-1} + \frac{1}{n}(x_n - X_{n-1}),$$

where X_{n-1} is recursively calculated in the same way:

$$X_{k+1} = X_k + \frac{1}{k+1}(x_{k+1} - X_k), \quad k = 1, \ldots, \ n-1, \quad X_1 = x_1.$$

From $\alpha_k = \frac{1}{k}$, we can immediately see that in the average calculation, the contribution x_k of the kth update step decreases. The average thus captures changes in the behavior of the target variable relatively poorly – it is in a certain sense "lazy" (which however is necessary in order for it to converge). But it is not so lazy that it cannot asymptotically capture any fluctuations in the target variable, that is, even in the kth step for a large k, α_k is never "too small." Mathematically, this is expressed in the well-known property $\sum_{k=1}^{\infty} \frac{1}{k} = \infty$. ■

Both properties can be generalized, and the conditions for the convergence of (3.8) are

$$\sum_{k=1}^{\infty} \alpha_k = \infty, \quad \sum_{n=1}^{\infty} \alpha_k^2 < \infty. \tag{3.9}$$

The first condition ensures that the step remains large enough to capture movements. The second condition guarantees that ultimately, the steps become small enough to ensure convergence. For the average, because $\sum_{k=1}^{\infty} \left(\frac{1}{k}\right)^2 < \infty$, the second property of the convergence conditions (3.9) is also satisfied.

As well as the average, we will now consider a further important case, namely, the constant step size $\alpha_k = \alpha$. Here – conversely to the average – the contribution of x_k in the kth update step is always the highest, and the contributions of the preceding values $x_l, \ \ l < k$ decrease (even exponentially) as l reduces. In this way, the constant step size adapts particularly quickly to changes in the target variable, that is, it is very "agile" (but also therefore less stable). In fact, it even violates the second convergence condition, since $\sum_{n=1}^{\infty} \alpha^2 = \alpha^2 \sum_{n=1}^{\infty} 1 = \infty$. Constant step sizes are used primarily for nonstationary problems, where violation of the second convergence condition is not critical and may even be desirable. A practical

advantage of constant step sizes is that we need not save the step-size value k, so we only need to update our estimate X_k.

In practice, the average or the constant step size is often used, and the importance of the correct step-size parameter α_k is totally underestimated. Without going into further details here, we stress once more the theoretical and practical importance of this aspect for achieving a quick convergence of the process.

Update equations of the form (3.8) thus enable us to calculate the transition probabilities $p_{ss'}^{a}$ and rewards $r_{ss'}^{a}$ incrementally. But how can we perform the policy iteration adaptively? It would be extremely computationally intensive to determine it from scratch in every update step. The inherently adaptive approach in fact permits a derived adaptive variant, *asynchronous dynamic programming* (ADP). Here the sequence of policies and action-value functions is executed almost as the realtime interaction proceeds, coupled with additional internal updates in order to ensure convergence.

This is also interesting in that it enables the explorative mode to be designed in a much more sophisticated way than just using the simple ε-greedy and softmax policies. In explorative mode, therefore, those actions that can most quickly reduce the statistical uncertainty in our system are selected, thus ensuring the most rapid convergence. Selecting the correct actions is generally one of the most interesting subject areas in RL.

3.8 The Model-Free Approach

The determination of the transition probabilities $p_{ss'}^{a}$ and rewards $r_{ss'}^{a}$ is often computationally intensive and memory intensive, especially for large state and action spaces S and A. This raises the question of whether the Bellman equation (3.4) cannot also be solved indirectly even without a model of the environment. In fact, this can be done, and the corresponding method is referred to as *model-free*. In the following, we shall briefly present the most important model-free algorithm, *temporal-difference learning* (TD) developed by Sutton.

The model-free approach is based on learning by iterative adaptation of the action-value function *q(s, a).* We begin with the simple TD(0) method. At every step t of the episode, the update is performed as follows:

$$q(s_t, a_t) := q(s_t, a_t) + \alpha_t(r_{t+1} + \gamma q(s_{t+1}, a_{t+1}) - q(s_t, a_t)). \tag{3.10}$$

Obviously, the equation is of the same form as the update (3.8), where $\tilde{q}(s_t, a_t) := r_{t+1} + \gamma q(s_{t+1}, a_{t+1})$ is the target variable to be estimated. Please note that here, t acts as an index for the state-action pair (s_t, a_t) rather than the update step k for the value $q(s_t, a_t)$. Strictly speaking, we must write

$$q^{k+1}(s_t, a_t) = q^k(s_t, a_t) + (\alpha_t)_{k+1}\left(r_{t+1} + \gamma q^k(s_{t+1}, a_{t+1}) - q^k(s_t, a_t)\right).$$

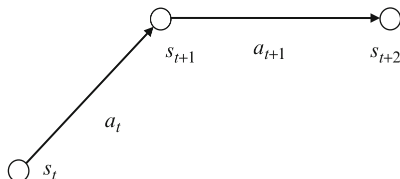


Fig. 3.7 A sequence of an episode of 2 steps

So while t indicates the step within the episode, that is, along the chain

$$(s_1, a_1) \rightarrow (s_2, a_2) \rightarrow (s_3, a_3) \rightarrow \ldots$$

k is the index of the update for a fixed pair (s_t, a_t) throughout all episodes. In order not to overburden the notation, we leave out the index k and in its place use the assignment symbol ":=."

Before we come to the explanation, the first question immediately arises: since, to carry out an update of the action value $q(s_t, a_t)$ at step t in realtime, we need the action value $q(s_{t+1}, a_{t+1})$ of the next step $t + 1$, how is this supposed to work in practice? Doesn't this remind you of Baron Münchhausen, who escapes from the swamp by pulling himself up by the hair?

There are simple solutions to this: we can, for instance, wait until step $t + 1$ and then perform the update (3.10), that is, always learn with a delay of one step. Or we may exploit the fact that we determine the actions ourselves via the policy (provided we are not learning from historical data): at step t, we already know our next action a_{t+1} and can thus work with the current $q(s_{t+1}, a_{t+1})$ (Fig. 3.7).

To continue with the explanation, α_t is the learning parameter at step t. The higher it is, the faster the algorithm learns. Thus, the current *temporal-difference* d_t is

$$d_t(s_t, a_t, s_{t+1}, a_{t+1}) = r_{t+1} + \gamma q(s_{t+1}, a_{t+1}) - q(s_t, a_t) \tag{3.11}$$

and (3.10) takes the following form:

$$q(s_t, a_t) := q(s_t, a_t) + \alpha_t d_t(s_t, a_t, s_{t+1}, a_{t+1}). \tag{3.12}$$

This means that we compute the new estimate $\widetilde{q}(s_t, a_t) := r_{t+1} + \gamma q(s_{t+1}, a_{t+1})$ and subtract the previous iterate $q(s_t, a_t)$ therefrom. If $\widetilde{q}(s_t, a_t)$ is greater than $q(s_t, a_t)$, then the latter is increased in accordance with (3.11); if $\widetilde{q}(s_t, a_t)$ is less than $q(s_t, a_t)$, then the latter is decreased in accordance with (3.11).

So what does $\widetilde{q}(s_t, a_t) := r_{t+1} + \gamma q(s_{t+1}, a_{t+1})$ mean? We know that $q(s_t, a_t)$ is the expected return taken across the remainder of the episode. The first term r_{t+1} is the direct reward of the recommendation a_t. The second term $\gamma q(s_{t+1}, a_{t+1})$ is the expected return from the new state s_{t+1}. It follows that there are once again two possibilities for the reason why $\widetilde{q}(s_t, a_t)$ may be higher than $q(s_t, a_t)$: either the direct reward r_{t+1} is high or the action a_t has led to a valuable state s_{t+1} with a high action value $q(s_{t+1}, a_{t+1})$ (or both).

At every step t, the TD(0) algorithm that was described modifies the action-value function only in the current state s_t. Since, however, the action values of the following states are also included in its calculation of the discount, conversely, the action values of all preceding states can also be updated at each step, which significantly increases the speed of learning. This is achieved using algorithms of the TD(λ) family. By this means, at every step, all action values are updated as follows:

$$q(s,a) := q(s,a) + \alpha_t d_t(s_t, a_t, s_{t+1}, a_{t+1}) z_t(s,a), \tag{3.13}$$

where the weighting function $z_t(s, a)$ describes the relevance of the temporary difference at the time point t for the state-action pair (s, a). The weighting function $z_t(s, a)$ is called *eligibility traces* and assigns recently visited states with a higher weighting than states which have not been visited for some time. The usual definition of eligibility traces is

$$z_t(s,a) = \begin{cases} \gamma\lambda z_{t-1}(s,a) + 1, & \text{if } (s,a) = (s_t, a_t) \\ \gamma\lambda z_{t-1}(s,a), & \text{if } (s,a) \neq (s_t, a_t) \end{cases}, \tag{3.14}$$

where $0 \leq \lambda \leq 1$ is the eligibility trace parameter. For this, $z_t(s, a)$ is initialized as zero for all states. Obviously, $z_t(s, a)$ is relatively high if (s, a) is visited often and (s_t, a_t) can be reached from (s, a) in only a few steps. The weighting $z_t(s, a)$ decreases exponentially with the number of steps since the last visit of the pair (s, a).

For computational purposes, $z_t(s, a)$ can be set to zero if it falls below an epsilon barrier specified beforehand, so that it is given a local support and thus can be implemented asymptotically optimal with respect to computation time and memory. In practice, this means that after each update, (3.12) has been performed for the current time step t, that is, for (s_t, a_t), the update (3.13) is performed for all preceding time steps in the current episode t-1, t-2, …, t-m, where t-m is the last preceding time step where $z_t(s, a)$ lies above the epsilon barrier. At the same time, $z_t(s, a)$ must be updated in accordance with (3.14) for all affected time steps.

In this way, the TD(λ) algorithm described above performs a continual adaptation of the action-value function $q(s, a)$ in an intuitive fashion. It can be shown that the TD(λ) algorithm converges in a certain probabilistic sense, which is technically referred to as *almost sure convergence*, to the optimal policy π^* and action-value function q^*. Of course, it can also be interpreted as a general policy iteration. At every step t, a policy evaluation is always performed in accordance with (3.13), and after that – based on the updated action-value function – a further policy improvement is performed, and so on.

Notice that there exist different versions of the TD(λ) algorithm; the one we have described here is called *Sarsa*(λ) because it works with the quintuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$. Further important versions are concerned with *Q learning*, for example, the *Watkins* Q(λ). We do not want to explain this here, and for details, refer to [SB98].

We have shown that RL can also learn in an online fashion, thus solving Problem 3 in Chap. 2.

3.9 Remarks on the Model

In what follows, we shall discuss some important details concerning the model that we have hitherto abstained from addressing for the sake of a concise introduction. Apart from being necessary for the exactness of mathematical model, however, they mostly bring about practical consequences as well.

3.9.1 Infinite-Horizon Problems

As rendered in Sect. 3.1, we distinguish between episodic task, that is, those that terminate, and continuing tasks, that is, those that do not terminate. To facilitate their treatment in form and content, they will be considered within a unified framework.

As regards continuing tasks, we have no further remarks apart from the requirement that $\gamma < 1$. As has already been established in Sect. 3.2, this requirement is necessary to ensure existence and uniqueness of the expected results.

With regard to episodic tasks, like those that primarily concern our recommendation engines, the following question arises: how, after all, do we describe the end of an episode? Since, according to (3.2), the transition probabilities for each state-action pair sum up to one, an episode actually never terminates. To circumvent this, we introduce a so-called *terminal* (or, *absorbing*) state, which allows for transitions to no state other than itself, and the corresponding reward is set to zero (Fig. 3.8).

Hence, after a certain time step at which the terminal state has been reached, all further rewards are (in the example depicted by Fig. 3.8, this time step is 3)

$$r_t = 0, \quad t > t_a.$$

Thus, the sum (3.1) is also well defined for episodic tasks, and we may therefore consider both continuous and episodic tasks as infinite sums. Both types of tasks are said to be *infinite-horizon problems* [BT96]. Eventually, it should be mentioned that episodic tasks with $\gamma = 1$ are referred to as *stochastic shortest path problems*, the special properties of which have been studied comprehensively in control theory.

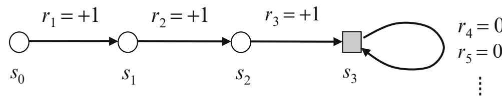


Fig. 3.8 Example of an episode with terminal state (*gray box*)

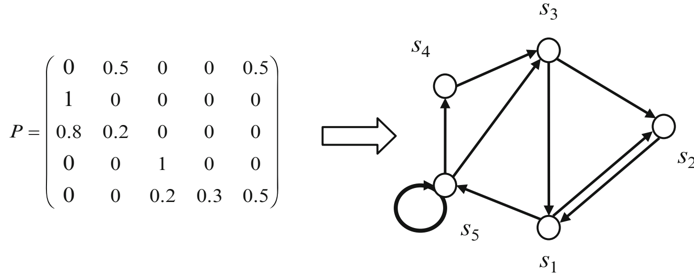


Fig. 3.9 Example of a matrix P (with five states) and the thereby induced graph $\Gamma(P)$. The latter contains, e.g., the cycles (s_1, s_2, s_1), $(s_1, s_5, s_4, s_3, s_1)$, as well as (s_5, s_5)

3.9.2 Properties of Graphs and Matrices

In the following, we shall, on one hand, consider the matrix of transition probabilities $P = (p_{ss'})_{s,s' \in S}$, on the other hand, the thereby induced graph $\Gamma(P)$. (Here, we shall ignore the actions a; $P = P^{\pi}$ will denote the transition probability matrix of the Markov chain M_{π}.) For a detailed version of this discussion, we refer the reader to the fundamental work by Paprotny [Pap10]. Furthermore, the general link between matrix analysis and numerical linear algebra on one hand and dynamic programming and stochastic iterative methods on the other hand has been studied therein.

We first define the graph $\Gamma(P)$ of the matrix P. We recall that a directed graph is described by its sets of nodes and edges. In our case, the set of nodes is precisely the state space, and the set of edges is the set of possible state transitions. Thus, formally, the directed graph of P is defined as

$$\Gamma(P) := \left(\underline{n}, E\right), \quad (i,j) \in E \Leftrightarrow p_{ij} \neq 0.$$

A tuple $(s_1, \ldots, s_l)$ of nodes s_i is said to be a *path* of length l-1 from s_1 to s_l, if

$$(s_i, s_{i+1}) \in E \quad \forall i \in \underline{l-1}.$$

A path is called a *cycle*, if $s_1 = s_l$. A node is *reachable* from $v \in S$, if there is a path from v to w in Γ. These notions are illustrated by Fig. 3.9.

Intuitively, a matrix is said to be *reducible* if it can be transformed into an upper block triangular matrix by some permutation. A matrix is called *irreducible* if it is not reducible. The graph Γ is said to be strongly connected, if for each pair of nodes v, $w \in S$, there is a path from v to w in Γ. The following holds: P is irreducible if and only if $\Gamma(P)$ is strongly connected. So, P in the example from Fig. 3.9 is irreducible.

Irreducibility of P is a prerequisite for several important properties in RL and, in particular, ensures convergence of some important procedures, as, for example, the

TD(λ) algorithm. In case that the system be reducible, it may be decomposed into smaller irreducible subsystems that may then be considered separately.

Since P is a *row stochastic* matrix, that is, a matrix of transition probabilities, its rows sum up to 1; see (3.2). In other words, the vector of all ones is an eigenvector of P corresponding to the eigenvalue 1. Since P is nonnegative, its largest row sum coincides with its row sum norm. Therefore, 1 is also the spectral radius of P, that is, the largest absolute value of its eigenvalues.

A fundamental result of the theory of nonnegative matrices states that any irreducible nonnegative matrix has a positive spectral radius which is itself an eigenvalue. Furthermore, this eigenvalue is algebraically simple. This eigenvalue and the corresponding left (right) eigenvector are referred to as *Perron eigenvalue* and left (right) *Perron vector*, respectively.

The matrix P is said to be *primitive* if it is irreducible and the absolute value of all of its eigenvalues except the Perron eigenvalue is strictly smaller than the spectral radius. This abstract definition may be captured by the following simple criterion: the matrix P is primitive if and only if P^k is (strictly) positive for some positive integer k. The following sufficient condition holds: the matrix P is primitive if it is irreducible and $p_{ii} > 0$ for some $i \in \underline{n}$. Hence, for example, our matrix from Fig. 3.9 is primitive, since $p_{55} = 0,5 > 0$. In terms of the graph $\Gamma(P)$, this corresponds to the criterion of the existence of a cycle of length 1, that is, a node is connected to itself.

Thus, we essentially conclude our brief introduction to fundamental algebraic properties of the transition probability matrix P. At the same time, we saw that each of these has an intuitive graph theoretical counterpart with respect to the graph induced by P. We will make use of these properties below.

3.9.3 The Steady-State Distribution

As noted above, the property (3.2) is called row stochasticity. If P is primitive, the Perron-Frobenius theorem implies that

$$\lim_{k\to\infty} \frac{P^k}{\rho(P)} = \frac{xy^T}{y^T x}, \quad x, y \in \Re^n,$$

where

$$Px = \sigma(P)x, \quad x > 0, \ (1 \ldots 1)\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = 1; \ \ y^T P = \sigma(P)y^T,$$

$$y > 0, \ (1 \ldots 1)\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = 1.$$

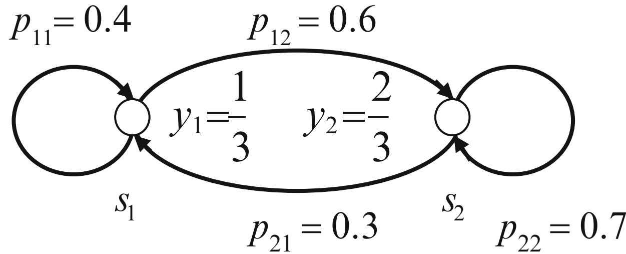


Fig. 3.10 A graph $\Gamma(P)$ (with two states) and its steady-state probabilities

Here, $\sigma(P)$ denotes the spectral radius of P, that is, the largest absolute value of its eigenvalues. Moreover, x and y are referred to as *right* and *left Perron vector*, respectively. As stated above, the spectral radius satisfies $\sigma(P) = 1$ since P is stochastic, and we may write the right Perron vector as

$$x = \frac{1}{n}\begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}.$$

Let us now consider the left Perron vector. Since, by definition, it is positive and satisfies $\mathbf{1}^{\mathbf{T}}y = 1$, we may consider it as a probability distribution on S. In virtue of the Perron-Frobenius theorem, we obtain

$$P^k \overset{k\to\infty}{\longrightarrow} \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} (y_1 \ldots y_n)$$

for primitive P. This distribution is referred to as the *steady-state distribution* (or *stationary distribution*), to which the user behavior converges. This property is a prerequisite for the convergence of the TD(λ) algorithm as well as other procedures.

Example 3.8 To illustrate the abstract discussion, we consider an outright simple example, which is depicted by Fig. 3.10.

We thus have two states and the following transition matrix P:

$$P = \begin{pmatrix} 0.4 & 0.6 \\ 0.3 & 0.7 \end{pmatrix}.$$

Then the left Perron vector is given by

$$(y_1 \;\; y_2)\begin{pmatrix} 0.4 & 0.6 \\ 0.3 & 0.7 \end{pmatrix} = (y_1 \;\; y_2), \quad (1 \quad 1)\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = y_1 + y_2 = 1,$$

and we obtain

$$y = \begin{pmatrix} \frac{1}{3} \\ \frac{2}{3} \end{pmatrix}.$$

In words, if a user in state 1 dwells on it with 40 % probability and switches to state 2 with 60 % probability and, conversely, remains in state 2 with 70 % probability and moves to state 1 with 30 % probability, he/she will – in a session of infinite length – be found in state 1 in 33 % and in state 2 in 66 % of the trajectory. ■

3.9.4 On the Convergence and Implementation of RL Methods

In this section, we will investigate the convergence of the two most important methods that we have introduced so far: the policy iteration from Sect. 3.6 and the temporal-difference learning from Sect. 3.8. So the question is: do these methods converge against the optimal policy and under which assumptions? The question of the convergence speed will not be addressed here.

The question of convergence is far more than of theoretical interest only. Because in the end for the solution of our practical problems, that is, computation of recommendations, we want to be sure that a solution even exists.

We start with the policy iteration. In Sect. 3.5, we mentioned that by selecting a policy π for the Markov decision process, we obtain a Markov chain. Indeed, by determining the decision in form of the policy, there remains nothing more to decide in the decision process but only to model the Markov chain.

The Bellman equation (3.7) can be written compactly in vector notation as

$$v^\pi = r^\pi + \gamma P^\pi v^\pi, \tag{3.15}$$

where the vectors of the state values v^π and rewards r^π are defined as follows:

$$\begin{aligned} v^\pi &= \left[v^\pi(s_1)\ \ v^\pi(s_2)\dots v^\pi(s_N)\right]^T, \\ r^\pi &= \left[r^\pi(s_1)\ \ r^\pi(s_2)\dots r^\pi(s_N)\right]^T, \\ r^\pi(s) &= \sum_a \pi(s,a) \sum_{s'} p^a_{ss'} r^a_{ss'} \end{aligned} \tag{3.16}$$

and P^π is the matrix of transition probabilities $p^\pi_{ss'}$ (3.3).

Similarly, the Bellman equation for the action-value function (3.6) can be reformulated as

$$w^{\pi} = \hat{r}^{\pi} + \gamma \hat{P}^{\pi} w^{\pi}, \tag{3.17}$$

where the vectors of the action values w^{π} and rewards $\hat{r}^{\pi}$ together with the matrix of transition probabilities $\hat{P}^{\pi}$ are defined as follows:

$$\begin{aligned} w^{\pi} &= \left[q^{\pi}(s_1, a_1)\ \ q^{\pi}(s_1, a_2) \ldots q^{\pi}(s_N, a_M)\right]^T, \\ \hat{r}^{\pi} &= \left[\hat{r}^{\pi}(s_1, a_1)\ \ \hat{r}^{\pi}(s_1, a_2) \ldots \hat{r}^{\pi}(s_N, a_M)\right]^T, \\ \hat{r}^{\pi}(s, a) &= \sum_{s'} p_{ss'}^{a} r_{ss'}^{a}, \\ \hat{p}^{\pi}_{s,a,s',a'}(s) &= p_{ss'}^{a} \pi(s', a'). \end{aligned} \tag{3.18}$$

We now exemplarily consider the state-value function and define the *Bellman operator* T^{π} for a policy π as

$$T^{\pi}(v) = r^{\pi} + \gamma P^{\pi} v$$

and consider the iteration

$$v^{k+1} = T^{\pi}\left(v^{k}\right). \tag{3.19}$$

The following proposition can be easily shown.

Proposition 3.1 For all MDPs, each policy has a unique and finite state-value function, which is the unique fix point of the iteration defined in Equation (3.19). The iteration converges to its unique fix point at asymptotic rate γ for any initial guess. The asymptotic rate is attained in l_{∞}.

Thus, Proposition 3.1 ensures that in each policy evaluation step of the policy iteration, we will find a unique solution of the Bellman equation.

We now turn our attention to the existence and uniqueness of an optimal policy π^*, that is, a policy fulfilling

$$v^{\pi^*} \leq v^{\pi}, \quad \pi \in \Pi_M,$$

where Π_M is the space of all policies of the Markov decision process M. The following theorem provides the desired result.

Theorem 3.1 *The policy iteration terminates after a finite number of iterations with the tuple* (π^*, v^{π^*}).

Thus, it turns out that the convergence of the policy iteration does not require further specific assumptions.

At the same time, iteration (3.19) answers the open question of Sect. 3.4 for a solution method of the Bellman equation for a given policy π. In case we are working with the action-value function instead of the state-value function, as, for example, in Sect. 3.6, everything described here carries over to the action values w^{π}, and we obtain instead of (3.19) the fix point equation

$$w^{k+1} = \hat{T}^{\pi}\left(w^{k}\right), \hat{T}^{\pi}(w) = \hat{r}^{\pi} + \gamma \hat{P}^{\pi} w.$$

Therefore, we can now enlist the complete algorithm of the policy iteration (Algorithm 3.1). To make the description easier, we again use the state-value function, having in mind that it looks similar for the action-value function.

Algorithm 3.1: Policy iteration

Input: transition probabilities P and -rewards R, discount rate γ, small inner bound $\theta \in \Re,\ \ \theta > 0$
Output: optimal state-value function v*

1. **procedure** POLICY_ITERATION(v, P, R, γ, θ)
2. **repeat** ▹ block of policy evaluation
3. $\Delta := 0$
4. **for** *each* $s \in S$ **do**
5. $v := v(s)$
6. $v(s) := \sum_{s'} p_{ss'}^{\pi(s)} [r_{ss'}^{\pi(s)} + \gamma v(s')]$
7. $\Delta := \max(\Delta, |v - v(s)|)$
8. **end for**
9. **until** $\Delta < \theta$
10. *policy-stable:=true* ▹ block of policy improvement
11. **for** *each* $s \in S$ **do**
12. $b := \pi(s)$
13. $\pi(s) := \arg\max_a \sum_{s'} r_{ss'}^{a} [p_{ss'}^{a} + \gamma v(s')]$
14. **if** $b \neq \pi(s)$ **then**
15. *policy-stable:=false*
16. **end if**
17. **end for**
18. **if** *policy-stable* **then**
19. **stop**
20. **else**
21. **goto** 2
22. **end if**
23. **return** v
24. **end procedure**
25. initialize $v(s)$, $\pi(s)$ $\forall\ s \in S$ arbitrarily
26. v:= POLICY_ITERATION(v, P, R, γ, θ)
27. **return** v

Of course, there are numerous variants of policy iteration for the solution of the Bellman equation. First, instead of directly applying (3.19) in the policy evaluation for the solution of (3.15), which may be considered as a simple Richardson iteration, we may employ more sophisticated iteration procedures. We shall address this in Chap. 6, in particular with respect to its connection with hierarchical methods for convergence acceleration.

Another important approach consists in linking policy evaluation and improvement more strongly: instead of carrying out the entire policy evaluation to compute the state-value function of a fixed policy exactly before executing the policy improvement step, we carry out the latter in after each step of the iteration in the policy evaluation. The resulting approach is thus more feasible (which does not imply that it be faster or better) and is referred to as *value iteration*.

The value iteration is represented by Algorithm 3.2.

Algorithm 3.2: Value iteration

Input: transition probabilities P and -rewards R, discount rate γ, small inner bound $\theta \in \Re, \ \theta > 0$

Output: optimal state-value function v^*

1: **procedure** VALUE_ITERATION(v, P, R, γ, θ)

2: **repeat**

3: $\Delta := 0$

4: **for** *each* $s \in S$ **do**

5: $v := v(s)$

6: $v(s) := \max_a \sum_{s'} p_{ss'}^a [r_{ss'}^a + \gamma v(s')]$

7: $\Delta := \max(\Delta, |v - v(s)|)$

8: **end for**

9: **until** $\Delta < \theta$

10: **return** V

11: **end procedure**

12: initialize $v(s), \ \pi(s) \ \ \forall \ s \in S$arbitrarily

13: v:= VALUE_ITERATION(v, P, R, γ, θ)

14: **return** v

Concerning the solution of the Bellman equation (3.15), the policy iteration may be algebraically interpreted as *additive* and the value iteration as *multiplicative* approach.

Finally, we will formally write down the Asynchronous DP algorithm. Since it exists in different version, we consider the one that we use in our tests in Chap. 5. This version is a fully adaptive one where also the transition probabilities and rewards are determined incrementally. Since it is an in-place method, it formally rather represents a value iteration.

Algorithm 3.3: Adaptive ADP

Input: online rewards r and -transitions s', discount rate γ, small inner bound $\theta \in \Re, \ \theta > 0$

Output: optimal state-value function v^*

1: initialize $v(s), \ \pi(s), \ p_{ss'}^a, \ r_{ss'}^a \ \ \forall \ s, s' \in S, a \in A(s)$arbitrarily

2: **repeat** *for each episode*

(continued)

Algorithm 3.3: (continued)

3: initialize s, a
4: **repeat** *each step of episode*
5: take action a, observe r, s'
6: update $p^a_{ss'}$, $r^a_{ss'}$, e.g. using (3.8)
7: $v(s) := \max_a \sum_{s'} p^a_{ss'} [r^a_{ss'} + \gamma v(s')]$
8: **until** *s is terminal*
9: v:= VALUE_ITERATION(v,P,R,γ,θ) ▷update v over all states
10: **until** *stop*

Of course, in step 9, also the policy iteration of Algorithm 3.1 may be applied. Additionally, step 9 is not required to be performed after each episode – the call may also occur less often. It is also not necessary to compute v exactly, for example, the iteration process may be stopped before the termination criterion is fulfilled. One iteration sweep through all states is enough. We only need to ensure that no states are left out permanently in the ADP.

Now we switch to the convergence of the TD(λ) method which is a more complex topic. First, we rewrite the TD(λ) method (3.13) in vector notation and include the iteration index k:

$$w^k := w^k + \alpha_t z_t d_t^k, \tag{3.20}$$

where w^k in accordance to (3.17) represents the vector of action values $q^k(s, a)$ and

$$z_t = \left[z_t(s_1, a_1) \;\; z_t(s_1, a_2) \;\; \ldots z_t(s_N, a_M)\right]^T.$$

The following theorem ensures the convergence.

Theorem 3.2 *Let the following assumptions hold:*

1. *π is a policy for an MDP M such that M_π is irreducible and aperiodic.*
2. *$(i_t)_{t \in \mathbf{N}_0}$ is a trajectory generated by M_π.*
3. $\lambda \in [0, 1]$
4. *The sequence of step sizes $(\alpha_t)_{t \in \mathbf{N}_0}$ satisfies (3.9).*

Then, for any initial guess w^0, the sequence $\left(w^k\right)_{k \in \mathbf{N}_0}$ generated by the iteration (3.20) converges almost surely to w^π.

Now following the reasoning of Sect. 3.6, by selecting a proper policy in each iteration step, we ensure the convergence of the TD(λ) method to the optimal policy π^*. For details, we refer to [BT96].

Finally, we enlist the TD(λ) algorithm, namely, the Sarsa(λ) version, in Algorithm 3.4.

Algorithm 3.4: Sarsa(λ)

Input: online rewards r and -transitions s', step-size α, discount rate γ, eligibility trace parameter λ
Output: optimal action-value function q^*
1: initialize arbitrarily $q(s, a)$, $z(s, a) = 0 \;\; \forall\, s \in S, \forall\, a \in A(s)$
2: **repeat** *for each episode*
3: initialize s, a
4: **repeat** *for each step of episode*
5: take action a, observe r, s'
6: choose a' from s' using policy derived from q (e.g. ε-greedy)
7: $d := r + \gamma q(s', a') - q(s, a)$
8: $z(s, a) := z(s, a) + 1$
9: **for** *all s, a* **do**
10: $q(s, a) := q(s, a) + \alpha d z(s, a)$
11: $z(s, a) := \gamma \lambda z(s, a)$
12: **end for**
13: **until** *s is terminal*
14: **until** *stop*

3.10 Summary

In this chapter, we gave a short introduction to reinforcement learning. We have seen that RL addresses the problems from Chap. 2. This shows that in principle, RL is a suitable tool for solving all of the four problems described in Chap. 2.

Furthermore, in addition to online learning, we had previously suggested offline learning by policy iteration for solving the Bellman equation. Both approaches are linked consistently via the action-value and state-value functions. We can, for instance, calculate the action-value function offline, using historical data, and then update it online. In this way, RL is also a very nice example of the link between the two types of learning, in accordance with Rule III in Chap. 1.

We now return to reinforcement learning and consider its application to recommendation engines.

Chapter 4
Recommendations as a Game: Reinforcement Learning for Recommendation Engines

Abstract We describe the application of reinforcement learning to recommendation engines. At this, we introduce RE-specific empirical assumptions to reduce the complexity of RL in order to make it applicable to real-live recommendation problems. Especially, we provide a new approach for estimating transition probabilities of multiple recommendations based on that of single recommendations. The estimation of transition probabilities for single recommendations is left as an open problem that is covered in Chap. 5. Finally, we introduce a simple framework for testing online recommendations.

An effective approach to using reinforcement learning for recommendation engines is described below. In the simplest case, the product detail views form the states, the recommended products the actions, and the rewards the clicks or purchases of the products. The goal consists (depending on the chosen reward) in maximizing the activity (clicks) or the success (sales).

Figure 4.1 illustrates the use of RL for product recommendations in a web shop and shows the interaction between the recommendation engine and the user. Here, the optimal proven recommendations are marked with "*."

In the first and third steps, the recommendation engine is following the proven recommendations (exploitation); in the second step, a new recommendation is tried out (exploration). The user ignores the first recommendation, but accepts the second and third. The feedback arrows symbolize the updating of the recommendations.

Although this modeling may appear self-evident, it nevertheless represents a highly complex task. Firstly, web shops generally offer very many products, as a rule between a few thousand up to a few million (for instance, at a bookshop). Many of these products have virtually no transaction history, i.e., they have scarcely ever been bought; indeed, some have never even been clicked on. Furthermore,

A. Paprotny and M. Thess, *Realtime Data Mining: Self-Learning Techniques for Recommendation Engines*, Applied and Numerical Harmonic Analysis,
DOI 10.1007/978-3-319-01321-3_4, © Springer International Publishing Switzerland 2013

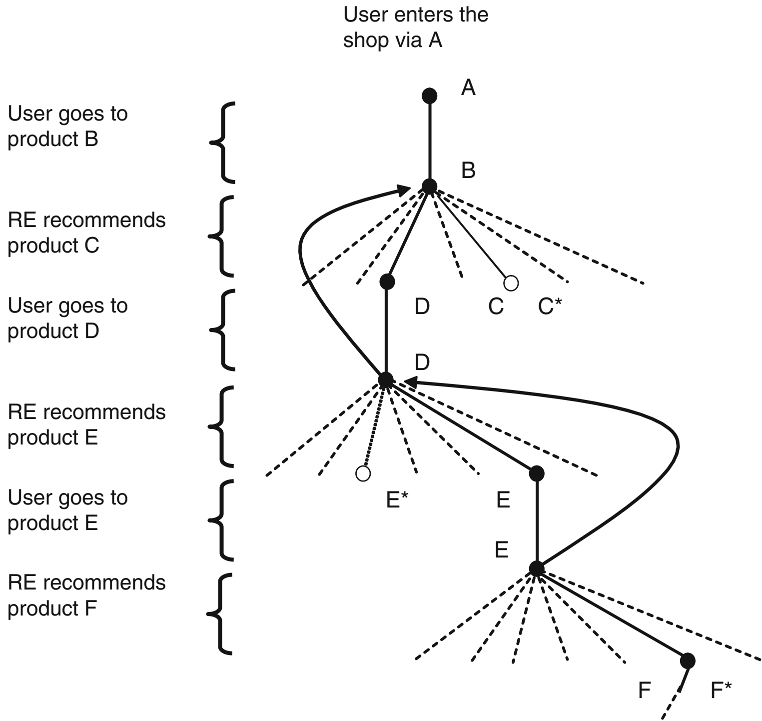


Fig. 4.1 Example of reinforcement learning for product recommendations in a web shop

the existing transactions are mostly clicks, whereas, on the other hand, placements in the shopping basket (SB) and purchases are far more infrequent. However, maximizing sales is the primary goal of REs. Let us summarize these two problems again:

1. High numbers of products, a majority of which have minimal transaction history.
2. The vast majority of transactions are clicks; only a fraction are placements in the shopping basket and purchases.

We know, however, from the theory and practice of RL that high transaction numbers are necessary in order to achieve convergence. The above problems therefore appear to be killer arguments against the direct use of RL for REs. There already exist first approaches for using reinforcement learning for recommendation engines [GR04, RSP05, SHB05, TGK07, Mah10]. However, most of them are not able to overcome the complexity problems.

Therefore, additional empirical assumptions are made and justified below, which reduce the complexity of using RL for REs.

4.1 Basic Approach

As described initially, each product view represents a state s and a recommendation of another product represents an action a. Each web session (session for short) forms an episode.

The result is that the interaction between user and recommendation engine in each web session can be considered as a sequence of product transitions under the influence of recommendations (Fig. 4.2):

This permits us to model the most important statistical characteristics such as action values, transition probabilities, and rewards using rules $s \rightarrow s'$, which can be saved, for instance, in files or database tables (we will explain the details of this later). Of course, not every action will necessarily lead to an accepted recommendation: the user can also ignore the recommendations and go to an entirely unrelated product. In this case, however, the product transition is added as a new rule to the rule base and thus provides a new potential action.

Since all actions a represent product views, the sets of states $\boldsymbol{S}$ and actions $\boldsymbol{A}$ are isomorphic:

$$\boldsymbol{S} \cong \boldsymbol{A}. \tag{4.1}$$

It should be noted that for reasons of complexity, not all actions $\boldsymbol{A}$ are considered for each state s, but only a subset $A(s)$, which initially contains all product transitions that have actually occurred, together with actions derived by other means such as hierarchies (Chap. 6). By this means, the action set $A(s)$ expands dynamically in the course of the learning process.

In accordance with (4.1), we introduce the notation s_a for the product associated with the recommendation a (i.e., the recommended product "a"). Conversely, a_s represents the recommendation associated with the product s. The successor states corresponding to the recommendations of the action set $A(s)$ are denoted by $S_{A(s)}$, and thus the isomorphism (4.1) also applies on the recommendations of any state s: $S_{A(s)} \cong A(s)$. The number of recommendations in s, i.e., the cardinality of the action set, is usually denoted by m.

At each step, the RE receives a reward r. The sum of all the rewards should be maximized over the complete session. The reward is defined for each step as follows: if a product s is placed in the shopping basket or is bought, the preceding action (i.e., the "recommendation" a, which has led to the product) receives the

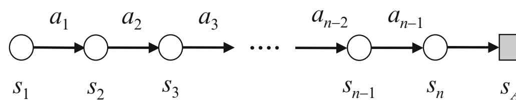


Fig. 4.2 Sequence of products and recommendations as states and actions including absorbing state s_A

value of the product s (price, revenue, etc.) as its reward; otherwise, it receives a small click reward, close to 0. This reflects the primary goal of seeking to maximize the shopping basket values or the sales/revenue. Note that orders constitute a delayed reward, since, in most cases, they appear only at the end of a session. The definition of the correct reward is linked to various refinements that will not be further explored here.

We now come to the statistical characteristics. Let us state our first fundamental assumption:

> **Assumption 4.1 (Markov property for REs):** In every state s, the optimal action a, i.e., the best recommendation, depends solely on the current state s, i.e., the product under consideration.

Of course, this Markov property for REs is satisfied only incompletely, since the best recommendation also depends on the preceding states of s together with their transactions. Nevertheless, for the evaluation of a recommendation by the user, the product currently viewed plays the main role, so the assumption may be considered reasonable. (There is also compelling empirical evidence on this point, namely, classic cross-selling, which is described using precisely this form of rules and whose effectiveness is beyond doubt.)

As a further simplification, let us assume that the reward in the state transition from s to s' is independent of the influence of the action a:

> **Assumption 4.2 (Reward property for REs):** For each state transition from s to s', the obtained reward $r_{ss'}^{a}$ is independent of the action a.

This means that

$$r_{ss'}^{a} = r_{ss'}. \tag{4.2}$$

In fact, it can be assumed that the user's decision as to whether or not to place a product in the shopping basket depends primarily on the product itself and not on the preceding recommendation. Thus, the estimated reward can technically be validly saved as a characteristic of the rule $s \to s'$.

The action-value function $q(s,a)$ assigns the expected return, i.e., the expected sales over the remainder of the session, to each product s and to each of its recommendations a. Technically, $q(s,a)$ can thus also be represented by the rule $s \to s_a$ from product s to the recommended product s_a.

There remains the question of the transition probabilities $p_{ss'}^{a}$. This is a complicated subject, which we shall consider in depth in Chap. 5.

4.2 Multiple Recommendations

So far, we have been assuming that there is only a single recommendation a. REs, however, generally recommend more than one product. We therefore now turn to the case of multiple (or composite) recommendations, i.e.,

$$\overline{a} = (a_1, \ldots, a_k)$$

is a recommendation composed of k single recommendations (Fig. 4.3).

From the point of the reinforcement learning theory as presented in Chap. 3, we can consider composite recommendations $\overline{a}$ as single actions in absolute the same way as we did it with our single recommendations a. The problems which we are facing are of rather computational nature since the number of admissible actions $\overline{a}$ in a state s may be huge and in general we are no more able to process all actions, e.g., for calculating the policy. However, we will solve this problem step by step.

First, we notice that due to Assumption 4.2, we do not need to care about multiple recommendations in the transition reward, i.e.,

$$r^{\overline{a}}_{ss'} = r_{ss'}.$$

Much more demanding is the problem of the transition probabilities. We will introduce two approaches on how transition probabilities for multiple recommendations $p^{\overline{a}}_{ss'}$ can be expressed through transition probabilities of single recommendations $p^{a}_{ss'}$ (the latter will be studied in the next chapter).

We firstly make the following assumption, which is usual for statistics:

Assumption 4.3 (Multiple recommendation probability property): For the multiple recommendation $\overline{a}$, the transition probabilities of the single recommendations $p^{a_i}_{ss'}$ can be considered as stochastically independent.

That means we assume that recommendations are not mutually cannibalistic. This is a reasonable assumption.

4.2.1 Linear Approach

Based on Assumption 4.3, the following approach is suitable: the transition probability for multiple recommendations $p^{\overline{a}}_{ss'}$ is equal to the average of the

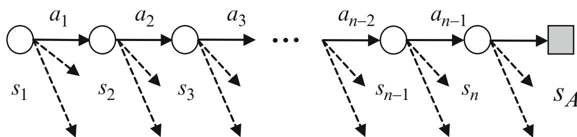


Fig. 4.3 Sequence of products and multiple recommendations as states and actions

transition probabilities of the single recommendations $p_{ss'}^{a}$. This gives the transition probability $p_{ss'}^{\overline{a}}$ as

$$p_{ss'}^{\overline{a}} = p_{ss'}^{(a_1,\ldots,a_k)} = \frac{1}{k}\sum_{i=1}^{k} p_{ss'}^{a_i}. \tag{4.3}$$

The advantage of this approach is that it is simple and linear with respect to the single probabilities $p_{ss'}^{a_i}$. However, for the case that we only work with probabilities of state transitions associated with the recommendations, i.e., $p_{ss_a}^{a}$, the presented approach is not applicable as we will show later. We therefore introduce a more sophisticated approach also based on Assumption 4.3.

4.2.2 Nonlinear Approach

For ease of reading, we will omit the product s from the indices and denote the recommended or transition product by its index. Thus, we write $p_{ss_j}^{a_i} =: p_j^i$, $\overline{a} =: (1, \ldots, k)$, and $p_{ss_j}^{(a_1,\ldots,a_k)} =: p_j^{(1,\ldots,k)}$.

In order to illustrate the problem, let us consider the case of two recommendations. For the case of the two single probabilities $p_1 := p_1^1$ and $p_2 := p_2^2$, we now need to determine the composite transition probabilities $\overline{p}_1 := p_1^{(1,2)}$ and $\overline{p}_2 := p_2^{(1,2)}$. Without loss of generality, let us consider $\overline{p}_1$, which we initially determine as follows:

$$\overline{p}_1 = p_1 = p_1(1-p_2) + p_1 p_2,$$

i.e., the probability of the transition for product 1 is the sum of the probabilities that the user is interested in product 1 and not in product 2 and that he/she goes to product 1, i.e., $p_1(1 - p_2)$, and that she is interested in both products and goes to both of them, i.e., $p_1 p_2$.

Now a user cannot, however, click on both recommendations at the same time; so it is reasonable to model the second case as $p_1 p_2 \frac{p_1}{p_1+p_2}$. The interest in the case of both recommendations is thus multiplied by the probability that in this case, the user decides in favor of product 1. This yields

$$\overline{p}_1 = p_1(1-p_2) + p_1 p_2 \frac{p_1}{p_1+p_2}.$$

Similarly, we obtain for $\overline{p}_2$

$$\overline{p}_2 = (1-p_1)p_2 + p_1 p_2 \frac{p_2}{p_1+p_2}.$$

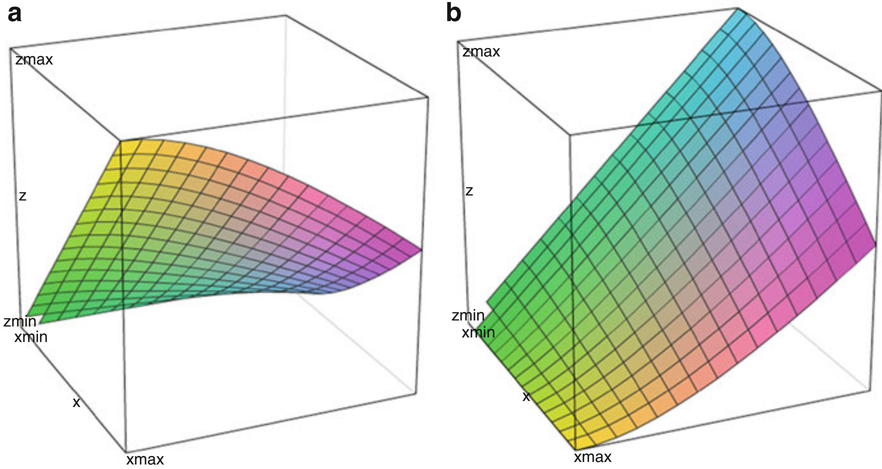


Fig. 4.4 Plot of the composite probabilities $\overline{p_1}$ and $\overline{p_2}$ as functions of the single probabilities p_1 and p_2

Both functions are illustrated in Fig. 4.4, wherein $\overline{p_1}$ is to be found in (a) and $\overline{p_2}$ in (b).

Example 4.1 We stick to the case of two recommendations. Suppose that for a single recommendation a_1 that probability that a user accepts this recommendation is 90 %, i.e., $p_1 = 0.9$. Let further the acceptance probability of a second single recommendation a_2 be 40 %, i.e., $p_2 = 0.4$.

Then, if both recommendations a_1 and a_2 are issued, the probability that the user accepts this first recommendation a_1 is

$$\overline{p_1} = 0.9(1 - 0.4) + 0.9 \cdot 0.4 \frac{0.9}{0.9 + 0.4} \approx 0.79.$$

Similarly, we obtain the probability that the user accepts the second recommendation a_2 as

$$\overline{p_2} = (1 - 0.9)0.4 + 0.9 \cdot 0.4 \frac{0.4}{0.9 + 0.4} \approx 0.15.$$

Thus, these probabilities are lower than their single recommendation counterparts p_1 and p_2, respectively. This is reasonable because of the obvious relation $\overline{p_1} + \overline{p_2} \leq 1$ (the user can accept at most one recommendation). We also see the nonlinearity of the approach: while the ratio of the initial probabilities was $p_1/p_2 = 0.7/0.4 = 2.25$, it now has increased to $\overline{p_1}/\overline{p_2} = 0.79/0.15 = 5.27$.

For comparison, we also consider our linear approach. Since we do not know the cross-product probabilities, we assume them to be zero, i.e., $p_1^2 = p_2^1 = 0$. Then (4.3) leads to

$$\overline{p_1} = \frac{1}{2}\left(p_1^1 + p_1^2\right) = \frac{1}{2}p_1 = 0.45$$

for the first recommendation and

$$\overline{p_2} = \frac{1}{2}\left(p_2^1 + p_2^2\right) = \frac{1}{2}p_2 = 0.2$$

for the second recommendation. Both probabilities are obviously lower than 50 %.

We now see the problem: if, e.g., $\overline{p_1}$ has been estimated as 0.7, i.e., for both recommendations a_1 and a_2 issued, the user in 70 % selects the first one, we would obtain $p_1 = 2\overline{p_1} = 1.4$. This is obviously nonsensical. For this reason, the linear approach can only by used meaningfully if we use all probabilities p_j^i instead of just those associated with the recommendations p_i. ■

The entire expression can be extended without principal difficulty to k recommendations displayed together, even though the notation is somewhat awkward:

$$p_l^{(1,\ldots,k)} = F_l(p_1,\ldots,p_k) = \sum_{Q\in\{l\}\cup\mathsf{P}\left(\underline{k}\setminus\{l\}\right)} \prod_{i\in Q} p_i \prod_{j\in\underline{k}\setminus Q}\left(1-p_j\right)\frac{p_l}{\sum_{i\in Q} p_i} \tag{4.4}$$

Here, $\underline{k} = \{1,\ldots,k\}$ is the index set of feasible recommendations, and P its power set, so $\{l\}\cup\mathsf{P}\left(\underline{k}\setminus\{l\}\right)$ is the set of subsets of $\underline{k}$ that do not contain l united with the singleton set of l.

In terms of our initial terminology, (4.4) may be expressed as follows:

$$p_{ss'}^{\overline{a}} = p_{ss_{a_l}}^{(a_1,\ldots,a_k)} = p_l^{(1,\ldots,k)} = F_l(p_1,\ldots,p_k) =: F_{a_l}\left(p_{ss_{a_1}}^{a_1},\ldots,p_{ss_{a_k}}^{a_k}\right), \quad s' = s_{a_l}. \tag{4.5}$$

This implies that as opposed to the linear approach (4.3), the approach (4.5) works out only for those successor products s_{a_l} that are associated with one of the single recommendations a_l. A further disadvantage compared to the linear case is, of course, the nonlinearity itself.

If we combine the probabilities as vectors $\overline{\mathbf{p}} = \left(p_1^{(1,\ldots,k)}\ldots p_k^{(1,\ldots,k)}\right)^T$ and $\mathbf{p} = (p_1\ \ldots\ p_k)^T$, this mapping of single into multiple recommendation probabilities is expressed as a vector function $\mathbf{F} = (F_1\ \ldots\ F_k)^T$

$$\overline{\mathbf{p}} = \mathbf{F}(\mathbf{p}). \tag{4.6}$$

Since in practice, however, we can determine only $\overline{\mathbf{p}}$, we need the inverse

$$\mathbf{p} = \mathbf{F}^{-1}(\overline{\mathbf{p}}). \tag{4.7}$$

Unfortunately, the function $\mathbf{F}$ from (4.4) is a nonlinear function. It is, however, smooth and (component-wise) monotone. In addition, the number of

recommendations k is not too high in practice, mostly less than 10, so the inverse $\mathbf{F}^{-1}$ can be determined robustly using, e.g., Newton's method, especially because in the incremental methods, the preceding values of $\mathbf{p}$ can be used as starting values.

In practice, this gives rise to, e.g., the following modified method for determining the transition probabilities according to the adaptive approach (3.8) as we need them in the ADP Algorithm 3.3.

Let us first consider the simple update of the single probabilities ${}^{j}\mathbf{p}$, where j is the update step. This is displayed in Algorithm 4.1 (of course, there are many other ways to calculate the transition probabilities).

Algorithm 4.1: Updating the single probabilities

Input: vector of single probabilities ${}^{j}\mathbf{p}$, index of the accepted recommendation l (-1 if none has been accepted), step size α_j
Output: updated vector of single probabilities ${}^{j+1}\mathbf{p}$

1: **procedure** UPDATE_P_SINGLE(${}^{j}\mathbf{p}$, l, α_j)
2: **for** $i = 1, \ldots, k$ **do**
3: **if** $i = l$ **then**
4: ${}^{j+1}p_i := {}^{j}p_i + \alpha_j(1 - {}^{j}p_i)$
5: **else**
6: ${}^{j+1}p_i := {}^{j}p_i + \alpha_j(0 - {}^{j}p_i)$
7: **end if**
8: **end for**
9: **return** ${}^{j+1}\mathbf{p}$
10: **end procedure**

For the probabilities of multiple recommendations, only the single probabilities ${}^{j}\mathbf{p}$ are stored internally. After issuing the multiple recommendation, we first compute the expected composite probabilities ${}^{j}\overline{\mathbf{p}} = \mathbf{F}({}^{j}\mathbf{p})$. Then, we update the latter according to the accepted and rejected recommendations by virtue of Algorithm 4.1, and we obtain the updated composite probabilities ${}^{j+1}\overline{\mathbf{p}}$. The latter, in turn, figure in the equation ${}^{j+1}\mathbf{p} = \mathbf{F}^{-1}\left({}^{j+1}\overline{\mathbf{p}}\right)$, which is solved by means of the Newton method with ${}^{j}\mathbf{p}$ as an initial guess, and we obtain the current single probabilities ${}^{j+1}\mathbf{p}$.

Algorithm 4.2: Updating the single probabilities from multiple recommendations

---

Input: vector of single probabilities ${}^{j}\mathbf{p}$, issued recommendations $\overline{a} = (a_1, \ldots, a_k)$, index of the accepted recommendation l (-1 if none has been accepted), step size α_j
Output: updated vector of single probabilities ${}^{j+1}\mathbf{p}$

(continued)

Algorithm 4.2: (continued)

1: **procedure** UPDATE_P_MULTI($^j\mathbf{p}, \overline{a}, l, \alpha_j$)
2: $^j\overline{\mathbf{p}} := \mathbf{F}(^j\mathbf{p})$ ▷ conversion into composite probabilities
3: $^{j+1}\overline{\mathbf{p}} :=$ UPDATE_P_SINGLE($^j\overline{\mathbf{p}}, l, \alpha_l$) ▷ update of composite probabilities
4: $^{j+1}\mathbf{p} := \mathbf{F}^{-1}(^{j+1}\overline{\mathbf{p}})$ ▷ conversion into single probabilities
5: **return** $^{j+1}\mathbf{p}$
6: **end procedure**

In practice, the function $\mathbf{F}$ together with its inverse $\mathbf{F}^{-1}$ turns out to be very helpful tools for converting back and forth between the transition probabilities of single and multiple recommendations.

We conclude with a final remark. So far, we have only considered one fixed multiple recommendation $\overline{a} = (a_1, \ldots, a_k)$. However, in reality, different multiple recommendations $\overline{a}_i = (a_{i_1}, \ldots, a_{i_k})$ are issued subsequentially, and even the number of their single recommendations k may vary. Different multiple recommendations $\overline{a}_i$ and $\overline{a}_j$ often share similar single recommendations, i.e., $a_{i_l} = a_{j_m} =: a_{s'}$, leading to the update of the same single probabilities $p_{ss'}^{a_{s'}}$. This raises the questions of the consistency and stability of Algorithm 4.2 when applied to all recommendations.

Let V be the space of all multiple recommendations $p_{ss'}^{\overline{a}_i}$ and their "recommended" states s':

$$V := \left\{ \left(p_{ss'}^{\overline{a}_i}\right)_{s' \in S_{\overline{a}_i}, i \in \mathrm{P}_s} \middle| 0 \le p_{ss'}^{\overline{a}_i} \le 1, \sum_{s' \in S_{\overline{a}_i}} p_{ss'}^{\overline{a}_i} \le 1 \right\}.$$

Here, we consider all multiple recommendations $\overline{a}_i$ over the power set $\mathrm{P}(\underline{n})$ where n is the number of all products and P_s is the corresponding index set. Further, let W be the space of all single recommendations $p_{ss'}^{a_{s'}}$:

$$W := \left\{ \left(p_{ss'}^{a_{s'}}\right)_{s' \in S_{A(s)}} \middle| 0 \le p_{ss'}^{a_{s'}} \le 1 \right\}.$$

We rewrite (4.6) by explicitly mentioning that it applies to a particular multiple recommendation $\overline{a}_i$:

$$\overline{\mathbf{p}}^{\overline{a}_i} = \mathbf{F}^{\overline{a}_i}(\mathbf{p}^{\overline{a}_i}), \quad \overline{\mathbf{p}}^{\overline{a}_i} := \overline{\mathbf{p}} = \left(p_{ss'}^{\overline{a}_i}\right)_{s'}, \ \mathbf{p}^{\overline{a}_i} := \mathbf{p} = \left(p_{ss'}^{a_{s'}}\right)_{s'}, \ \mathbf{F}^{\overline{a}_i} := \mathbf{F}.$$

Next, we extend $\mathbf{F}^{\overline{a}_i}$ to $\mathbf{F}_G^{\overline{a}_i}$ such that it works on the whole space W by simply ignoring all single probabilities not belonging to the recommendations of $\overline{a}_i$. By

$$\overline{\mathbf{p}}_G = \left(\overline{\mathbf{p}}^{\overline{a}_i}\right)_i = \left(\mathbf{F}_G^{\overline{a}_i}(\mathbf{p}_G)\right)_i =: \mathbf{F}_G(\mathbf{p}_G), \quad \overline{\mathbf{p}}_G \in V, \ \mathbf{p}_G \in W,$$

we have formally introduced the vector function $\mathbf{F}_G : W \to V$:

$$\overline{\mathbf{p}}_G = \mathbf{F}_G(\mathbf{p}_G). \tag{4.8}$$

In the same way, from (4.7), we derive the inverse $\mathbf{F}_G^{-1} : V_F \subset V \to W$:

$$\mathbf{p}_G = \mathbf{F}_G^{-1}(\overline{\mathbf{p}}_G). \tag{4.9}$$

Example 4.2 To make the discussion less abstract, we give a very simple example. Consider the case where we have only two products, i.e., $n = 2$. Then we have two single recommendations a_1 and a_2 and three multiple recommendations $\overline{a}_1 = (a_1),\ \overline{a}_2 = (a_2),\ \overline{a}_3 = (a_1, a_2)$. This yields the following probability spaces:

$$V := \left\{ \left\{ p_{ss_1}^{\overline{a}_1}, p_{ss_2}^{\overline{a}_2}, p_{ss_1}^{\overline{a}_3}, p_{ss_2}^{\overline{a}_3} \right\} \middle| 0 \le p_{ss'}^{\overline{a}_i} \le 1,\ p_{ss_1}^{\overline{a}_3} + p_{ss_2}^{\overline{a}_3} \le 1 \right\},$$
$$W := \left\{ \left\{ p_{ss_1}^{a_{s_1}}, p_{ss_2}^{a_{s_2}} \right\} \middle| 0 \le p_{ss'}^{a_{s'}} \le 1 \right\}.$$

Let us consider the first "multiple" recommendation $\overline{a}_1 = (a_1)$. Then the relation

$$\overline{\mathbf{p}}^{\overline{a}_1} = \mathbf{F}^{\overline{a}_1}\left(\mathbf{p}^{\overline{a}_1}\right)$$

simply translates into

$$\left(p_{ss_1}^{\overline{a}_1} \right) = \mathbf{F}^{\overline{a}_1}\left(p_{ss_1}^{a_1} \right) = \left(p_{ss_1}^{a_1} \right).$$

For the global function $\mathbf{F}_G^{\overline{a}_1}$, we get

$$\left(p_{ss_1}^{\overline{a}_1} \right) = \mathbf{F}_G^{\overline{a}_1}\begin{pmatrix} p_{ss_1}^{a_1} \\ p_{ss_2}^{a_2} \end{pmatrix} = \left(p_{ss_1}^{a_1} \right).$$

Same holds for the second "multiple" recommendation $\overline{a}_2 = (a_2)$. For the third (and only real) multiple recommendation $\overline{a}_3 = (a_1, a_2)$, the mapping $\overline{\mathbf{p}}^{\overline{a}_3} = \mathbf{F}^{\overline{a}_3}\left(\mathbf{p}^{\overline{a}_1}\right)$ now becomes more complex:

$$\begin{pmatrix} p_{ss_1}^{\overline{a}_3} \\ p_{ss_2}^{\overline{a}_3} \end{pmatrix} = \mathbf{F}^{\overline{a}_3}\begin{pmatrix} p_{ss_1}^{a_1} \\ p_{ss_2}^{a_2} \end{pmatrix} = \begin{pmatrix} p_{ss_1}^{a_1}\left(1 - p_{ss_2}^{a_2}\right) + p_{ss_1}^{a_1} p_{ss_2}^{a_2} \dfrac{p_{ss_1}^{a_1}}{p_{ss_1}^{a_1} + p_{ss_2}^{a_2}} \\ p_{ss_2}^{a_2}\left(1 - p_{ss_1}^{a_1}\right) + p_{ss_1}^{a_1} p_{ss_2}^{a_2} \dfrac{p_{ss_2}^{a_2}}{p_{ss_1}^{a_1} + p_{ss_2}^{a_2}} \end{pmatrix}.$$

In this case, $\mathbf{F}^{\overline{a}_3}$ is also our global function, i.e., $\mathbf{F}_G^{\overline{a}_3} = \mathbf{F}^{\overline{a}_3}$. Thus, we arrive at the complete global mapping:

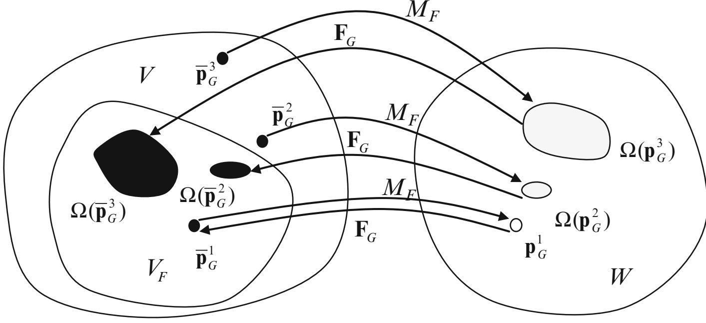


Fig. 4.5 Illustration of the stability of the mapping M_F. The first probability vector $\overline{\mathbf{p}}_G^1$ is inside V_F and modeled accurately by $\mathbf{p}_G^1$. The second probability vector $\overline{\mathbf{p}}_G^2$ is slightly outside V_F and modeled by the domain $\Omega\left(\mathbf{p}_G^2\right)$. Backward transformation by $\mathbf{F}_G$ leads to the domain $\Omega\left(\overline{\mathbf{p}}_G^2\right)$ which is quite close to $\overline{\mathbf{p}}_G^2$. The third vector $\overline{\mathbf{p}}_G^3$ is far from V_F leading to bad approximation results

$$\begin{pmatrix} p_{ss_1}^{\overline{a}_1} \\ p_{ss_2}^{\overline{a}_2} \\ p_{ss_1}^{\overline{a}_3} \\ p_{ss_2}^{\overline{a}_3} \end{pmatrix} = \begin{pmatrix} \mathbf{F}_G^{\overline{a}_1}\left(p_{ss_1}^{a_1}\right) \\ \mathbf{F}_G^{\overline{a}_2}\left(p_{ss_2}^{a_2}\right) \\ \left(\mathbf{F}_G^{\overline{a}_3}\left(p_{ss_1}^{a_1}, p_{ss_2}^{a_2}\right)\right)_1 \\ \left(\mathbf{F}_G^{\overline{a}_3}\left(p_{ss_1}^{a_1}, p_{ss_2}^{a_2}\right)\right)_2 \end{pmatrix} = \begin{pmatrix} p_{ss_1}^{a_1} \\ p_{ss_2}^{a_2} \\ p_{ss_1}^{a_1}\left(1 - p_{ss_2}^{a_2}\right) + p_{ss_1}^{a_1} p_{ss_2}^{a_2} \dfrac{p_{ss_1}^{a_1}}{p_{ss_1}^{a_1} + p_{ss_2}^{a_2}} \\ p_{ss_2}^{a_2}\left(1 - p_{ss_1}^{a_1}\right) + p_{ss_1}^{a_1} p_{ss_2}^{a_2} \dfrac{p_{ss_2}^{a_2}}{p_{ss_1}^{a_1} + p_{ss_2}^{a_2}} \end{pmatrix}$$

$$= \mathbf{F}_G \begin{pmatrix} p_{ss_1}^{a_1} \\ p_{ss_2}^{a_2} \end{pmatrix}.$$

In the same way, the global mapping $\mathbf{F}_G^{-1}$ can be derived. Here, $\left(\mathbf{F}_G^{\overline{a}_1}\right)^{-1} = \mathbf{F}_G^{\overline{a}_1}$, $\left(\mathbf{F}_G^{\overline{a}_2}\right)^{-1} = \mathbf{F}_G^{\overline{a}_2}$, and $\left(\mathbf{F}_G^{\overline{a}_2}\right)^{-1}$ is defined as $\mathbf{F}^{-1}$ according to (4.7). It is easy to see that $\mathbf{F}_G^{-1}\mathbf{F}_G\mathbf{p}_G = \mathbf{p}_G \;\; \forall \; \mathbf{p}_G \in W$.

However, since the space V consists of four free probabilities $p_{ss_1}^{\overline{a}_1}, p_{ss_2}^{\overline{a}_2}, p_{ss_1}^{\overline{a}_3}, p_{ss_2}^{\overline{a}_3}$ (apart from the restriction $p_{ss_1}^{\overline{a}_3} + p_{ss_2}^{\overline{a}_3} \leq 1$) and the space W only of the two $p_{ss_1}^{a_{s_1}}, p_{ss_2}^{a_{s_2}}$, the image space $V_F = \mathbf{F}_G(W)$ is really a proper subset of V, i.e., there exist probabilities $\mathbf{p}_G \in V: \;\; \mathbf{p}_G \notin V_F$. ■

Thus, $\mathbf{F}_G^{-1}$ is really the inverse of $\mathbf{F}_G$. Since ran $\mathbf{F}_G = V_F \subset V$ is a proper subset of V, the inverse $\mathbf{F}_G^{-1}$ is not defined over all elements of V. Unfortunately, our observations are made in the space V. But technically, the mapping M_F defined by Algorithm 4.2 can be applied to all vectors of V.

The good news is: since $\mathbf{F}_G$ and $\mathbf{F}_G^{-1}$ are continuous and V, W are convex, as long as our assumptions hold (Assumption 4.3 and the one of the nonlinear approach), i.e., $\overline{\mathbf{p}}_G \in V_F$, the result of the mapping M_F converges to the right solution $\mathbf{p}_G = \mathbf{F}_G^{-1}(\overline{\mathbf{p}}_G)$. If our assumptions are violated, i.e., $\overline{\mathbf{p}}_G \notin V_F$, the result of M_F is oscillating. Yet, the second good news is: the closer the distance of $\overline{\mathbf{p}}_G$ to V_F, the less is the oscillation. This is illustrated in Fig. 4.5.

This stability result means that the less our model assumptions are violated, the better our observed probabilities can be estimated. Similar reasoning applies to the methods for the estimation of transition probabilities presented in Sect. 5.2.3.

4.3 Remarks on the Modeling

In what follows, we would like to study the remarks from Sect. 3.9 with regard to the above devised model of recommendation engines.

As we are dealing with recommendation engines with episodic tasks, the question for the terminal state arises. Indeed, we have already seen the latter in Figs. 4.2 and 4.3. It is, indeed, meaningful for two reasons: first, it has a clear interpretation with regard to content, as it assigns to each product the probability that a user terminates the session afterward, i.e., leaves the shop. Second, it is relevant for the computation of transition probabilities. According to (3.2), these must sum up to one, as P is stochastic. Indeed, one could ignore the end of the session when computing P, but this would result in adulterated transition probabilities. Since the latter are multiplied with the rewards in the Bellman equation (3.4), their actual magnitude matters.

As opposed to all the other states, which represent products, there is, of course, no action affiliated to the terminal state – since this would mean to suggest the user to leave the shop.

Another issue is the question of whether it is meaningful to consider recommendations from products to themselves, i.e., p_{ss}. This corresponds to the representation of rules of the form $s \to s$. We remind the reader that this is a sufficient condition for primitivity of the matrix P (together with irreducibility, which we shall address later on). At the first glance, these rules do not convey much information; they only signify that the user repeatedly calls the product up, i.e., hits the refresh button. (This is different when we operate on the level of categories as in Chap. 6.) On the other hand, they are, for the same reasons as the terminal state, relevant with regard to the computation of transition probabilities. Hence, the internal usage of these transitions is recommendable. They must, however, not serve as recommendations, as they would give rise to products recommending themselves.

Finally, let us turn to the question of irreducibility. In most practical applications, it does not hold. In Chaps. 6, 7, 8, and 9 on hierarchical methods and factorizations, we shall, however, deal with procedures that enable to compute an almost unlimited amount of recommendations for each product, i.e., transitions satisfying $p_{ss'} > 0$. This may easily be exploited to render P irreducible. At the same time, irreducibility may also have positive effects since it decomposes the global problem into uncoupled subproblems. An example is Theorem 6.1 about the convergence of the multigrid method. Thus, the value of irreducibility has to be checked depending on the used method.

Let us summarize: it is reasonable to include the terminal state in the model of the RE. So is it, in general, to capture cycles of length 1. Invoking special tools, it is possible to ensure that P be irreducible. Thus, the essential conditions for convergence of the TD algorithm are satisfied.

4.4 Verification Methods

Subsequently, we shall address the verification methods for RL procedures that we shall use in Chap. 5.

To this end, we employ historical data from real online shops. These consist of log files that have been generated by the recommendation module of prudsys RDE (Sect. 12.3.1). The files contain basic transactions, as well as the recommendations issued by the RDE (for the recommendation group). The most relevant columns of the log files (CSV format) are listed in Table 4.1 (further columns like user ID and channel ID have been omitted).

The control group serves the purpose of comparing the efficiencies of running the shop with or without the recommendation engine (Chap. 11).

A row of a log file then looks as follows (for confidentiality reasons, all entries have been made up):

```
2009-04-18 11:36:21, 386AC17893,0,0045322,17.48,0,0,7889965
05564556
```

This row tells us that on April 18, 2009, at 11:36:21 AM, the product 0045322, the price of which amounts to 17.48 EUR, was viewed within the session 386 AC17893, which belongs to the recommendation group, and the products 7889965 and 05564556 were thereupon recommended.

For the online recommendation algorithms, we make use of the following online verification method: the historical data are parsed session-wise, and the transactions are carried out in their original order of sequence. For each product view, the engine's recommendations are requested. Subsequently, the latter are compared with the actual views, adds to basket, and purchases and the surveys of correct predictions as well as forecasted revenue are updated. Table 4.2 illustrates the procedure by an example of a session.

In step 1, the user views product A and the RE issues the recommendations C and B. The counter of views is increased by one. In the second step, the user switches to product D and the RE recommends E and A; the view counter is again increased by

Table 4.1 Description of the columns of the transaction files generated by prudsys RDE

Column label	Description
time	Date and time of the transaction
transactID	Unique ID of the session
group	The session's group affiliation (0, recommendation; 1, control group)
itemID	Unique ID of the product
price	Price of the product
transType	Type of transaction (0, click; 1, add to basket; 2, purchase; etc.)
order	Number of purchased units (only for *transType* = 2)
itemsAction	Recommended products

Table 4.2 Illustration of the simulation

			Real				Correct forecast			
Step	TA	REs	Views	Baskets	Buys	Rev.	Views	Baskets	Buys	Rev.
1	A	C, B	1	0	0	0	0	0	0	0
2	D	E, A	2	0	0	0	0	0	0	0
3	D basket		2	1	0	0	0	0	0	0
4	A	C, E	3	1	0	0	1	0	0	0
5	A basket		3	2	0	0	1	1	0	0
6	A bought		3	2	1	35.00	1	1	1	35.00
Quality of forecast							33 %	50 %	100 %	100 %

one and attains the value 2. Since D is none of the previous recommendations, the counter of views correctly predicted by the RE remains at 0. In step 3, the product D is added to basket; the basket counter is increased.

In step 4, another view of A takes place; this time, this coincides with the second recommendation issued by the RE, and hence, its view counter is now also increased by one. Since, in step 5, the correctly predicted product has been added to basket, the RE's basket counter also increases by one. In the last step 6, only the product recommended by the RE is eventually bought. Therefore, the buy counter increases by one and the forecasted revenue by the price of the purchased product (here, 35 EUR). By and large, the RE has correctly predicted 33 % of the actual views, 50 % of the products that have actually been added to basket, and 100 % of the purchased products.

Please mind that in step 4, the RE issues recommendations that differ from those for the same product in step 1 (namely, C and E instead of C and B). This is because the RE learns dynamically. As a result of this dynamic, the measurements obtained in simulation mode differ slightly from those obtained in online mode. Nevertheless, the order of magnitude is the same, and with an increasing number of transactions, the measurements assimilate toward each other.

Besides the above-described online procedure for assessing the quality of forecast, we also occasionally deploy offline validation procedures. Here, we subdivide the transaction data into a training set and a test set, which are, respectively, drawn from transaction logs of different days, for example, the file of April 18th for training and that of April 19th for testing. Sometimes, however, we also carry out the training-test decomposition within the same session, which is, e.g., accomplished by using all but the last n transactions of each session for training and the remaining ones for testing.

We now learn the recommendation model from the training set, as in the online test, and apply it to forecasting on the test set. The qualities of forecasts are computed with respect to the same characteristic figures for views, adds to basket, purchases, and revenue. In terms of the terminology commonly used in the area of verification of recommendation algorithms [SKKR00], we measure the *precision* (what we have used so far). We abstain from using the *recall,* another popular measure, which describes the coverage ratio of the test set by recommendations. To avoid overloading the already fairly complex testing procedures, we also refrain from further measures.

4.5 Summary

This chapter was devoted to the application of reinforcement learning to recommendation engines. We have introduced RE-specific *empirical* assumptions to reduce the complexity of RL in order to make it applicable to real-life recommendation problems. Especially, we provided a new approach for estimating transition probabilities of multiple recommendations from that of single recommendations. Nevertheless, the estimation of transition probabilities for single recommendations was left as an open problem that will be addressed in the next chapter. Finally, we introduced a simple framework for testing online recommendations.

Chapter 5
How Engines Learn to Generate Recommendations: Adaptive Learning Algorithms

Abstract This chapter is mainly devoted to the question of estimating transition probabilities taking into account the effect of recommendations. It turned out that this is an extremely complex problem. The central result is a simple empirical assumption that allows reducing the complexity of the estimation in a way that is computationally suitable to most practical problems. The discussion of this approach gives a deeper insight into essential principles of realtime recommendation engines. Based on this assumption, we propose methods to estimate the transition probabilities and provide some first experimental results. Although the results look promising, more advanced techniques are highly desirable. Such techniques like hierarchical and factorization methods are presented in the following chapters.

In Chap. 4, we have gathered all the ingredients to apply the reinforcement learning approaches described in Chap. 3 to recommendation engines. Except for one thing, we still do not have the transition probabilities $p_{ss'}^{a}$! These are really problematic, because we would have to save not only all the product transitions $s \to s'$ but also those for all recommendations a that are generated. For large web shops, in particular, this can result in huge numbers of rules, of the order of magnitude of all transactions and thus containing thousands of millions of rules. Not only would this be technically difficult, it would also be extremely unstable, because most of those rules would have hardly any statistical basis.

Therefore, we must make some empirical assumptions in order to achieve plausible simplifications. The simplest approach is the classical one: we simply ignore the recommendations a. We therefore work only with the transition probabilities $p_{ss'}$, i.e., without considering the actions a. We refer to $p_{ss'}$ as *unconditional transition probabilities*, in contrast to *conditional transition probabilities* $p_{ss'}^{a}$.

A. Paprotny and M. Thess, *Realtime Data Mining: Self-Learning Techniques for Recommendation Engines*, Applied and Numerical Harmonic Analysis,
DOI 10.1007/978-3-319-01321-3_5, © Springer International Publishing Switzerland 2013

We designate the corresponding approach using unconditional transition probabilities as the *unconditional* or *probabilistic* approach in contrast to the *conditional* approach using conditional probabilities. Below, we will derive expressions for both approaches and then combine them in a useful form.

5.1 Unconditional Approach

The probabilistic approach thus corresponds to the classical view described by Approach 1 in Chap. 2. For this, we apply RL approaches formally, so as, for instance, to factor in rewards and chain optimization, i.e., we use the formal advantages of RL in order to broaden the classical approach. Thus, instead of recommending a product s', which after viewing s is bought most frequently, we incorporate its reward r. That seems logical. In addition, we give preference to those products which, *including subsequent purchases* by existing customers, lead to the highest sales. So, chain optimization seems reasonable as well.

In the strict RL sense, that is nonsensical. We only learn the policy which users pursue on their own initiative without regard to the recommendations, and we reinforce this. But as we have seen, the generalized general policy iteration at least offers a general justification of this approach. And the results are good in practice.

The approach thus tends to implement the recommendations with the highest unconditional transition probabilities and action values. Let us start with the "simple" case (3.5). The direct use of (3.5) would take us no further forward here, since

$$q^{\pi}(s,a) = \sum_{s'} p_{ss'}^{a} r_{ss'}^{a} = \sum_{s'} p_{ss'} r_{ss'} = q^{0}(s,a) = q^{0}(s)$$

would yield the same action value for all recommendations a. In fact, if the transition probabilities were independent of issuing recommendations, one would not need any recommendations at all. Therefore, we make the following assumption:

Assumption 5.1 (Unconditional probability property): For each state transition from s to the state s_a associated with the action a, the transition probability $p_{ss_a}^{a}$ is considered as proportional to the unconditional probability p_{ss_a}, i.e., $p_{ss_a}^{a} = dp_{ss_a'}$ with the factor $d > 1$. For any other state transition from s to s' under the action a, the transition probability $p_{ss'}^{a}$ is likewise considered as proportional to the unconditional probability $p_{ss'}$, i.e., $p_{ss'}^{a} = cp_{ss'}$, but with the factor $c < 1$.

In other words, delivering recommendation a increases the probability of the transition for the product s_a associated with it. The higher the unconditional

probability p_{ss_a}, the higher the conditional probability $p^a_{ss_a}$. The transition probabilities $p^a_{ss'}$ for all other products $s'$are, conversely, influenced negatively by delivering a, since (3.2) applies. Of course, our probability property is of a somewhat abstract nature, since, because of the equation system being strongly overdetermined, c and d cannot be uniquely determined in general. Nevertheless, it is helpful for qualitative discussion.

Thus (3.5) takes the following form:

$$q^{\pi}(s,a) = p^a_{ss_a} r_{ss_a} + \sum_{s' \neq s_a} p^a_{ss'} r_{ss'} = d p_{ss_a} r_{ss_a} + \sum_{s' \neq s_a} c p_{ss'} r_{ss'}$$

and yields

$$\begin{aligned} q^{\pi}(s,a) - q^{\pi}(s,b) &= (d-c)\left(p_{ss_a} r_{ss_a} - p_{ss_b} r_{ss_b}\right) \\ &> 0 \Leftrightarrow p_{ss_a} r_{ss_a} > p_{ss_b} r_{ss_b}. \end{aligned}$$

The formula for calculating the action value can be derived immediately from this:

$$q^P(s,a) = p_{ss_a} r_{ss_a}, \tag{5.1}$$

which we will refer to as the (simplified*) P-Version* below. A recommendation is thus strong if it is either frequently clicked on, or carries a high reward, or both.

Approach (5.1) may now be expanded for case $\gamma > 0$ in accordance with (3.6), whereupon we obtain the full P-Version:

$$q^P(s,a) = p_{ss_a} r_{ss_a} + \gamma p_{ss_a} \sum_{a'} \pi\left(s_a, a'\right) q^P\left(s_a, a'\right). \tag{5.2}$$

As described in Chap. 3, we can now update p_{ss_a} and r_{ss_a} in realtime and thus calculate (5.1) and (5.2) either in an off-line fashion or (5.1) directly online or (5.2) online using ADP methods like Algorithm 3.3.

Alternatively, for the model-free case, we can very easily apply the TD-Version in a similar way, although we have to employ a few empirical tricks to overcome the problem of multiple recommendations. In practice, the unconditional approach works quite successfully; the P-Version works better than the TD-Version.

Example 5.1 Subsequently, we shall illustrate the results of the unconditional approach by means of a practical example. Here, we shall employ the online verification methods described in Sect. 4.4. We forgo the chain property, i.e., we assign $\gamma = 0$. Thus, we use the simple P-Version according to (5.1) with an adaptive update of the transition probabilities p_{ss_a} and rewards r_{ss_a}. To observe unbiased user behavior, only transactions of sessions belonging to the control group have been included in the analysis.

Table 5.1 Simulation results comparing prediction rates of manual recommendations with those of recommendations generated by the P-Version

	Manual	P-Version
Clicks	4.15	8.63
Baskets	3.57	8.24
Orders	3.71	8.70
Revenue	4.30	8.37

Table 5.2 Prediction rates against the number of transaction steps

Steps	PVs manual	Revenue manual	PVs P-Version	Revenue P-Version
1,000	4.49	3.26	5.28	3.64
2,000	4.37	5.26	5.76	4.90
3,000	4.45	4.93	6.19	5.49
4,000	4.40	4.44	6.54	6.10
5,000	4.36	4.25	6.76	6.56
6,000	4.34	4.27	7.08	6.88
7,000	4.36	4.28	7.23	7.23
8,000	4.32	4.30	7.38	7.38
9,000	4.32	4.40	7.43	7.39
10,000	4.31	4.36	7.45	7.31
11,000	4.26	4.36	7.42	7.44
12,000	4.19	4.37	7.46	7.28

Here, on one hand, the back-then recommendations manually devised by the shop operators have been issued. As these are static, we obtain the actual prediction rate of the manual recommendations. On the other hand, we study recommendations of the P-Version. To render the task more challenging, the algorithm literally starts from scratch, i.e., the learning starts with the simulation data.

The data are courtesy of a major mail-order company. They have been purified by removing invalid products as well as multiple calls of product views, which all in all results in a higher prediction rate. Approximately 600,000 transactions with 4,500 products are left in the purified set. The corresponding prediction rates (in terms of percentage) are displayed in Table 5.1.

The P-Version turns out to achieve about twice the rate of the manual recommendations, which gives evidence for the quality and learning performance of the approach.

As a further refinement, the prediction rates of both types of recommendations with respect to product views (PVs) and revenue have been simulated for the first 12,000 transaction steps. The results (in terms of percentage) are displayed in Table 5.2 and Fig. 5.1.

The P-Version turns out to surpass the manual recommendations in terms of all measures of prediction quality after approximately 5,000 steps. Moreover, the learning rate of the former exhibits a logarithmic decay thereafter. After only

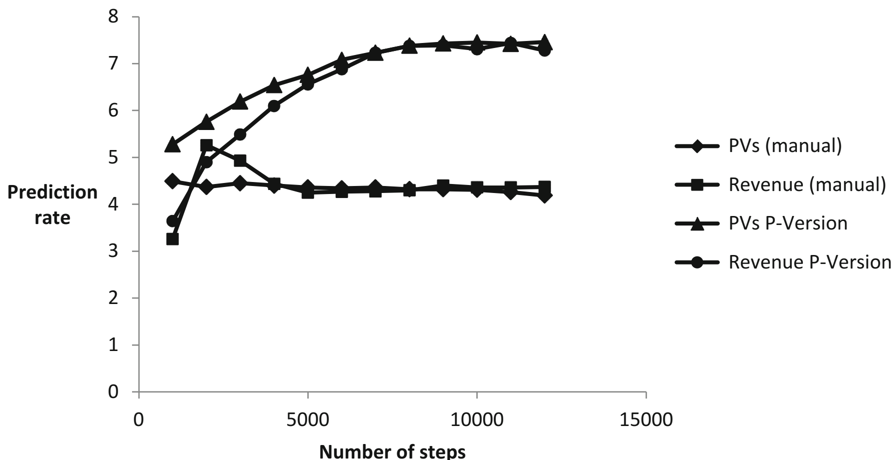


Fig. 5.1 Prediction rates against the number of transaction steps

12,000 steps, the P-Version attains a rate of almost 7.5 %, which in the course of the day increases to no more than about 8.5 %. Taking preceding days into account yields an increase of only about 10 %. ■

5.2 Conditional Approach

Next, we proceed a step further than the unconditional approach and consider the solution of (3.5) as a control problem, i.e., using the conditional probabilities $p_{ss'}^{a}$.

To this end, we need to simplify the determination of the transition probabilities $p_{ss'}^{a}$. We make the following assumption introduced by Thess:

> **Assumption 5.2 (Conditional probability property):** For each state transition from s to s' under the action a, for which s' is not the state s_a associated with the action a, the transition probability $p_{ss'}^{a}$ is considered as proportional to the unconditional probability $p_{ss'}$, i.e., $p_{ss'}^{a} \sim p_{ss'}$.

It follows that (in addition to the direct determination of $p_{ss_a}^{a}$)

$$p_{ss'}^{a} = c(s,a)p_{ss'}, \quad s' \neq s_a \tag{5.3}$$

where $c(s, a)$ is a constant depending only on s and a. We will call the approach based on Assumption 5.2 the *DP-Version*.

In other words, if a recommendation a is delivered but not accepted, and instead of the recommended product s_a, the user clicks on another product s', we assume that the user selected this independently of the current recommendation a.

This description is somewhat simplified, as the delivery of the recommendation a has already been incorporated into the scaling via the coefficient $c(s, a)$; however, the relationship of the values for the conditional probabilities $p^{a}_{ss'}$ in respect of s' is determined by each of the unconditional probabilities $p_{ss'}$. We therefore assume that there are some users who are not influenced by recommendations (because, for instance, they are working from a shopping list). Although their overall influence is already limited by the recommendation a (since there is another user group which is open to the recommendations, hence the scaling factor c), their specific behavior in the transition from s to s' is unaffected.

It follows that instead of saving all transition probabilities $p^{a}_{ss'}$, we only need to save the conditional probabilities $p^{a}_{ss_a}$ (i.e., between recommendation and recommended product) together with the unconditional probabilities $p_{ss'}$. Technically, this means that for every rule $s \rightarrow s'$, both $p^{a_{s'}}_{ss'}$ (i.e., the probability that the recommendation of the product s' be accepted) and $p_{ss'}$ (i.e., the probability that a user goes from product s to product s' without a recommendation s') are saved.

A similar method was proposed some time ago in [SHB05], but in a more incomplete form. In particular, the coefficients $c(s, a)$ were merely modeled as $c(s)$ therein, which prevents adequate handling of down-selling, as we shall shortly see.

Let us firstly consider again the solution to (3.5), in order to determine the optimal recommendations. So as now to determine the complete transition probabilities $p^{a}_{ss'}$, we use the relationship (3.2), and in combination with (5.3), we obtain

$$p^{a}_{ss_a} + c(s,a) \sum_{s' \neq s_a} p_{ss'} = 1. \tag{5.4}$$

Furthermore, from

$$\sum_{s'} p_{ss'} = 1$$

follows the relation

$$\sum_{s' \neq s_a} p_{ss'} = 1 - p_{ss_a},$$

which, when used in (5.4), finally enables the calculation of $c(s, a)$:

$$c(s,a) = \frac{1 - p^{a}_{ss_a}}{1 - p_{ss_a}}, \tag{5.5}$$

which represents the change of all transition probabilities induced by recommendation a (except for the target state s_a).

Using (5.5), (3.5) can be determined:

$$q^{\pi}(s,a) = p^{a}_{ss_a} r_{ss_a} + c(s,a) \sum_{s' \neq s_a} p_{ss'} r_{ss'}. \tag{5.6}$$

Expressed in words, for the product s, the action value of the recommendation a is equal to the probability of the acceptance of this recommendation multiplied by the corresponding reward plus the weighted sum of the unconditional transition probabilities multiplied by their rewards. For simplicity's sake, we will occasionally refer to the first term $p^{a}_{ss_a} r_{ss_a}$ as the *conditional* and the second term (without the scaling factor) $\sum_{s' \neq s_a} p_{ss'} r_{ss'}$ as the *unconditional action value*.

Despite their apparent simplicity, equations (5.5) and (5.6) are extremely interesting. As with many fundamental relationship equations, they deserve detailed study. Consider, for instance, Maxwell's equations. Through their fundamental interpretation, James Clerk Maxwell was able to predict the existence of electromagnetic waves, which were later confirmed experimentally by Heinrich Hertz. By further studying them, Henri Poincare was able to derive the invariability of the speed of light, from which Albert Einstein finally developed the special relativity theory. Of course, (5.5) and (5.6) are not Maxwell's equations, and the authors are no Poincare or Einstein; nevertheless, it is rewarding to spend some more time on them.

5.2.1 Discussion

If we consider (5.6) more closely, we see that the conditional action value is the direct value of the recommendation, whereas the unconditional action value represents the value of the other product transitions. The scaling factor c controls the weighting between the two. While, therefore, the conditional action value reflects the success of the recommendation, the unconditional action value gives the "recommendation-free" potential (i.e., the product transitions that occur even without a recommendation), whose influence is reduced by the success of the recommendation.

If the transition probabilities $p^{a}_{ss_a}$ and p_{ss_a} are both small, $c(s, a)$ can be regarded as approximately 1, and both action values have the same weighting. As soon, however, as $p^{a}_{ss_a}$ is significantly greater than $p_{ss_{a'}}$ i.e., the recommendation a is strongly accepted, $c(s, a) << 1$ is true, and the influence of the unconditional action value is reduced as compared to that of the conditional one. If the improvement in the conditional action value overtakes the reduction in the unconditional, the recommendation is successful. On the other hand, the recommendation may suffer from down-selling: the popular recommendation captures more reward overall from the previous product transitions than it generates as new reward.

Conversely, it may occur, though rather seldom in practice, that the response to the recommendation a leads to a decrease in the associated transition probability (for instance, by cannibalizing multiple product recommendations). We then have $p_{ss_a}^a << p_{ss_a}$ and hence $c(s, a) >> 1$: the unconditional action value would be extremely strongly weighted – an apparently absurd effect. However, it should be noted here that because of the relationship $\sum_{s' \neq s_a} p_{ss'} = 1 - p_{ss_a}$, a large p_{ss_a} leads to a small unconditional action value, and we must perform a limit value consideration here. We will explore this in more depth in the course of the special cases of (5.6).

For a quantitatively better understanding of (5.6), let us consider for the product s the difference between the action values of two recommendations a and b:

$$q^{\pi}(s,a) - q^{\pi}(s,b) = p_{ss_a}^a r_{ss_a} + c(s,a) \sum_{s' \neq s_a} p_{ss'} r_{ss'} - p_{ss_b}^b r_{ss_b} - c(s,b) \sum_{s' \neq s_b} p_{ss'} r_{ss'}$$

$$= p_{ss_a}^a r_{ss_a} - p_{ss_b}^b r_{ss_b} + c(s,a) p_{ss_b} r_{ss_b} - c(s,b) p_{ss_a} r_{ss_a}$$

$$+ c(s,a) \sum_{s' \neq s_a, s_b} p_{ss'} r_{ss'} - c(s,b) \sum_{s' \neq s_a, s_b} p_{ss'} r_{ss'}$$

$$= \left[p_{ss_a}^a - c(s,b) p_{ss_a} \right] r_{ss_a} - \left[p_{ss_b}^b - c(s,a) p_{ss_b} \right] r_{ss_b} + [c(s,a) - c(s,b)] \sum_{s' \neq s_a, s_b} p_{ss'} r_{ss'}.$$

By preliminary use of the estimate $c(s,a) = \frac{1 - p_{ss_a}^a}{1 - p_{ss_a}} \approx 1$ and similarly $c(s, b) \approx 1$, we obtain

$$q^{\pi}(s,a) - q^{\pi}(s,b) \approx \underbrace{\left[p_{ss_a}^a - p_{ss_a} \right]}_{\Delta p^a} \underbrace{r_{ss_a}}_{r_a} - \underbrace{\left[p_{ss_b}^b - p_{ss_b} \right]}_{\Delta p^b} \underbrace{r_{ss_b}}_{r_b} = \Delta p^a r_a - \Delta p^b r_b. \quad (5.7)$$

If, for the sake of simplicity, we initially set all rewards to 1, we have

$$q^{\pi}(s,a) - q^{\pi}(s,b) \approx \Delta p^a - \Delta p^b.$$

Since we can generally assume that for a product s the probability of a product transition to a product s_y is higher if y is recommended, we have

$$p_{ss_y}^y > p_{ss_y},$$

and we obtain the following interpretation. The recommendation a is then certainly better than the recommendation b if the difference Δp^a between the transition probabilities increased by the product recommendation is greater than the similar difference Δp^b for the recommendation b. Instead therefore of making recommendations a with the highest transition probability p^a $(:= p_{ss_a})$ as in classical data mining, the recommendations a that are made are those with the highest difference between the

conditional and unconditional transition probability Δp^a. That is, those recommendations which most increase the transition probability. This is also logical, since the unconditional transition probabilities p^a are in fact achieved even without recommendations. By the way, it can be generally assumed that in most cases the relationship

$$p^a \sim \Delta p^a$$

has an approximate validity, which is why classical data mining methods work to some extent (see also previous section: $\Delta p^a = \left(p^a_{ss_a} - p_{ss_a}\right) = dp_{ss_a} - p_{ss_a} = (d-1)p_{ss_a} \sim p^a$).

If we now discard the restriction regarding the 1-rewards, then, according to (5.7), recommendation a will then certainly be better than recommendation b if the difference in action value $\Delta p^a r_a$ for the transition probability increased by the product recommendation is higher than the difference in action value $\Delta p^b r_b$ due to recommendation b. This is an extension of the preceding case and its content is clear.

It should, however, be remembered that even the simplification $c(s, a) \approx 1$ is not always valid in practice and, moreover, is not always useful. In order to understand this, consider just our two products a and b. We assume that a has a high reward r_a, but $\Delta p^a = 0$, i.e., the transition to the product, occurs equally well without a recommendation and brings high sales. Now let b be a product whose recommendation is strongly accepted but which is associated with a low reward r_b. The reality is $q^\pi(s, a) > q^\pi(s, b)$, but the simplification delivers the opposite:

$$q^\pi(s,a) - q^\pi(s,b) \approx \Delta p^a r_a - \Delta p^b r_b = 0 - \Delta p^b r_b < 0.$$

So the simplification $c(s, a) \approx 1$ conceals the risk of down-selling; therefore, we do not generally apply it in practice. If we ignore it, we obtain for our example

$$q^\pi(s,a) - q^\pi(s,b) = \underbrace{\left[p^a_{ss_a} - c(s,b)p_{ss_a}\right]}_{\Delta p^a(b)} r_a - \Delta p^b r_b = \Delta p^a(b) r_a - \Delta p^b r_b.$$

Since with increasing $\Delta p^b = p^b_{ss_b} - p_{ss_b}$, conversely, $c(s,b) = \frac{1-p^b_{ss_b}}{1-p_{ss_b}}$ decreases, this leads in turn to an increasing $\Delta p^a(b)$. The raising of the action value by the increased acceptance of b works against its decrease by the reduced action potential of a, which is reflected by the scaling factor c. The decision between recommendations a and b is dependent on which of the two effects predominates.

We have therefore established that if $p^a_{ss_a}$ and p_{ss_a} are both small, it is practical to work with $c(s, a) = 1$ (even if $p^a_{ss_a}$ is higher than p_{ss_a} by a multiple, or vice versa). Otherwise we must work with the exact scaling factor $c(s, a)$.

5.2.2 Special Cases

We investigate and interpret below important special cases of (5.6). For this we firstly write out in full (5.6) with (5.5):

$$q^{\pi}(s,a) = p^a_{ss_a} r_{ss_a} + \frac{1 - p^a_{ss_a}}{1 - p_{ss_a}} \sum_{s' \neq s_a} p_{ss'} r_{ss'}. \tag{5.8}$$

The following special cases arise:

1. $p^a_{ss_a} = p_{ss_a}$ (no effectiveness of the recommendation):

$$q^{\pi}(s,a) = p_{ss_a} r_{ss_a} + \sum_{s' \neq s_a} p_{ss'} r_{ss'} = \sum_{s'} p_{ss'} r_{ss'} = q^0(s),$$

   and all recommendations a lead to the same q^0 action value
2. $p^a_{ss_a} = 0$ (no acceptance of the recommendation):

$$q^{\pi}(s,a) = \frac{1}{1 - p_{ss_a}} \sum_{s' \neq s_a} p_{ss'} r_{ss'}.$$

   The interpretation is that the action value of the recommendation a corresponds to the weighted unconditional action value. The conditional action value disappears. The reward for the recommendation plays no role at all, since there never is a transition to the product s_a.
3. $p^a_{ss_a} = 1$ (total acceptance of the recommendation):

$$q^{\pi}(s,a) = r_{ss_a},$$

   which means that the recommendation a always obtains its full reward. The unconditional action value disappears.
4. $p_{ss_a} = 0$ (no acceptance of the "recommendation" in the control group):

$$q^{\pi}(s,a) = p^a_{ss_a} r_{ss_a} + \left(1 - p^a_{ss_a}\right) \sum_{s' \neq s_a} p_{ss'} r_{ss'}.$$

   The interpretation is the action value corresponds to the conditional action value of recommendation a plus the probability of nonacceptance of the recommendation times the unconditional action value for all the other states $s' \neq s_a$.
5. $p_{ss_a} = 1$ (total acceptance of the "recommendation" in the control group):

$$q^{\pi}(s,a) = p^a_{ss_a} r_{ss_a} + \left(1 - p^a_{ss_a}\right) \frac{1}{k} \sum_{s' \neq s_a} r_{ss'},$$

where k is the number of all the other states $s' \neq s_a$. This is the most complicated case. The above equation follows from the elementary limit value consideration with the equation $p_{ss'} = \frac{1-p_{ss_a}}{k} \quad \forall s' \neq s_a$:

$$\lim_{p_{ss_a}\to 1} \frac{1-p^a_{ss_a}}{1-p_{ss_a}} \sum_{s'\neq s_a} \frac{1-p_{ss_a}}{k} r_{ss'} = \lim_{p_{ss_a}\to 1} \frac{1-p^a_{ss_a}}{1-p_{ss_a}} \frac{1-p_{ss_a}}{k} \sum_{s'\neq s_a} r_{ss'} = \left(1-p^a_{ss_a}\right) \frac{1}{k} \sum_{s'\neq s_a} r_{ss'}.$$

Of course, the distribution of the unconditional probabilities $p_{ss'}$ may also be modeled differently (which may lead to a different result), but this is the most natural approach. The interpretation is since in the recommendation-free case, the transition always leads to s_a, the distribution of the $p_{ss'}$ is unknown if, in the recommendation case, the transition leads to $s' \neq s_a$. Therefore, the $p_{ss'}$ are assumed to be equal. Hence, the action value corresponds to the conditional action value of recommendation a plus the probability of nonacceptance of the recommendation times the average reward over the other states $s' \neq s_a$.

The approach described for estimation of the transition probabilities in (3.5) can now be applied similarly for a positive discount rate γ (3.6), so that in the result we again obtain an equation similar to (5.6), albeit of course more complex:

$$\begin{aligned} q^\pi(s,a) = p^a_{ss_a}\left[p^a_{ss_a} + \gamma \sum_{a'} \pi(s_a,a') q^\pi(s_a,a')\right] \\ + c(s,a) \sum_{s'\neq s_a} p^a_{ss'} \left[r^a_{ss'} + \gamma \sum_{a'} \pi(s',a') q^\pi(s',a')\right]. \end{aligned} \quad (5.9)$$

This is then solved in realtime once again using ADP. Concerning a corresponding TD algorithm, it is not easy to derive because of the nonlinearity of (5.8) with respect to p_{ss_a}. We leave this as an open problem. Nevertheless, we will also consider the TD-Version in the course of this book, especially in Chaps. 6 and 10.

5.2.3 Estimation of Transition Probabilities

The conditional transition probabilities $p^a_{ss_a}$ may be computed from the transactions according to Algorithm 4.1 or, in case that multiple recommendations have been issued, Algorithm 4.2.

Computation of the unconditional transition probabilities $p_{ss'}$ has not yet been addressed. It may be calculated either from sessions in the control group (the topic

of measuring success and control groups is addressed in Chap. 11). Since no recommendations are issued here, Algorithm 4.1 may straightforwardly be used to determine the $p_{ss'}$.

Control groups, however, are not always available. Furthermore, the approach is prone to inconsistencies. Hence, we shall jointly estimate both the conditional as well as the unconditional probabilities invoking our central Assumption 5.2.

5.2.3.1 One Recommendation

We shall first consider the case of one recommendation a issued in state s and introduce the following notation for the internally used probabilities:

$$p_{ss'}^{[a]} = \begin{cases} p_{ss'}, & s' \neq s_a \\ p_{ss'}^a, & s' = s_a \end{cases}.$$

For a fix p_{ss_a} Assumption 5.2 with (5.3) included stipulates the following 1–1 mapping between our internal probabilities and the conditional ones:

$$p_{ss'}^a = F_{p_{ss_a}}\left(p_{ss'}^{[a]}\right) = \begin{cases} c(s,a)p_{ss'}, & s' \neq s_a \\ p_{ss'}^a, & s' = s_a \end{cases} = \begin{cases} \dfrac{1-p_{ss_a}^a}{1-p_{ss_a}} p_{ss'}, & s' \neq s_a \\ p_{ss'}^a, & s' = s_a \end{cases}. \tag{5.10}$$

(The special cases $p_{ss_a}^a = 1$ and $p_{ss_a} = 1$ will not be addressed here, since they are cumbersome to deal with and do not figure decisively in the basic approach.)

Introducing the vectors $\mathbf{p}^{[a]} = (p_{ss'}^{[a]})_{s'}$, $\mathbf{p}^a = (p_{ss'}^a)_{s'}$ and letting m be the number of successor states s', (5.10) defines the vector function $\mathbf{F}_{p_{ssa}} = \left(F_{p_{ssa}}^1 \ldots F_{p_{ssa}}^m\right)^T$:

$$\mathbf{p}^a = \mathbf{F}_{p_{ssa}}\left(\mathbf{p}^{[a]}\right). \tag{5.11}$$

Since in practice, however, we can update only $\mathbf{p}^a$, we also need the inverse

$$\mathbf{p}^{[a]} = \mathbf{F}_{p_{ssa}}^{-1}(\mathbf{p}^a). \tag{5.12}$$

which is equally easy to compute. Equipped with this mapping, we proceed in essentially the same fashion as Algorithm 4.1 from Sect. 4.2.

We store only the internal probabilities ${}^j\mathbf{p}^{[a]}$, where j is again the update step, in our rule base. After the recommendation a has been issued, we first compute the expected conditional probabilities ${}^j\mathbf{p}^a = \mathbf{F}_{{}^jp_{ssa}}\left({}^j\mathbf{p}^{[a]}\right)$. The unconditional probability of the issued recommendation is kept fix, since we cannot update it anyway, as the recommendation has been issued! The conditional probabilities ${}^j\mathbf{p}^a$ are now computed by virtue of Algorithm 4.1 according to accepted or rejected recommendation,

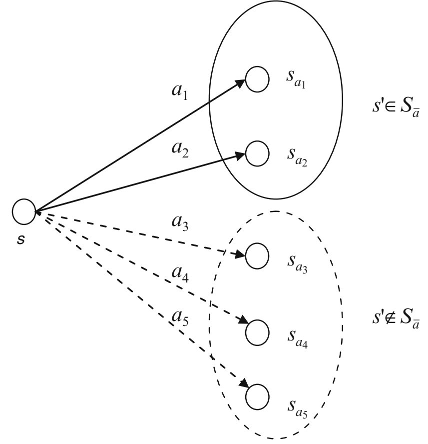


Fig. 5.2 Example of a product with two recommendations $\overline{a} = (a_1, a_2)$ and a total of five following products

and we obtain the updated conditional probabilities ${}^{j+1}\mathbf{p}^{a}$. To the latter, we apply the inverse mapping ${}^{j+1}\mathbf{p}^{[a]} = \mathbf{F}^{-1}_{{}^{j}p_{ss_a}}\left({}^{j+1}\mathbf{p}^{a}\right)$ and obtain the current internal probabilities ${}^{j+1}\mathbf{p}^{[a]}$. Subsequently, we carry over ${}^{j}p_{ss_a}$ unchanged to the next update.

Algorithm 5.1: Update of the internal from conditional probabilities for one recommendation

---

Input: vector of internal probabilities ${}^{j}\mathbf{p}^{[a]}$ and fixed probability ${}^{j}p_{ss_a}$, delivered recommendation a, index of product transition l, step size α_j
Output: updated vector of internal probabilities ${}^{j+1}\mathbf{p}^{[a]}$ and ${}^{j+1}p_{ss_a}$

1: **procedure** UPDATE_P_DP_SINGLE(${}^{j}\mathbf{p}^{[a]}, {}^{j}p_{ss_a}, a, l, \alpha_j$)
2: ${}^{j}\mathbf{p}^{a} := \mathbf{F}^{j}p_{ss_a}({}^{j}\mathbf{p}^{[a]})$ ▷ conversion into conditional probabilities
3: ${}^{j+1}\mathbf{p}^{a} :=$ UPDATE_P_SINGLE(${}^{j}\mathbf{p}^{a}$, l, α_j) ▷ update of conditional probabilities
4: ${}^{j+1}\mathbf{p}^{[a]} := \mathbf{F}^{j}p_{ss_a} - 1\left({}^{j+1}\mathbf{p}\right)$ ▷ conversion into internal probabilities
5: ${}^{j+1}p_{ss_a} := {}^{j}p_{ss_a}$ ▷ unchanged take-over of the fixed component
6: **return** (${}^{j+1}\mathbf{p}^{[a]}$, ${}^{j+1}p_{ss_a}$)
7: **end procedure**

5.2.3.2 Multiple Recommendations

We shall now attend to the case of multiple recommendations. Let again $\overline{a} = (a_1, \ldots, a_k)$ be the k issued recommendations and $S_{\overline{a}}$ be the set of states corresponding to the former; see Fig. 5.2. Let further $S^{C}_{\overline{a}} = S_{A(s)}/S_{\overline{a}}$ be the complementary set of all not-recommended successor states, i.e., $s' \notin S_{\overline{a}}$. We denote the fixed

set of unconditional probabilities assigned to the recommendations as $\Pi_{\overline{a}} = \{p_{ss'}\}_{s' \in S_{\overline{a}}}$.

Firstly, we again introduce our internal probabilities:

$$p_{ss'}^{[\overline{a}]} = \begin{cases} p_{ss'}, & s' \notin S_{\overline{a}} \\ p_{ss'}^{a_{s'}}, & s' \in S_{\overline{a}} \end{cases},$$

i.e., $p_{ss'}^{[\overline{a}]}$ corresponds to $p_{ss'}^{[a]}$, where the grouping is now over a multitude of issued recommendations. For the special case of one recommendation $S_{\overline{a}} = \{s_a\}$, the two coincide.

The question, which also figures in the simulations described in Sect. 5.4, is: How can we calculate the transition probabilities of multiple recommendations $p_{ss'}^{\overline{a}}$ from those of single recommendations $p_{ss_a}^{a}$ and $p_{ss'}$?

According to Sect. 4.2, we may calculate $p_{ss'}^{\overline{a}}$ either following the linear approach (4.3). Alternatively, we may resort to the nonlinear approach (4.5).

Linear Approach

In conjunction with the linear approach (4.3), we generalize (5.10) and again obtain the 1–1 mapping between the internal and conditional probabilities:

$$p_{ss'}^{\overline{a}} = F_{\Pi_{\overline{a}}}\left(p_{ss'}^{[\overline{a}]}\right) = \frac{1}{k}\sum_{i=1}^{k} F_{s_{a_i}}\left(p_{ss'}^{[a_i]}\right) = \begin{cases} \frac{1}{k}\sum_{i=1}^{k} c(s,a_i)p_{ss'}, & s' \notin S_{\overline{a}} \\ \frac{1}{k}\left(p_{ss'}^{a_{s'}} + \sum_{i=1,\,a_i \neq a_{s'}}^{k} c(s,a_i)p_{ss'}\right), & s' \in S_{\overline{a}} \end{cases}$$

$$= \begin{cases} \frac{1}{k}\sum_{i=1}^{k} \frac{1 - p_{ss_{a_i}}^{a_i}}{1 - p_{ss_{a_i}}} p_{ss'}, & s' \notin S_{\overline{a}} \\ \frac{1}{k}\left(p_{ss'}^{a_{s'}} + \sum_{i=1,\,a_i \neq a_{s'}}^{k} \frac{1 - p_{ss_{a_i}}^{a_i}}{1 - p_{ss_{a_i}}} p_{ss'}\right), & s' \in S_{\overline{a}} \end{cases}. \tag{5.13}$$

As for the one-recommendation case, we change over to the vectors $\mathbf{p}^{[\overline{a}]} = \left(p_{ss'}^{[\overline{a}]}\right)_{s'}$ and $\mathbf{p}^{\overline{a}} = \left(p_{ss'}^{\overline{a}}\right)_{s'}$ and introduce the vector function $\mathbf{F}_{\Pi_{\overline{a}}} = \left(F_{\Pi_{\overline{a}}}^{1} \ldots F_{\Pi_{\overline{a}}}^{m}\right)^{T}$ through

$$\mathbf{p}^{\overline{a}} = \mathbf{F}_{\Pi_{\overline{a}}}\left(\mathbf{p}^{[\overline{a}]}\right) \tag{5.14}$$

and its inverse through

$$\mathbf{p}^{[\overline{a}]} = \mathbf{F}_{\Pi_{\overline{a}}}^{-1}\left(\mathbf{p}^{\overline{a}}\right). \tag{5.15}$$

The inverse mapping requires the solution of an equation system which is not difficult to compute. Although (5.14) is formally nonlinear with respect to $\mathbf{p}^{[\overline{a}]}$, in reality the solution can be done in a quite similar way. In fact, first we consider all recommended successor products $s' \in S_{\overline{a}}$ and calculate their corresponding conditional probabilities $p_{ss'}^{a_{s'}}$. This requires the solution of a linear equation system with $|S_{\overline{a}}|$ unknowns. Then we turn to the remaining, not-recommended successor states $s' \notin S_{\overline{a}}$ and directly calculate their unconditional probabilities $p_{ss'}$.

That way, we end up at Algorithm 5.2 for multiple recommendations which is quite similar to Algorithm 5.1 of single recommendations.

Algorithm 5.2: Update of the internal from conditional probabilities for multiple recommendations, linear mapping

---

Input: vector of internal probabilities ${}^{j}\mathbf{p}^{[\overline{a}]}$ and fixed probabilities ${}^{j}\Pi_{\overline{a}}$, delivered recommendations $\overline{a} = (a_1, \ldots, a_k)$, index of product transition l, step size α_j

Output: updated vector of internal probabilities ${}^{j+1}\mathbf{p}^{[\overline{a}]}$ and ${}^{j+1}\Pi_{\overline{a}}$

1: **procedure** UPDATE_P_DP_MULTI_LIN(${}^{j}\mathbf{p}^{[\overline{a}]}, {}^{j}\Pi_{\overline{a}}, \overline{a}, l, \alpha_j$)

2: ${}^{j}\mathbf{p}^{\overline{a}} = \mathbf{F}_{{}^{j}\Pi_{\overline{a}}}\left({}^{j}\mathbf{p}^{[\overline{a}]}\right)$ ▷ conversion into conditional probabilities

3: ${}^{j+1}\mathbf{p}^{\overline{a}}$:= UPDATE_P_SINGLE(${}^{j}\mathbf{p}^{\overline{a}}, l, \alpha_j$) ▷ update of conditional probabilities

4: ${}^{j+1}\mathbf{p}^{[\overline{a}]} := \mathbf{F}_{{}^{j}\Pi_{\overline{a}}}^{-1}\left({}^{j+1}\mathbf{p}^{\overline{a}}\right)$ ▷ conversion into internal probabilities

5: ${}^{j+1}\Pi_{\overline{a}} := {}^{j}\Pi_{\overline{a}}$ ▷ unchanged take-over of the fixed component

6: **return** (${}^{j+1}\mathbf{p}^{[\overline{a}]}, {}^{j+1}\Pi_{\overline{a}}$)

7: **end procedure**

We now turn to the action-value function. For complexity reasons we cannot check all combinations of single recommendations in order to determine the greedy policy. We need a more efficient approach.

At this, we plug in (4.3) into (3.5) and rewrite the action-value function for multiple recommendations:

$$q^{\pi}(s,\overline{a}) = \sum_{s'} p_{ss'}^{\overline{a}} r_{ss'} = \sum_{s'} \frac{1}{k}\left(\sum_{i=1}^{k} p_{ss'}^{a_i}\right) r_{ss'} = \frac{1}{k}\sum_{i=1}^{k}\sum_{s'} p_{ss'}^{a_i} r_{ss'} = \frac{1}{k}\sum_{i=1}^{k} q^{\pi}(s,a_i).$$

This nice result tells us that in order to find the highest action value of k recommendations, we just need to select the k recommendations of the highest single action values. The same holds for the full action-value function (3.6).

Nonlinear Approach

We remind the reader, though, that according to (4.5), $p_{ss'}^{\overline{a}}$ can only be applied to $s' \in S_{\overline{a}}$, i.e., to the successor states corresponding to the recommendations. Yet we need $p_{ss'}^{\overline{a}}$ for all admissible subsequent states s'.

To this end, we extend Assumption 5.2 to all of the k issued recommendations and generalize (5.3) to

$$p_{ss'}^{\overline{a}} = c(s, \overline{a}) p_{ss'}, \quad s' \notin S_{\overline{a}}. \tag{5.16}$$

While being able to calculate the transition probabilities to each of the issued recommended states $p_{ss'}^{\overline{a}}$, $s' \in S_{\overline{a}}$ according to (4.4), we use (5.16) for the remaining successor states. The scaling factor $c(s, \overline{a})$ is determined similarly to $c(s, a)$ as described in Sect. 5.2:

$$\sum_{s' \in S_{\overline{a}}} p_{ss'}^{\overline{a}} + c(s, \overline{a}) \sum_{s' \notin S_{\overline{a}}} p_{ss'} = 1$$

together with $\sum_{s'} p_{ss'} = 1$ yields

$$c(s, \overline{a}) = \frac{1 - \sum_{s' \in S_{\overline{a}}} p_{ss'}^{\overline{a}}}{1 - \sum_{s' \in S_{\overline{a}}} p_{ss'}}. \tag{5.17}$$

We are now in a position to generalize Assumption 5.2 to the case of a given multiple recommendation $\overline{a}$ and to compute its corresponding transition probabilities $p_{ss'}^{\overline{a}}$ from the single recommendation probabilities $p_{ss'}^{(a)}$.

Thus, we have gathered all of the tools necessary to estimate the transition probabilities from multiple recommendations.

Using the transformation $F_{s'}$ from (4.5), we may calculate the joint probabilities $p_{ss'}^{\overline{a}}$, $s' \in S_{\overline{a}}$ from the single probabilities of the issued recommendations $p_{ss'}^{a_{s'}}, s' \in S_{\overline{a}}$ (equivalent notation: $p_{ss_{a_l}}^{a_l}$, $l = 1, \ldots, k$) and thereupon define the intermediate probabilities $p_{ss'}^{\{\overline{a}\}}$:

$$p_{ss'}^{\{\overline{a}\}} = \begin{cases} p_{ss'}, & s' \notin S_{\overline{a}} \\ p_{ss'}^{\overline{a}}, & s' \in S_{\overline{a}} \end{cases} = \begin{cases} p_{ss'}, & s' \notin S_{\overline{a}} \\ F_{s'}\left(p_{ss_{a_1}}^{a_1}, \ldots, p_{ss_{a_k}}^{a_k}\right), & s' \in S_{\overline{a}} \end{cases}, \tag{5.18}$$

which enables to introduce the vector function $\mathbf{G}_{S_{\overline{a}}}$ in terms of the vectors $\mathbf{p}^{[\overline{a}]} = \left(p_{ss'}^{[\overline{a}]}\right)_{s'}$ and $\mathbf{p}^{\{\overline{a}\}} = \left(p_{ss'}^{\{\overline{a}\}}\right)_{s'}$:

$$\mathbf{p}^{\{\overline{a}\}} = \mathbf{G}_{S_{\overline{a}}}\left(\mathbf{p}^{[\overline{a}]}\right) \tag{5.19}$$

and its inverse

$$\mathbf{p}^{[\overline{a}]} = \mathbf{G}_{S_{\overline{a}}}^{-1}\left(\mathbf{p}^{\{\overline{a}\}}\right).$$

The inverse $\mathbf{G}_{S_{\overline{a}}}^{-1}$ is not quite easy to compute, since it involves the inversion of the vector function for multiple recommendations $\mathbf{F}^{-1}$ described in Sect. 4.2, though along with a suitable approach thereto.

For the fixed set of unconditional probabilities assigned to the recommendations $\Pi_{\overline{a}}$, we again obtain the following 1–1 correspondence $F_{\Pi_{\overline{a}}}$ between the intermediate and conditional probabilities:

$$p_{ss'}^{\overline{a}} = F_{\Pi_{\overline{a}}}\left(p_{ss'}^{\{\overline{a}\}}\right) = \begin{cases} c(s,\overline{a})p_{ss'}, & s' \notin S_{\overline{a}} \\ p_{ss'}^{\overline{a}}, & s' \in S_{\overline{a}} \end{cases}. \tag{5.20}$$

Here, $c(s,\overline{a})$ is defined in compliance with (5.16) and calculated through (5.17). Again, we change over to the vectors $\mathbf{p}^{\{\overline{a}\}} = \left(p_{ss'}^{\{\overline{a}\}}\right)_{s'}$ and $\mathbf{p}^{\overline{a}} = \left(p_{ss'}^{\overline{a}}\right)_{s'}$ and introduce the vector function $\mathbf{F}_{\Pi_{\overline{a}}} = \left(F_{\Pi_{\overline{a}}}^{1} \ldots F_{\Pi_{\overline{a}}}^{m}\right)^{T}$ through

$$\mathbf{p}^{\overline{a}} = \mathbf{F}_{\Pi_{\overline{a}}}\left(\mathbf{p}^{\{\overline{a}\}}\right) \tag{5.21}$$

and its inverse through

$$\mathbf{p}^{\{\overline{a}\}} = \mathbf{F}_{\Pi_{\overline{a}}}^{-1}\left(\mathbf{p}^{\overline{a}}\right).$$

Thus, we are in a position to state Algorithm 5.3 for the calculation of the transition probabilities from multiple recommendations similarly to Algorithm 5.1.

Again, we start with the internal probabilities ${}^{j}\mathbf{p}^{[\overline{a}]}$ in the jth update step. After the multiple recommendation $\overline{a}$ has been issued, we first need to compute the predicted multiple probabilities ${}^{j}p_{ss'}^{\overline{a}},\ s'{\in}S_{\overline{a}}$ from the single probabilities ${}^{j}p_{ss'}^{a_{s'}},\ s'{\in}S_{\overline{a}}$ of the recommendations. This is the core of the operation ${}^{j}\mathbf{p}^{\{\overline{a}\}} = {}^{j}\mathbf{G}_{S_{\overline{a}}}\left(\mathbf{p}^{[\overline{a}]}\right)$ which produces the conditional probabilities. Now we have to convert the unconditional probabilities into the conditional ones, which is in turn the core of the operation ${}^{j}\mathbf{p}^{\overline{a}} = \mathbf{F}_{\Pi_{\overline{a}}}\left({}^{j}\mathbf{p}^{\{\overline{a}\}}\right)$. Here, we keep the unconditional probability of the issued recommendations ${}^{j}\Pi_{\overline{a}} = \left\{{}^{j}p_{ss'}\right\}_{s'\in S_{\overline{a}}}$ fixed.

Thus, we have completed the conversion into the conditional probabilities ${}^{j}\mathbf{p}^{a}$ which we again update with respect to the accepted and rejected recommendations according to Algorithm 4.1 to obtain the updated conditional probabilities ${}^{j+1}\mathbf{p}^{a}$. Using the inverse mappings $\mathbf{F}^{-1}_{{}^{j}\Pi_{\overline{a}}}$ and $\mathbf{G}^{-1}_{S_{\overline{a}}}$, we reconvert them into our internal single probabilities ${}^{j+1}\mathbf{p}^{[a]}$. We then carry the unconditional probabilities of the recommendations ${}^{j+1}\Pi_{\overline{a}}$ over unchanged to the next update step.

Algorithm 5.3: Update of the internal from conditional probabilities for multiple recommendations

---

Input: vector of internal probabilities ${}^{j}\mathbf{p}^{[\overline{a}]}$ and fixed probabilities ${}^{j}\Pi_{\overline{a}}$, delivered recommendations $\overline{a} = (a_1, \ldots, a_k)$, index of product transition l, step size α_j
Output: updated vector of internal probabilities ${}^{j+1}\mathbf{p}^{[\overline{a}]}$ and ${}^{j+1}\Pi_{\overline{a}}$
1: **procedure** UPDATE_P_DP_MULTI(${}^{j}\mathbf{p}^{[\overline{a}]}, {}^{j}\Pi_{\overline{a}}, \overline{a}, l, \alpha_j$)
2: ${}^{j}\mathbf{p}^{\{\overline{a}\}} = \mathbf{G}_{S_{\overline{a}}}\left({}^{j}\mathbf{p}^{[\overline{a}]}\right)$ ▷ conversion into intermediate probabilities
3: ${}^{j}\mathbf{p}^{\overline{a}} = \mathbf{F}_{{}^{j}\Pi_{\overline{a}}}\left({}^{j}\mathbf{p}^{\{\overline{a}\}}\right)$ ▷ conversion into conditional probabilities
4: ${}^{j+1}\mathbf{p}^{\overline{a}} :=$ UPDATE_P_SINGLE(${}^{j}\mathbf{p}^{\overline{a}}, l, \alpha_j$) ▷ update of conditional probabilities
5: ${}^{j+1}\mathbf{p}^{\{\overline{a}\}} = \mathbf{F}^{-1}_{{}^{j}\Pi_{\overline{a}}}\left({}^{j+1}\mathbf{p}^{\overline{a}}\right)$ ▷ conversion into intermediate probabilities
6: ${}^{j+1}\mathbf{p}^{[\overline{a}]} = \mathbf{G}^{-1}_{S_{\overline{a}}}\left({}^{j+1}\mathbf{p}^{\{\overline{a}\}}\right)$ ▷ conversion into internal probabilities
7: ${}^{j+1}\Pi_{\overline{a}} := {}^{j}\Pi_{\overline{a}}$ ▷ unchanged take-over of the fixed component
8: **return** (${}^{j+1}\mathbf{p}^{[\overline{a}]}, {}^{j+1}\Pi_{\overline{a}}$)
9: **end procedure**

A closer look at Algorithm 5.3 reveals that it may be arranged in a different way by updating the conditional recommendation probabilities in a bundle by means of Algorithm 4.2 and updating the unconditional (non-fixed) recommendations separately.

Indeed, since the unconditional probabilities of the recommended products $s' \in S_{\overline{a}}$ are kept fix, i.e., ${}^{j+1}\Pi_{\overline{a}} = {}^{j}\Pi_{\overline{a}}$, also their sum $\sum_{s' \in S_{\overline{a}}} p_{ss'}$ does not change, and due to $\sum_{s' \notin S_{\overline{a}}} p_{ss'} = 1 - \sum_{s' \in S_{\overline{a}}} p_{ss'}$ also the sum of all unconditional probabilities of the non-recommended products is constant. Thus, if one of the recommendations is accepted, all unconditional probabilities remain unchanged. Only if no recommendation is accepted, the unconditional probabilities of the non-recommended products will change (but not their sum).

In order to formulate the algorithm, let us denote all parts of vectors corresponding to the recommended products by index c and to the non-recommended products by index u. Especially, we denote $\mathbf{p}_c^{[\overline{a}]} = \left(p_{ss'}^{[\overline{a}]}\right)_{s' \in S_{\overline{a}}}$

$= \left(p_{ss'}^{a_{s'}}\right)_{s' \in S_{\overline{a}}}$ and $\mathbf{p}_u^{[\overline{a}]} = \left(p_{ss'}^{[\overline{a}]}\right)_{s' \notin S_{\overline{a}}} = (p_{ss'})_{s' \notin S_{\overline{a}}}$. Similar from the transition index l, we derive the indexes $l_c = \begin{cases} l, & s_l \in S_{\overline{a}} \\ -1, & s_l \notin S_{\overline{a}} \end{cases}$ and $l_u = \begin{cases} l, & s_l \notin S_{\overline{a}} \\ -1, & s_l \in S_{\overline{a}} \end{cases}$.

Then we can reformulate Algorithm 5.3 in a more compact way (and computational cheaper) as Algorithm 5.4.

Thus, we first update the conditional probabilities of the recommended products. Only in case that no recommendation has been accepted, the unconditional probabilities of all non-recommended products are updated. Strictly speaking Step 4 requires additional scaling since the update of the unconditional probabilities of the non-recommended products by Algorithm 4.1 may change their sum $\sum_{s' \notin S_{\overline{a}}} p_{ss'}$.

However, this is an easy task. We just need to store the sum before and after the update and then to multiply all updated probabilities by the corresponding factor.

Algorithm 5.4: Update of the internal from conditional probabilities for multiple recommendations, Version 2

Input: vector of internal probabilities ${}^{j}\mathbf{p}^{[\overline{a}]}$ and fixed probabilities ${}^{j}\Pi_{\overline{a}}$, delivered recommendations $\overline{a} = (a_1, \ldots, a_k)$, index of product transition l, step size α_j
Output: updated vector of internal probabilities ${}^{j+1}\mathbf{p}^{[\overline{a}]}$ and ${}^{j+1}\Pi_{\overline{a}}$

1: **procedure** UPDATE_P_DP_MULTI2(${}^{j}\mathbf{p}^{[\overline{a}]}, {}^{j}\Pi_{\overline{a}}, \overline{a}, l, \alpha_j$)
2 ${}^{j+1}\mathbf{p}_c^{[\overline{a}]}$:= UPDATE_P_MULTI
($ {}^{j}\mathbf{p}_c^{[\overline{a}]}, \overline{a}, l_c, \alpha_j$) ▷ update of recommendation probabilities
3: **if** $l_c = -1$ **then**
4: ${}^{j+1}\mathbf{p}_u^{[\overline{a}]}$:= UPDATE_P_SINGLE
($ {}^{j}\mathbf{p}_u^{[\overline{a}]}, l_u, \alpha_j$) ▷ update of non-recomm. probabilities
5: **else**
6: ${}^{j+1}\mathbf{p}_u^{[\overline{a}]} := {}^{j}\mathbf{p}_u^{[\overline{a}]}$ ▷ unchanged take-over of non-recomm. probabilities
7: **end if**
8: ${}^{j+1}\Pi_{\overline{a}} := {}^{j}\Pi_{\overline{a}}$ ▷ unchanged take-over of the fixed component
9: **return** (${}^{j+1}\mathbf{p}^{[\overline{a}]}, {}^{j+1}\Pi_{\overline{a}}$)
10: **end procedure**

Although Algorithms 5.3 and 5.4 deliver in principal the same result, they represent different perspectives on the update of conditional and unconditional probabilities. While Algorithm 5.3 updates both probability types by one algorithm, Algorithm 5.4 separates this calculation and allows calculating conditional and unconditional probabilities by different update algorithms. Even in case of the same update rule (3.8), as used in Algorithm 5.4, this may result in different counters for the coefficients α_j and lead to different results. We will not further deepen this special topic here.

We conclude this section with the formulation of the action-value function for multiple recommendations. From (5.16) it follows that (5.6) takes the form

$$q^{\pi}(s,\overline{a}) = \sum_{s' \in S_{\overline{a}}} p^{\overline{a}}_{ss'} r_{ss'} + c(s,\overline{a}) \sum_{s' \neq s_{\overline{a}}} p_{ss'} r_{ss'}.$$

Due to the nonlinearity of $c(s,\overline{a})$, this means that we would need to consider all possible combinations of recommendations $\overline{a}$ in order to find the highest action value. This would result into an enormous complexity that cannot be handled for real-life problems. For the moment, we use the simplest approach by selecting the k best single recommendations, like in the linear case. This approach, however, only for the simplified DP-Version (5.26) works exactly. (The simplified DP-Version will be introduced in the next section.)

So although we are able to correctly determine all transition probabilities for multiple recommendations and to use them accurately in the simulations (Sect. 5.4), we still lack a computationally efficient approach to calculate the best multiple recommendations. Of course, this should be a subject for future studies. Special optimization techniques shall be able to solve this problem.

Comparison of Linear and Nonlinear Approaches

Summing up, the linear approach seems to be favorable to the nonlinear one because it is easier to implement and does not leave any principal problems open.

5.3 Combination of Conditional and Unconditional Approaches

In this section, we want to present a first approach how to deal with uncertain data.

So far, we have always assumed that all estimated probabilities $p^{(a)}_{ss'}$, i.e., $p^{a}_{ss'}$ and $p_{ss'}$, are equally reliable. However, in most applications for a state s, new transitions $s \rightarrow s'$ (usually represented as rules) are dynamically added during the process of learning. For example, if s is a long-standing product of a web shop and recently a new product s'' was included into the assortment of the shop, then we may dynamically add the transition probabilities $p^{(a)}_{ss''}$ to the existing ones.

Let ${}^{j}p^{(a)}_{ss'}$ represent the estimated probabilities $p^{(a)}_{ss'}$ after j update steps. Then the described dynamic approach means that different target states s' may have different counter values j. Obviously, for a state s_1 with a large counter j_1 in general, the corresponding transition probabilities ${}^{j1}p^{(a)}_{ss_1}$ can be considered as more reliable than ${}^{j2}p^{(a)}_{ss_2}$ for a state s_2 with a small counter j_2.

In order to calculate (5.8) meaningfully, the transition probabilities $p^{(a)}_{ss'}$ must be statistically stable, at least to some extent. In other words, adding transition

probabilities $p_{ss''}^{(a)}$ of a very new product s'' may deteriorate our whole approach (5.8). To overcome this problem we will present some first ideas here.

At this, we replace ${}^{n}p_{ss'}^{(a)}$ with the "stabilized" probabilities

$$ {}^{n}\widetilde{p}_{ss'}^{(a)} := \begin{cases} {}^{n}p_{ss'}^{(a)}, & \text{if } n \geq n_{\min} \\ 0, & \text{if } n < n_{\min} \end{cases}, \tag{5.22} $$

where $n_{\min}$ is a threshold value for the minimum statistical mass (usually 20 or more) and instead of (5.8) now calculate

$$ \widetilde{q}^{\pi}(s,a) = \widetilde{p}_{ss_a}^{a} r_{ss_a} + \frac{1-\widetilde{p}_{ss_a}^{a}}{1-\widetilde{p}_{ss_a}} \sum_{s' \neq s_a} \widetilde{p}_{ss'} r_{ss'}. \tag{5.23} $$

There remains a problem at (5.23) however because of the fact that for new transitions the conditional action value $\widetilde{p}_{ss_a}^{a} r_{ss_a}$ initially is 0 or small. That means that its recommendations are scarcely delivered and $\widetilde{p}_{ss_a}^{a} r_{ss_a}$ can scarcely grow (unless $\widetilde{q}^{\pi}(s,a)$ increases via its unconditional action value). We then have a vicious circle.

In order to escape this, we modify (5.22) for the conditional probabilities $p_{ss'}^{a}$ as follows:

$$ {}^{n}\hat{p}_{ss'}^{a} := \begin{cases} {}^{n}p_{ss'}^{a}, & \text{if } n \geq n_{\min} \\ s_C {}^{m}p_{ss'}, & \text{if } n < n_{\min} \end{cases}, \tag{5.24} $$

where $s_C \in [1, \infty)$ is a fixed scaling factor. Here the n refers to the counter of the conditional probability (i.e., for delivery of a), whereas on the other hand m is the counter for the unconditional probability! In this way we replace (5.23) with the final estimation

$$ \hat{q}^{\pi}(s,a) = \hat{p}_{ss_a}^{a} r_{ss_a} + \frac{1-\hat{p}_{ss_a}^{a}}{1-\widetilde{p}_{ss_a}} \sum_{s' \neq s_a} \widetilde{p}_{ss'} r_{ss'}. \tag{5.25} $$

We now come to the interpretation of (5.25). Since the unconditional probabilities $p_{ss'}$ are continually updated, even without delivery of s', these have real chances of being delivered as recommendations, and the conditional probability counter increases. As soon as it reaches the threshold $n_{\min}$, the initial auxiliary probability $\hat{p}_{ss_a}^{a}$ is replaced by the conditional probability $p_{ss'}^{a}$.

The scaling factor s_C should be motivated in the broader sense. If $s_C = 1$ is set, there is the risk that the new recommendations are often not sufficiently strong to be shown. (Generally $p_{ss_a}^{a} > p_{ss_a}$ is the case, i.e., the probability of the transition to a product s_a is generally higher if it is also recommended.) It follows from this that in general $s_C > 1$ should be selected, so that the transition probabilities p_{ss_a} have a real chance of being delivered.

The selection $s_C > 1$ is also useful in respect of the delivery of competing initial recommendations. For this we initially assume $s_C = 1$. As long as ${}^n\hat{p}^{\,a}_{ss'}$ is in the initialization phase, i.e., $n < n_{\min}$, under the mostly valid (and not crucial) assumption $\widetilde{p}_{ss'} = p_{ss'}, \ \forall s' \neq s_a$ (i.e., the other unconditional probabilities are stable), (5.25) takes the form

$$\hat{q}^{\pi}(s,a) = p_{ss_a} r_{ss_a} + 1 \sum_{s' \neq s_a} p_{ss'} r_{ss'} = \sum_{s'} p_{ss'} r_{ss'} = q^0(s,a),$$

and thus $\hat{q}^{\pi}(s,a)$ is the same for all recommendations in the initial phase. On the other hand, the introduction of the scaling factor $s_C > 1$ yields the desired behavior:

$$\hat{q}^{\pi}(s,a) > \hat{q}^{\pi}(s,b) \quad \Leftrightarrow p_{ss_a} r_{ss_a} > p_{ss_b} r_{ss_b}.$$

Thus the method for the initial recommendations works similarly to that of the P-Version. For methodological purposes we therefore introduce a simplified version of (5.25):

$$\hat{q}_s^{\,\pi}(s,a) = \hat{p}^{\,a}_{ss_a} r_{ss_a}. \tag{5.26}$$

This therefore combines the P-Version for unconditional and conditional probabilities. As long as $n < n_{\min}$, it corresponds largely to the P-Version for the corresponding recommendation, i.e.,

$$\hat{q}_s^{\,\pi}(s,a) = s_C p_{ss_a} r_{ss_a},$$

and for $s_C = 1$ it is actually identical:

$$\hat{q}_s^{\,\pi}(s,a) = p_{ss_a} r_{ss_a} = q^P(s,a).$$

As soon as the threshold value $n_{\min}$ is reached, it changes into a P-Version operating on the basis of conditional probability:

$$\hat{q}_s^{\,\pi}(s,a) = p^a_{ss_a} r_{ss_a}.$$

The transition from unconditional to conditional probabilities in (5.25) or (5.26) makes sense in terms of content too: as long as the statistical mass is small, one should not operate with the complex conditional probabilities. Therefore, the unconditional probabilities are used, whose stability increases more quickly – and without requiring the delivery of recommendations. If then the necessary statistical mass is reached, we change over to the qualitatively more demanding conditional probabilities. In this way we achieve a continuous transition from the P- to the DP-Version.

The question of the best value of s_C is difficult and a subject of forthcoming investigations and will not be addressed further here.

In closing let us turn our attention to a further special problem in the conditional version. If a rule is no longer applied for recommendations after exceeding the threshold value $n_{\min}$ (because in the meantime other rules have become preferred), it has – at least in the simplified version (5.26) – in general little chance to be applied again, since the conditional probability $p_{ss'}^a$ is no longer being updated. This holds even if its potential acceptance has increased again.

In order to get around this, we introduce a special explorative delivery mode for the DP algorithm. For this, similarly to the ε -greedy policy, a percentage rate ε_{DP} is specified, in which instead of being delivered according to the action-value function $\hat{q}^{\pi}(s,a)$, the recommendations are delivered in descending order according to the following criterion:

$$\Theta(s,a) = \left(p_{ss_a} - p_{ss_a}^a\right) r_{ss_a} = -\Delta p^a r_a. \tag{5.27}$$

Thus, the idea is that the difference between the unconditional probability p_{ss_a} and the conditional probability $p_{ss_a}^a$ is a good indicator for whether a rule has become more attractive again. For if the difference increases, the user will be more inclined toward product s_a even without a recommendation, and the necessity of its delivery increases.

Let us emphasize that the empirical approach of this section just presents some very first and simple approaches to handle the crucial problem of statistical stability of the DP-Version. Surely, much more advanced instruments can be developed. Despite this, in Chaps. 6, 7, 8, 9, and 10 we will develop mathematically more demanding methods to increase the stability of our RL approach for recommendations.

That concludes our trip around the basic RL methods for our RE framework. Let us now consider their experimental evaluation.

5.4 Experimental Results

In this section we will present experimental results for the approaches of Sects. 5.2 and 5.3. Therefore, we will first verify the central Assumption 5.2 experimentally. After that we extend the simulation of Sect. 4.4 in such a way that we first model the environment, i.e., the conditional transition probabilities and rewards. Then, for the actual simulation, we will use the environment model in order to generate an arbitrary number of virtual sessions for testing the recommendation algorithms under conditions close to reality. At the end of this section, we will use the extended simulation for testing the algorithms introduced in this chapter.

5.4.1 Verification of the Environment Model

We consider the following example.

Example 5.2 We use the data of an online furniture shop We start with the off-line test method of Sect. 4.4. The shop contains approximately 1,900 products. We use the data of one day as training set; it contains 9,736 sessions with 31,349 transactions. The test set consists of the data of the following day; it has 7,430 sessions with 24,161 transactions. Up to removing multiple clicks, we did not change the data.

We want to check the plausibility of Assumption 5.2. In the shop of our test data, no control sessions exist, and all sessions get recommendations of the prudsys RDE. In order to check the influence of the recommendations on the browsing behavior of the shop visitors as good as possible, the RDE varies the recommendations strongly. This was achieved by applying the softmax policy (Sect. 3.3), where the control parameter τ was adjusted to select approximately 50 % of all recommendations as "greedy," i.e., corresponding to the strongest action values, and the remaining 50 % explorative. There are always 4 recommendations displayed for each product.

We use the training data set to determine both the conditional transition probabilities $p^a_{ss_a}$ and the unconditional ones p_{ss_a} by means of the adaptive algorithms 5.1 and 5.2.

We now consider all product views s that actually received at least one recommendation a *and* where at least one rule $s \to s_a$ exists that was learned on the training set. We call this set of product views *recommendation relevant*.

We now follow the notation of Sect. 5.3. Let n be the number of updates of conditional probabilities ${}^n p^a_{ss'}$ and m the number of updates of unconditional probabilities ${}^m p_{ss'}$ of the rule $s \to s'$ on the training data. We define

$$k_{\min} = \min(n, m) \tag{5.28}$$

as minimum of both updates. The higher $k_{\min}$, the better the conditional probability ${}^n p^a_{ss'}$ can be compared with the unconditional ${}^m p_{ss'}$ because for s the recommendation a was sufficiently often delivered (high n) and at the same time also sufficiently often not delivered (high m).

We want to compare the probability types $p^a_{ss_a} = {}^n p^a_{ss_a}$ and $p_{ss_a} = {}^m p_{ss_a}$ depending on $k_{\min}$ in order to see how the recommendation of a increases the transition to s_a. So we calculate the mean values of the transition probabilities over all product recommendation pairs $(s,a)_{k_{\min}}$ whose update number is not smaller than $k_{\min}$, i.e.,

$$\overline{p}_a = \frac{1}{\left|(s,a)_{k_{\min}}\right|} \sum_{(s,a)_{k_{\min}}} {}^n p^a_{ss_a}, \quad \overline{p} = \frac{1}{\left|(s,a)_{k_{\min}}\right|} \sum_{(s,a)_{k_{\min}}} {}^m p_{ss_a}$$

and their coefficient:

Table 5.3 Averaged transition probabilities for different $k_{\min}$

$k_{\min}$	$\overline{p}$	$\overline{p}_a$	$\overline{p}_{\overline{a}}$	rs_C
1	0.007	0.011	0.013	1.49
2	0.012	0.017	0.022	1.49
5	0.021	0.028	0.030	1.31
10	0.024	0.029	0.033	1.21
20	0.024	0.034	0.042	1.40
50	0.024	0.042	0.051	1.78

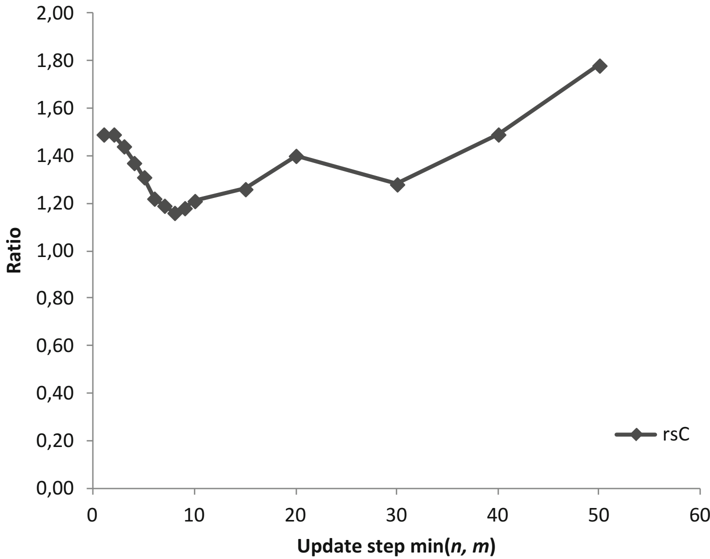


Fig. 5.3 Averaged ratio rs_C of conditional to unconditional transition probabilities for different minimum update steps $k_{\min}$

$$rs_C = \frac{\overline{p}_a}{\overline{p}}.$$

We use Algorithm 5.1. Since we display multiple recommendations, we calculate the conditional probabilities $p^a_{ss_a}$ additionally by Algorithm 5.2 and denote their calculated conditional probabilities by $\overline{p}_{\overline{a}}$. The result is shown in Table 5.3.

The coefficients rs_C are graphically represented in Fig. 5.3. The main behavior looks good: the coefficients are always larger than 1, so displaying recommendations increases the corresponding transition probabilities, and they are not unrealistically large. The graph does not follow any special pattern what is expected, too, since its variations shall be distributed randomly. The only trend we might induce is a slight increase when $k_{\min}$ reached a number with critical statistical volume between 20 and 30.

Table 5.4 Prediction qualities for different prediction methods

		Rate for $p_{ss'}$		Rate for $p_{ss'}^{\{\overline{a}\}}$		Rate for $p_{ss'}^{\overline{a}}$	
$k_{\min}$	n_s	1 rec	3 recs	1 rec	3 recs	1 rec	3 recs
0	15,235	8.72	18.32	7.11	15.01	8.79	19.02
1	13,088	8.90	18.70	7.47	15.61	8.31	17.47
2	10,316	8.82	19.02	7.58	16.51	7.97	17.43
5	6,891	9.26	20.24	7.92	17.21	8.84	18.53
10	4,498	9.20	19.83	8.09	17.23	8.62	17.96
20	1,877	8.95	20.78	9.85	19.82	10.12	20.03
50	172	7.56	23.84	10.47	20.93	10.47	24.42

As we see the special treatment of multiple recommendations does not seem to have great impact. Of course, the relation $p_{ss_a}^{\overline{a}} > p_{ss_a}^{a}$ holds but the difference is relatively small.

Now we will compare our both Assumptions 5.1 and 5.2 regarding their prediction quality of product views (clicks): for each recommendation-relevant product view, we first recommend the products s‘ having the highest unconditional probabilities $p_{ss'} = {}^{m}p_{ss'}$. We use Algorithm 4.1 but applied to all product transitions (instead recommendations only). This corresponds to Assumption 5.1 and the P-Version.

For Assumption 5.2 of the DP-Version, we secondly recommend the products of the highest probabilities $p_{ss'}^{a}$ according to (5.3). Since we have multiple recommendations in the transaction data, we need the probabilities $p_{ss'}^{\overline{a}}$ instead of just $p_{ss'}^{a}$. Their computation was done by Algorithm 5.2. In order to estimate the efficiency of our approach, we will include the unconditional probabilities $p_{ss'}$ calculated by Algorithm 5.2 in the comparison which we will denote by $p_{ss'}^{\{\overline{a}\}}$ in order to avoid confusion with unconditional probabilities $p_{ss'}$ of Assumption 5.1.

The comparison of the prediction methods is again provided for different $k_{\min}$, by imposing the requirement that at least one of the recommendations must satisfy $k_{\min}$. The number of these valid product views is denoted by n_s. For $k_{\min} = 0$ we obtain all recommendation-relevant product views, $n_s = 15{,}235$. With increasing $k_{\min}$ this number is correspondingly decreasing. Furthermore, we test one and three recommendations. The result is given in Table 5.4.

As we can see, $p_{ss'}^{\overline{a}}$ exhibits comparable prediction rates to $p_{ss'}$. At the first sight this may look like a sad result. However, a deeper analysis leads to a more optimistic interpretation. First we emphasize that our aim is not to make good predictions but to find good recommendations. This means that even if our model does not possess the highest prediction quality, as far as it is applicable in principle, the separation into unconditional and conditional probabilities and their right treatment provide an increased return. We will see this impressively in the experiment of the next section where the P-Version exhibits a slightly higher prediction quality than the DP-Version but leads to a much lower return. Having said all this, of course, we do not question the need of good predictions. They are integral for good recommendations.

Second we observe that with increasing $k_{\min}$ the prediction quality of $p_{ss'}^{\overline{a}}$ improves, and for higher $k_{\min}$ it even outperforms $p_{ss'}$. This is because Assumption 5.2 requires a more complex treatment of the data, including a partition of the transactions and their different handling for the two probability types. On the contrary, Assumption 5.1 makes use of all data for unified learning, and hence its algorithms achieve good prediction results even for small statistical volumes. The higher the statistical mass, however, algorithms based on Assumption 5.2 increasingly benefit from their structural advantage and finally outperform the simple ones. Of course, this only applies if Assumption 5.2 is actually realistic! But Table 5.4 seems to confirm that and that's another good news. Finally, we emphasize that Assumption 5.1 used in conjunction with simple update schemas like Algorithm 4.1, though quite simple, exhibits a good overall prediction rate that is really hard to top. We will see this, for example, in Sect. 8.4.4 where we will continue this discussion. ■

So first experience supports Assumption 5.2. The presented results are also confirmed by similar tests on other data sets. Nevertheless, it is too early to speak about a full improvement. Yet our methodology may be subject to another critical objection: despite all random variations by the softmax policy, our recommendations are still the result of previous analyses and thus not fully statistically independent. This raises the question whether the presented results are indeed based on the effect of recommendations rather than their analytical selection.

Luckily the effect can be studied by comparison with the control group. We remember that in the control group no recommendations of the RE algorithm are displayed. In the transaction log files described in Sect. 4.4 (column *itemsAction*), the RDE also stores the products *that it would recommend* if it would be allowed to do that. Since recommendation and control sessions are always mixed in time, these recommendations represent that current one of the RE algorithm. By treating these would-like recommendations in the same way as "real" recommendations, we can repeat all tests and compare them for both recommendation and control group.

Example 5.3 We again used data from a real-world web shop; this time it was a fashion shop. We have analyzed data from two days with (in total) about 12,500 different products and 1.6 Mio. transactions. The procedure was exactly like that of Example 5.2 but now separately for the recommendation and the control group. Although the recommendations have been less explorative than that of Example 5.2, we obtained similar results.

Figure 5.4 shows the quotient of conditional and unconditional probabilities for both groups in the same setting as Fig. 5.3.

Not surprising the control group coefficient rs_C_ctrl is about 1, whereas the one of the recommendation group is higher, between 2 and 3. As in Fig. 5.3 it clearly increases at $k_{\min} = 20$ but then only slightly. The recommendation coefficient rs_C is about twice as high as that of Example 5.2 – this also corresponds to reality (recommendations in the fashion shop are more accepted) and confirmed by click statistics. ■

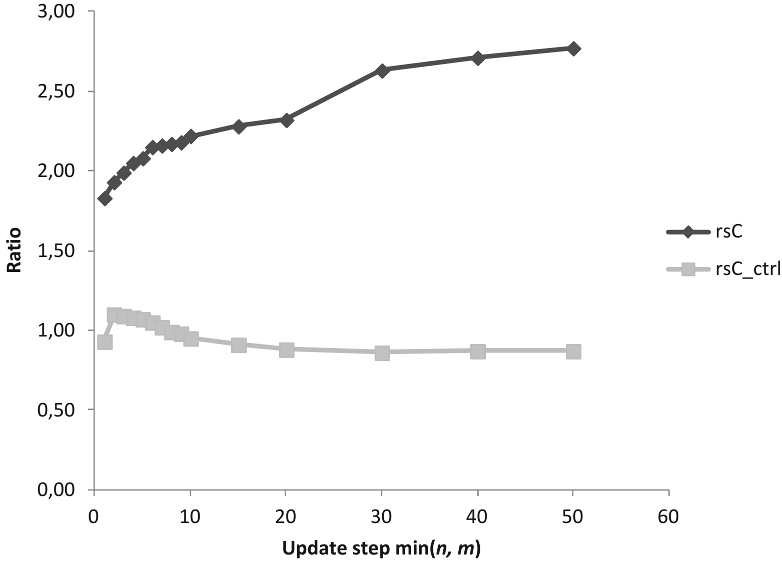


Fig. 5.4 Average probability ratios rs_C and rs_{C}_ctrl for recommendation and control groups

We summarize that first tests indicate the correctness of Assumption 5.2. However, more advanced instruments, like factorizations presented in Chaps. 8, 9, and 10, are required to increase its effectiveness.

5.4.2 *Extension of the Simulation*

We first estimate the model of the environment from the transaction data. This model later will enable us to create an arbitrary number of virtual sessions.

Therefore we subsequently process the transaction data described in Sect. 4.4 and calculate the transition probabilities $p_{ss'}^{\overline{a}}$ using the Algorithm 5.2. For our environment model we also need the transition rewards $r_{ss'}^{a}$ that we estimate by (3.8) in conjunction with Assumption 4.2. To get results of the form of Table 4.2 which includes baskets and orders, we follow a more granular approach, and for all transitions $s \rightarrow s'$ we estimate the probabilities that product s' will be afterward added to the basket as well as the average number of finally ordered units. This enables us to simulate additionally the numbers of baskets and orders as well as the revenue instead the click number only.

In order to generate virtual sessions, we need to know when a session terminates. Here the absorbing state s_A comes to the aid. As soon as we reach the absorbing state according to the transition probability $p_{ss_A}^{\overline{a}}$, we terminate the session. There only remains the question which products to select at the beginning of the sessions. For this we introduce the *generating state* s_G which can be viewed as counterpart to

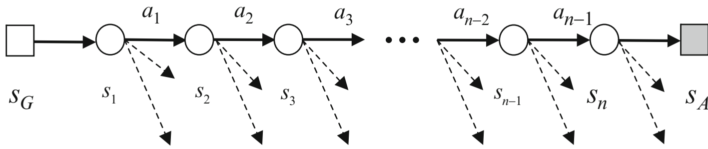


Fig. 5.5 Sequence of products and multiple recommendations as states and actions extended by generating node s_G

the absorbing state s_A. In some sense it is placed before the beginning of each session, and the corresponding transition probability $p_{s_G s}$ specifies the probability that product s will be selected as first product of the session (Fig. 5.5).

From a technical point of view, the transitions from the generating and into the absorbing state can be comfortably stored as rules $s \to s'$ in the same way as for all other product transitions. Thus, the rules $s_G \to s$ represent the transition from the generating state into state s and the rules $s \to s_A$ the transition from state s into the absorbing state.

The flow diagram of the extended simulation is depicted in Fig. 5.6b, whereas Fig. 5.6a shows the basic simulation of Sect. 4.4 for comparison.

In the first phase of the extended simulation, we go through the historical transaction data and estimate the transition probabilities and -rewards which form the model of the environment. In contrast to the simulation of Sect. 4.4, here we do not estimate the unconditional probabilities $p_{ss'}$ but the complete $p_{ss'}^a$ which incorporate the influence of (multiple) recommendations.

This is very important because it allows us to run the actual simulation in the second phase under quite realistic conditions. At the beginning of each session, we calculate the initial (visited) product by virtue of the generating node. This product is passed to the RE algorithm which learns and at the same time delivers recommendations. Based on the current product *and* the recommendations, the simulation environment calculates the next product and decides whether the product will be added to the basket and ordered at the end of the session. This information (including the basket event), in turn, is passed to the RE algorithm which learns again and returns new recommendations, etc. As soon as the transition in the absorbing state takes place, the session terminates. Before the termination, when indicated products marked for purchase are ordered, this information is also transferred to the RE algorithm as tracking event. Then the next session starts. After the specified number of virtual sessions has been reached, the simulation terminates.

An important aspect of the analysis of the extended simulation is that, unlike as in the first simulation of Sect. 4.4, the prediction rates do no longer play the central role. Instead, now the main characteristics are the cumulated values, i.e., the cumulated reward over all sessions and the cumulated numbers of clicks, baskets, orders, and, last not least, the cumulated revenue. In the next section we present the results of the extended simulation for the P- and DP-Version introduced in Sects. 5.1 and 5.2 using an artificial and a real-life data set, respectively. At this, we will state the values of the cumulated rewards only, since it is obvious that they are correlated – depending on their definition – to the shop characteristics like clicks and baskets.

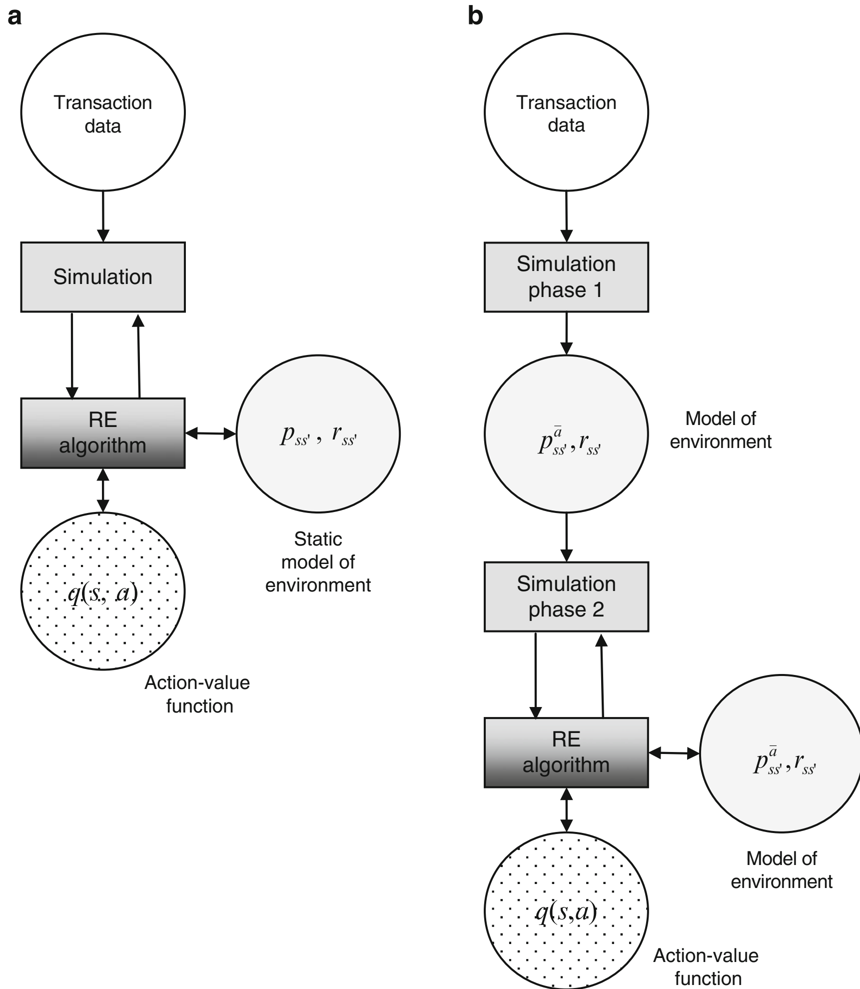


Fig. 5.6 Diagram of both simulation types of Sects. 4.4 and 5.4.2

5.4.3 Experimental Results

Example 5.4 We start our virtual simulation with an artificial example before we turn to a real-life data set.

To do so, we consider a small shop with only 6 products 1–6. We use the reward 1 for clicks and 1 + *pr* if the product was added to the basket, where *pr* is the price of the product. The product prices of our mini shop are listed in Table 5.5.

Suppose there have been the following 4 sessions (star indicates that the product has been added to the basket after it was viewed):

Table 5.5 Products of test shop with their prices

Product ID	Price
1	12.00
2	10.00
3	4.00
4	140.00
5	15.00
6	4.50

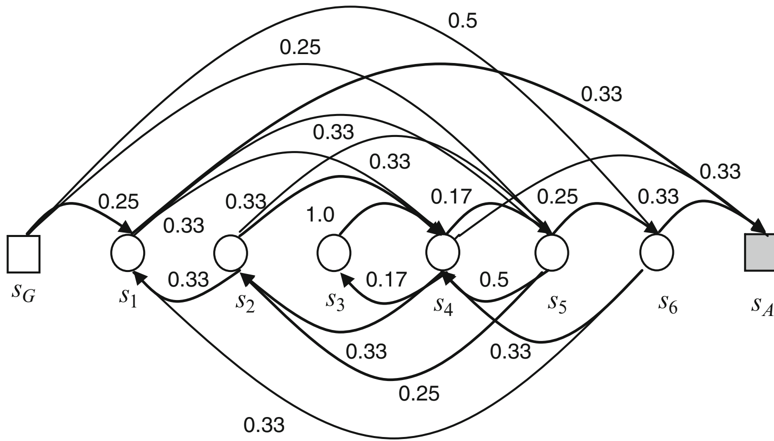


Fig. 5.7 Transition probabilities for the test shop

- $1 \rightarrow 5^* \rightarrow 4 \rightarrow 3 \rightarrow 4^* \rightarrow 5 \rightarrow 4 \rightarrow 2^* \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 6^*$
- $6 \rightarrow 1^* \rightarrow 4$
- $6 \rightarrow 4^*$
- $5 \rightarrow 2 \rightarrow 1$

We further suppose that during the sessions no recommendations have been displayed. Then we can easily calculate the unconditional transition probabilities. They are shown in Fig. 5.7 When we look at the graph $\Gamma(P)$ induced by these transition probabilities P, we see that it is strongly connected and thus P is irreducible.

We next establish 3 different functions to derive the conditional probabilities from the unconditional ones:

(a) $p^a_{ss_a} = \sqrt{p_{ss_a}}$
(b) $p^a_{ss_a} = p_{ss_a}$
(c) $p^a_{ss_a} = p_{ss_a}/2$

Due to Assumption 5.2 all probabilities $p^a_{ss'}$ are completely defined.

For this simple example, we now run the simulation using the DP-Version. First we verify the estimation of the transition probabilities proposed in Sect. 5.2.3.

Table 5.6 Frobenius error norms for simulations over virtual sessions

	1 recommendation				2 recommendations			
	Linear		Nonlinear		Linear		Nonlinear	
#sessions	$\|\|\Delta^c\|\|_F$	$\|\|\Delta^u\|\|_F$	$\|\|\Delta^c\|\|_F$	$\|\|\Delta^u\|\|_F$	$\|\|\Delta^c\|\|_F$	$\|\|\Delta^u\|\|_F$	$\|\|\Delta^c\|\|_F$	$\|\|\Delta^u\|\|_F$
10	1.032	1.642	1.215	1.428	0.579	1.748	1.086	1.950
100	0.505	0.522	0.654	0.781	0.852	1.763	0.240	1.744
1,000	0.233	0.299	0.231	0.256	0.711	1.820	0.102	1.763
10,000	0.057	0.171	0.072	0.093	1.126	1.723	0.040	1.752
100,000	0.021	0.054	0.045	0.088	0.963	1.800	0.020	1.720
1,000,000	0.005	0.012	0.015	0.022	0.579	1.748	0.082	1.679

We separately consider the matrices of the conditional probabilities $P^c := \left\{p_{ss'}^{a_{s'}}\right\}$ $s\in S, s'\in A(s)$ and the unconditional ones $P^u := \{p_{ss'}\}_{s\,\in\,S, s'\,\in\,A(s)}$. Let the symbol tilde represent the probabilities estimated by the algorithms of Sect. 5.2.3. Then we use the Frobenius norm of the differences matrices $\Delta^\alpha := P^\alpha - \widetilde{P}^\alpha, \alpha\in\{c, u\}$ to estimate the error. The *Frobenius norm* of a matrix A is defined as

$$\|A\|_F^2 := \sum_{i=1, j=1}^{m, n} |a_{ij}|^2, \quad A\in\Re^{m\times n}. \tag{5.29}$$

Table 5.6 contains the errors $\|\Delta^\alpha\|_F$ for the function of case a) for one and two recommendations depending on the number of sessions. We compare the linear Algorithm 5.2 with the nonlinear Algorithm 5.3.

The result confirms that for one recommendation both algorithms work equally well. However, for two recommendations the linear algorithm fails completely, and the nonlinear works only for the conditional probabilities. The reason for this behavior is that the unconditional probabilities are not estimated correctly.

Since the linear Algorithm 5.2 is based on a joined estimation of conditional and unconditional probabilities, this also leads to wrong estimates of the conditional probabilities. In contrast, the nonlinear Algorithms 5.3 and 5.4 both estimate the conditional probabilities independent on the unconditional ones.

What is the reason for the wrong calculation of unconditional probabilities? We suffer from a special case here. If the number of recommendations k is equal to the number of successor states m, the unconditional probabilities cannot change because the fixed component includes all unconditional probabilities $\Pi_{\overline{a}} = \{p_{ss'}\}_{s'\in S_{\overline{a}}}$. Moreover, the same applies to the case where the $k = m\text{-}1$. This follows from the relation $\sum_{s'\notin S_{\overline{a}}} p_{ss'} = 1 - \sum_{s'\in S_{\overline{a}}} p_{ss'}$, where the right-hand side is fixed, and the fact that the "free" set $S_{\overline{a}}^C = \{s' \notin S_{\overline{a}}\}$ consists of one element only. Notice also that this problem is truly a special case and far from reality since usually the number of successor states available is quite high, so that $m \gg k$.

To overcome this problem, we just need to omit the recommendations from time to time. In our test, each 10th request did not issue recommendations, i.e., then $k = 0$ applied. The result is shown in Table 5.7. Now the algorithms work correctly.

Table 5.7 Frobenius error norms for simulations over virtual sessions with sometimes recommendations left

	2 recommendations			
	Linear		Nonlinear	
#sessions	$\|\Delta^c\|_F$	$\|\Delta^u\|_F$	$\|\Delta^c\|_F$	$\|\Delta^u\|_F$
10	1.458	1.044	1.095	1.382
100	1.073	0.667	0.189	0.616
1,000	0.508	0.240	0.051	0.151
10,000	0.172	0.102	0.029	0.069
100,000	0.041	0.016	0.004	0.017
1,000,000	0.006	0.001	0.002	0.086

Table 5.8 Cumulated rewards for P- and DP-Versions for simulations over virtual sessions

	$p^a_{ss_a} = \sqrt{p_{ss_a}}$		$p^a_{ss_a} = p_{ss_a}$		$p^a_{ss_a} = p_{ss_a}/2$	
γ	P	DP	P	DP	P	DP
0.0	17,258,161	18,439,767	8,671,018	8,671,018	5,909,776	7,591,168
0.5	25,511,885	25,348,836	8,671,018	8,671,018	4,485,351	8,937,920
1.0	23,418,997	25,710,600	8,671,018	8,671,018	4,853,128	8,957,431

Next now compare the P-Version (Sect. 5.1) with the complete DP-Version (Sect. 5.2) for this example using the cumulated rewards over all sessions. We use only one recommendation and Algorithm 5.1 in the DP-Version.

Table 5.8 shows the results for 100,000 sessions depending on the three cases a)–c) and for different discount rates γ.

Case (b) is clear: the conditional probabilities are identical to the unconditional ones, and so recommendations are without effect. Provided the conditional probabilities are larger than their unconditional counterparts, namely, in case (a), what means that delivery of recommendations increases the probability of the transition into the recommended state, both P-Version and DP-Version perform equally well. Also the chain optimization turns out to be effective.

More complex is case (c). It may look a bit academic, because here on the contrary the delivery of recommendations decreases their transition probability, but is important for the understanding of the methods. Here the DP-Version performs much better than the P-Version. As regards content this is clear because the P-Version in some sense suffers from a kind of Russell's paradox: it recommends products with highest $p_{ss_a} r_{ss_a}$, but by recommending them on the contrary, they were accepted less frequently! The chain optimization makes the situation even worse, because it calculates the expected rewards more accurately, and in doing so it further worsens the recommendations! All this is certainly rooted in the fact that for case c) Assumption 5.1 concerning the P-Version is not only simply violated but turned into its complete opposite. At the same time we see that the DP-Version handles the problem correctly.

Table 5.9 Cumulated rewards for P-, DP-, and P-DP-Versions for simulations over virtual sessions

γ	P-Version	DP-Version	P-DP-Version ($n_{\min} = 50$)
0.0	443,005.74	536,764.94	540,197.18
0.5	435,934.75	584,838.17	585,926.50
1.0	450,942.64	608,496.66	601,278.30

In reality the connection between conditional and unconditional probabilities is, of course, more complex and constitutes a qualitative mixture of all three cases a)–c). Fortunately, the case a), where the conditional probability is higher than the unconditional one, dominates as we have also seen in Sect. 5.4.1. This explains why the P-Version in most practical applications works very well. ■

Example 5.5 Now we return to the data set of Example 5.2 and run a virtual simulation with 100,000 sessions. We use 3 recommendations and the estimation of the transition probabilities $p^a_{ss'}$ was performed by Algorithm 5.3.

We now again compare the P-Version with the DP-Version for different discount rates. Additionally, we test the combined P-DP-Version with the threshold value $n_{\min} = 50$. In all versions, the ADP Algorithm 3.3 was applied, in which the full calculation of the state-value function in line 9 was provided for each 1,000th session.

Table 5.9 shows the results.

From the results we see that the DP-Version clearly outperforms P-Version. Also the chain optimization works well, for the DP-Version better than for the P-Version. In contrast, the results of the P-DP-Version do not indicate a clear improvement compared to the DP-Version. ■

5.5 Summary

This chapter was mainly devoted to the question of estimating transition probabilities taking into account the effect of recommendations. It turned out that this is an extremely complex problem. The central result was a simple empirical assumption that allows reducing the complexity of the estimation in a way that is suitable to most practical problems. The discussion of this approach gave a deeper insight into essential principles of recommendation engines. Based on this assumption we proposed methods to estimate the transition probabilities and provided some first experimental results. Although the results look promising, more advanced techniques are highly desirable. This will be the central topic of the next chapters.

Chapter 6
Up the Down Staircase: Hierarchical Reinforcement Learning

Abstract We address the question of how hierarchical, or multigrid, methods may figure in dynamic programming and reinforcement learning for recommendation engines.

After providing a general introduction, we approach the framework of hierarchical methods from both the historical analytical and algebraic viewpoints; we proceed to devising and justifying approaches to apply hierarchical methods to both the model-based as well as the model-free case. In regard to the latter, we set out from the multigrid reinforcement learning algorithms introduced by Ziv in [Ziv04] and extend these methods to finite-horizon problems.

Back in Chap. 4 we established that reinforcement learning methods usually converge only slowly. The introduction of hierarchical concepts is necessary in order to solve this problem. Instead of trying to solve the Bellman equation (3.6) directly, using the GPI method on enormous quantities of states and actions, we must split the problem into a hierarchy of subtasks and then solve them in succession.

For many years, the development of hierarchical solution methods has been one of the central fields of research for RL. Here we commonly distinguish between *temporal* approaches, that is, the aggregation of actions over a sequence of steps, and *spatial* approaches, that is, the aggregation over states. There exist a number of spatial approaches [AR02, Diet00, MRLG05, PR98, SPS99]. However, they do not seem to be suitable for our recommendation engine approach.

But state aggregations do. We remember that the Bellman equation can be regarded as a discrete counterpart of a differential equation. Therefore, it is evident that the multilevel methods developed in the course of numerical analysis – which are used especially for differential equations – can be applied to provide a solution to the Bellman equation. This approach was first developed by the Israeli information technologist Omer Ziv in his remarkable doctoral thesis [Ziv04], in which at the same time he proved fundamental convergence statements. We can develop these approaches for recommendation engines further, in

A. Paprotny and M. Thess, *Realtime Data Mining: Self-Learning Techniques for Recommendation Engines*, Applied and Numerical Harmonic Analysis,
DOI 10.1007/978-3-319-01321-3_6, © Springer International Publishing Switzerland 2013

particular by using the isomorphism (4.1) between states and actions in REs and so further extend the concept of the hierarchical splitting of states to actions.

6.1 Introduction

We will approach the problem of hierarchical methods from two sides: firstly from the historical – analytical – viewpoint, then from the algebraic viewpoint. We deliberately omit most of the mathematical infrastructure, which in parts is exceedingly complex, and attempt to explain the underlying ideas in an understandable (and sometimes slightly simplified) fashion.

6.1.1 Analytical Approach

So far we have only ever considered the state-value and action-value functions *v(s)* and *q(s, a)* in tabular form. However, we are dealing with functions, and so in RL we often have to resort to approximation methods such as linear and polynomial functions or, for instance, neural networks in order to represent them using only a few coefficients. We therefore start now from the actual functions.

Since most useful function spaces V are infinite dimensional, we will instead consider finite-dimensional subspaces $V_n \subset V$, where n is their dimension. In most cases this is the central assumption for being able to efficiently find a numerical solution of the associated operator equation.

We can represent a finite-dimensional function $f_n \in V_n$ as follows:

$$f_n(x) = \sum_{i=1}^{n} c_i \phi_i(x), \tag{6.1}$$

where ϕ_i are the basis functions and c_i are their coefficients. By inserting the proposition (6.1) into the operator equation, for instance, the Bellman equation (3.7), we can reduce the determination of the function $f \in V$ to the determination of the coefficients c_i of our approximated function $f_n \in V_n$. (We are omitting now subtleties such as the use of test functions in variation formulations.)

Now as a rule there are several bases for a function space V_n. Let us consider in addition to the basis $\Phi_n = [\phi_1,\ \phi_2, \ldots, \phi_n]$ another basis $\Psi_n = [\psi_1,\ \psi_2, \ldots, \psi_n]$ in V_n. Then every function $f_n \in V_n$ also over the basis Ψ_n can be represented by coefficients d_i:

$$f_n(x) = \sum_{i=1}^{n} c_i \phi_i(x) = \sum_{i=1}^{n} d_i \psi_i(x).$$

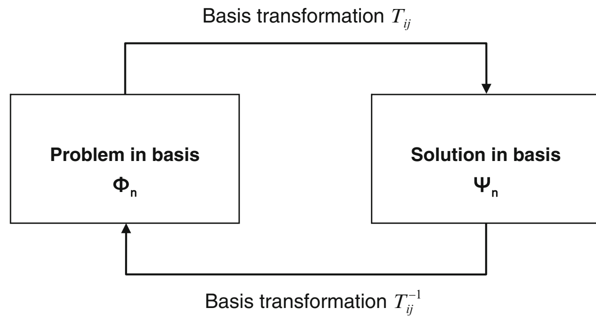


Fig. 6.1 Using the dual basis over basis transformation

The matrix $T_{ij}\big|d_i = \sum_{j=1}^{n} T_{ij}c_j$ thereby defines the *basis transformation* of Φ_n into Ψ_n and its inverse T_{ij}^{-1} the basis transformation from Ψ_n into Φ_n. Now we can state one of the most fundamental findings of numerical analysis during the 1990s:

> **Basis principle:** The selection of the correct basis is of central importance to the solution of many tasks in numerical analysis. A basis can also be selected virtually by transforming a problem into the dual basis, solving it there (at least approximately) and transforming it back again.

In a word, you have to have the correct basis! But there is also a body of opinion that states that the critical point is the correct selection of the function space by which the functions are specified. What role does the basis then play? The reality in theory and practice teaches us however that different tasks can be solved in different bases with greater or lesser efficiency. And there's more: In most cases the function space is already specified by the practical requirements. The basis is thus central.

In some cases the operator equation is formulated in a suitable basis from the start. This applies in particular if the formulation in the basis is efficient (e.g., in the case of sparse grids described below). Most cases however are processed using a dual basis: the problem is formulated in the basis Φ_n that is the most efficient for the formulation and is solved in the basis Ψ_n that is the most efficient for the solution. The idea of changing the basis is illustrated in Fig. 6.1.

In this connection we should also mention that the idea of basis transformation is extraordinarily powerful and in a general sense goes well beyond approximation theory and even beyond mathematics. So, for instance, the transfer of data to a data warehouse can also be interpreted (in a generalized way) as a basis transformation. While the data in the operative systems is in most cases held in relational form, it is transferred to a data warehouse by ETL processes, where it is stored in multidimensional form. The data is thus in principle the same, but while in the relational form of the operative systems it is better suited to updating and extension, the multidimensional form in the data warehouse is better suited to analysis.

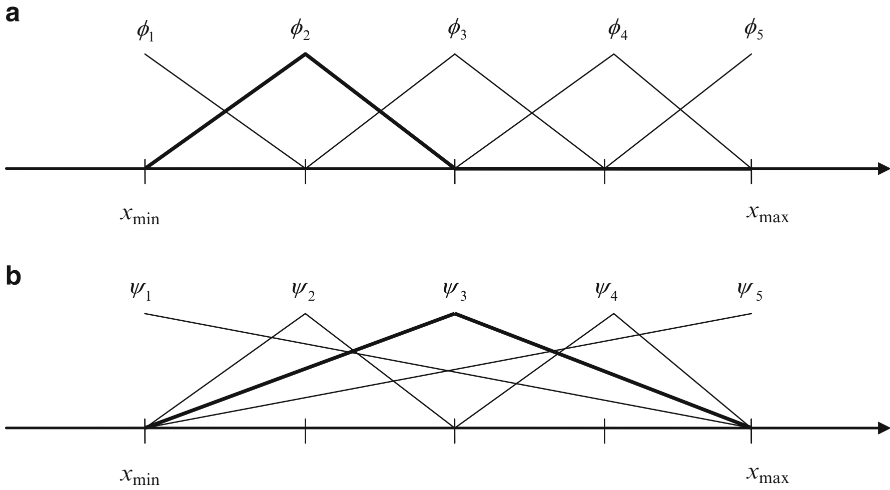


Fig. 6.2 Nodal basis (**a**) and multilevel basis (**b**) at the interval $[x_{min}, x_{max}]$. For the nodal basis the function ϕ_2 is shown in bold; for the multilevel basis the function ψ_3 is in bold

In practice of approximation theory, two types of bases are of special importance: the nodal basis and the multi-scale basis. The nodal basis has a local support and is easy to use in practice. It can well approximate jumping functions ("well" means it requires few coefficients), but poorly however smooth functions. An example of a nodal basis in the one-dimensional case is shown in Fig. 6.2a for the linear case, where this basis is also called the Courant hat function. In its assigned node this takes the value 1 and declines linearly to the two adjacent nodes. Apart from this it is constantly 0. The Courant hat function can easily be calculated and is often used in finite element method (FEM) approaches.

The multi-scale basis has in most cases a global support and employs basis functions of different "frequencies." It is usually less easy to use. In contrast to the nodal bases, multi-scale bases well approximate smooth functions but poorly however jumping functions. A classic example is the Fourier basis. An example of a multi-scale basis in the same linear function space as in Fig. 6.2a is shown in Fig. 6.2b. This type of basis is also called a multilevel basis, since its nodes are distributed hierarchically in different levels. In the example, the basis function nodes ψ_1, ψ_3, and ψ_5 form the coarse grid and are called coarse grid functions. The basis function nodes ψ_2 and ψ_4 form the fine grid and are called fine grid functions (in our example they correspond to the basis functions ϕ_2 and ϕ_4 of the nodal basis). Of course also more than two levels can be used. (Note: In contrast to the Fourier basis, in the multilevel basis the support of the functions is global only to a certain degree, and in fact the multilevel basis considered here exhibits poorer approximation properties for smooth functions than does the Fourier basis. We use it however in the interests of a better illustration, since its function space is effectively identical to that of the nodal basis under consideration.)

In summary one can say, in the language of signal processing, that nodal bases localize well in time and multi-scales bases localize well in frequency. Now we want to combine the two approaches – nodal basis and multi-scales basis. (We will come to this later.) The Heisenberg uncertainty principle applies here: the product of time and frequency resolutions is always greater than a natural constant. Using Δt as the time interval and Δf as the frequency interval, we have:

$$\Delta t \cdot \Delta f > \frac{\pi}{4}. \tag{6.2}$$

If the time resolution increases, of necessity the frequency resolution decreases, and vice versa. This is a fundamental relationship in multi-scale approximation theory.

In practice, nodal bases are mostly used to formulate problems: for instance, in signal processing, sounds (1D) or images (2D) can immediately be recorded electronically in the nodal basis. Nodal bases are also preferred for solution of differential equations in the field of FEM, because of their flexibility for modeling. In practice however most of the cases we are dealing with are for the most part smooth functions (speech in signal processing, images in image processing, deformations or flows in differential equations, models in data mining, etc.). In most cases these can be better approximated in a multi-scale basis.

The result of this is to use the most efficient method as shown in Fig. 6.1. The problem is formulated in a nodal basis Φ_n, then a basis transformation is applied to convert the function (or the error) into a multi-scale basis Ψ_n; the problem is solved there and by means of the inverse basis transformation converted back into the nodal basis Φ_n.

Examples 6.1 We now give a few examples of the use of basis transformations into multi-scale bases:

- *Data compression*: The signals (sounds, images, etc.) are converted by basis transformations such as Fourier or wavelet transformation (called encoding) into the multi-scale basis. There they can be represented efficiently, that is, with few coefficients d_i and thus efficiently stored and transmitted. As soon as they are required again, the inverse basis transformation (decoding) is performed, and the signals are once again available in the practical nodal basis.
- *Signal processing*: Process as described in data compression. In addition the signals can be better analyzed and smoothed in the multi-scale basis. For instance, for smoothing, the coefficients d_i of the high-frequency basis functions can simply be set to 0.
- *Solution of differential and integral equations*: The operator equation is formulated in the nodal basis, after which a basis transformation is applied to convert it to the multi-scale basis. For most important differential and integral equations, it can be shown that the solution by means of iteration methods in the space of good multi-scale bases is asymptotically optimal. After the efficient solution in the space of the multi-scale basis, the solution is transformed back to the nodal basis. ■

Let us remain a little longer with the last case of the efficient solution of differential equations and go into a little more detail, since this will help us for the Bellman equation.

Let us consider the original approach, the *multigrid* method. It was initially developed by the Soviet mathematicians Fedorenko [Fed64] and Bakhvalov [Bakh66], and main contributions are made by Achi Brandt [Bra77] and Wolfgang Hackbusch [Ha85].

After the discretization of the differential equation has been performed, the FEM approach leads via nodal bases to the solution of the equation system

$$Kx = f, \tag{6.3}$$

where K is what we call the stiffness matrix, f is the load vector, and x is the solution vector for our coefficients c_i in the nodal basis. Let IT be a simple iteration method like Richardson or Gauss-Seidel. For the solution of (6.3), it requires a lot of iterations and is therefore very slow.

Then the solution approach of multigrid is as follows: at every iteration step i, we start with the current solution vector x^i, perform some iteration steps ν_1 with the simple iteration method IT, and obtain the new approximation x_f^i (relaxation). Now it can be shown that in the new approximation x_f^i, the high-frequency components of the error are significantly reduced, but the low-frequency components hardly at all. Therefore, we calculate the residuum:

$$y_f^i = Kx_f^i - f$$

and project it onto the coarse grid using what we call *restrictor* $I_f^g : y_g^i = I_f^g y_f^i$. Now we solve the equation system for the coarse grid:

$$K_g w_g^i = y_g^i,$$

where K_g is the coarse grid matrix. The coarse grid matrix can be calculated either by directly discretizing (6.3) on the coarse grid or by application of the restrictor I_g^f and *interpolator* I_g^f as a *Galerkin* operator

$$K_g = I_f^g K I_g^f.$$

We project the calculated correction vector w_g^i by use of the interpolator I_g^f back again on to the fine grid $w_f^i = I_g^f w_g^i$ to obtain the new approximation:

$$\hat{x}^{i+1} = x_f^i + w_f^i.$$

Using $\hat{x}^{i+1}$ as start vector, we again run some iteration steps ν_2 by our simple iteration method IT and obtain the new iterate x^{i+1}. After this the iteration starts all over again.

Instead of using only two grids, we can recursively repeat the whole method over several grids – that is, perform a multigrid method. The procedure for four

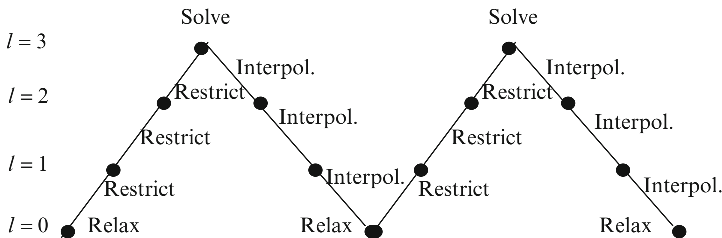


Fig. 6.3 Multigrid method (V cycle) for four levels by two iteration steps

levels is shown schematically in Fig. 6.3. We are running up the down staircase, which is also the title of this chapter in acknowledgment of the well-known American film.

The multigrid method we discussed is summarized in Algorithm 6.1. At this, the simple iteration method *IT* is called with the current start vector and the right-hand side as arguments, that is,

$$\widetilde{x} := IT(\widetilde{x},\ \ y)$$

approximately calculates $K^{-1}y$, where $\widetilde{x}$ is the start vector, and assigns the result to $\widetilde{x}$.

Notice that the terms "restrictor" and "interpolator" for the inter-level operators I_f^g and I_g^f are rather unusual in multilevel approximation theory. In general, we speak about *restriction* and *interpolation* (or *prolongation*). Most important terms are *restriction operator*, *interpolation operator* (or *prolongator*), *restriction matrix,* and *interpolation matrix*. However, to keep the notation short, we will stick to *restrictor* and *interpolator*.

Algorithm 6.1: Multigrid V-cycle

Input: coefficient matrix $K \in \Re^{n \times n}$, right-hand side $f \in \Re^n$, interpolators I_{l+1}^l, restrictors I_l^{l+1} for all levels $l = 0, \ldots, l_{\max} - 1$, number of pre-smoothing steps ν_1 and post-smoothing steps ν_2, initial guess $x^0 \in \Re^n$

Output: approximate solution $\widetilde{x} \in \Re^n$ von (6.3)

1: **procedure** VCYCLE(y^l,l)
2: **for** $k = 1, \ldots, \nu_1$ **do**
3: $x^l := IT(x^l,\ K^l x^l - y^l)$ ▷ pre-smoothing
4: **end for**
5: $y^{l+1} := I_l^{l+1}(K^l x^l - y^l)$ ▷ computing the residual
6: **if** $l + 1 < l_{\max}$ **then**

(continued)

Algorithm 6.1: (continued)

7: $x^{l+1} := \text{VCYCLE}(y^{l+1}, l)$ ▷ recursive calls at coarser grid
8: **else**
9: $x^{l+1} := (K^{l+1})^{-1} y^{l+1}$ ▷ direct solver on the coarsest grid
10: **end if**
11: $x^l := x^l + I^l_{l+1} x^{l+1}$ ▷ coarse grid correction
12: **for** $k = 1, \ldots, \nu_2$ **do**
13: $x^l := IT(x^l,\ K^l x^l - y^l)$ ▷ post-smoothing
14: **end for**
15: **return** x^l
16: **end procedure**
17: **return** VCYCLE(f,0) ▷ initial call

We do not wish here to explore the complicated refinements and numerous variants of the multigrid approach, much less the mathematical proof of its optimality. What is critical is the fundamental idea:

Multigrid approach: If for an operator equation the high-frequency error components can be quickly reduced by classic iteration methods, the use of the multigrid methods in the concrete and multi-scale methods in general leads to the efficient solution of the operator equation. This applies particularly to wide classes of differential and integral equations.

Now the multigrid approach is not a basis transformation (since the multigrid hierarchy, which represents a *generating system*, is not unique in relation to the coefficient splitting and thus is not a basis), but it is possible to find multi-scale bases which work equally well. For this, what we call *wavelets* play a central role. Wavelets have a long history; modern wavelet theory is largely based on Stéphane Mallat *multiresolution analysis* [Ma99] and the work of Ingrid Daubechies [Dau92].

Wavelets are a combination of nodal and global multi-scale bases: they use multiple levels, but always a quasi-local support. The *multilevel basis* in Fig. 6.2b is essentially a wavelet basis (more precisely, a bi-orthogonal wavelet basis), although classic wavelets are orthogonalized and hence more complicated.

Although the uncertainty principle (6.2) still applies for wavelets of course, in many respects they constitute an optimal compromise between a nodal and global multi-scale basis: they are asymptotically optimal for the smoothing and compression of smooth signals and for solving differential and integral equations. In addition, using special anisotropic grids which we call *sparse grids,* it is possible for the first time to approximate high-dimensional smooth functions efficiently. By this means, for instance, differential equations in a 20-dimensional space can be solved (Fig. 6.4).

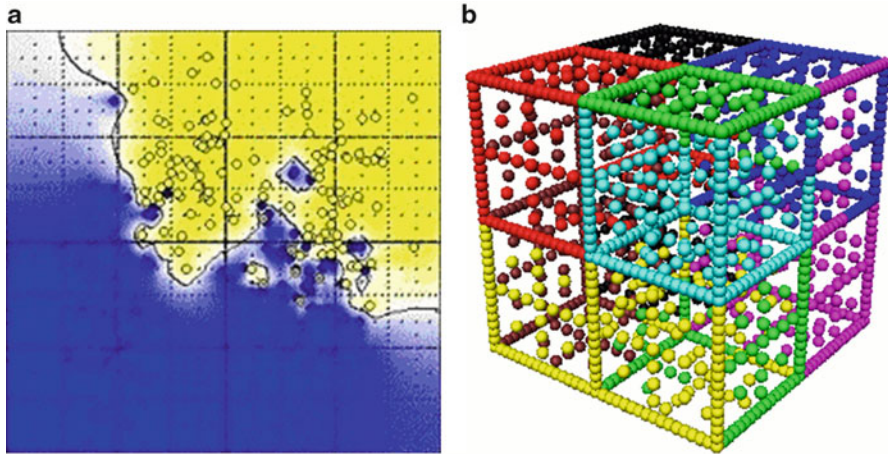


Fig. 6.4 (**a**) Sparse grid and function in 2D, (**b**) sparse grid in 3D [Zu00]

We will describe sparse grids in detail in Chap. 7 and summarize once again the advantages of wavelets. Wavelets are (asymptotically) optimal:

- For the signal processing and compression of smooth signals
- For the solution of the central classes of differential and integral equations
- For the approximation of multidimensional smooth functions

In a certain sense they represent the optimal compromise between particles (nodal basis) and waves (multi-scale basis): by this means for the first time we can optimally solve multidimensional operator equations and optimally transmit, smooth, and analyze their solution functions. Speaking philosophically, the revolutionary wave-particle dualism of quantum physics (Chap. 1) has its approximation theory counterpart here. Therefore, multi-scale bases and wavelets represent one of the hottest research topics in numerical analysis. For that reason we want to use this groundbreaking approach for RL too.

6.1.2 Algebraic Approach

After so much euphoria on the topic of hierarchical bases and wavelets, we now encounter the first problems. In the case of the recommendation engine, in accordance with the modeling in Chap. 4, our states and actions are inherently discrete. Therefore, we cannot directly change over to a continuous state-value and action-value function $v(s)$ and $q(s, a)$. We know however that the Bellman equation (3.6) is the discrete counterpart to the Hamilton-Jacobi-Bellman differential equation. The question then arises of whether despite this the multilevel approach is useful here.

The answer is provided by access via algebraic multigrid methods (AMG). These constitute the algebraic counterpart to the analytical multigrid method

introduced above. It was initiated in 1982 by Achi Brandt, Steve McCormick, and John Ruge [BMR82] and especially improved by Stüben [TOS01].

The historical starting point was the recognition that FEM equations in technical practice are often solved for objects with complex structures (house roofs, cars, airplanes, etc.). Thus, it is difficult to define a hierarchy on the FEM grids. Now however it has been established, in particular for discretized differential equations (6.3) under consideration of their eigenvalue properties of the stiffness matrix K, that *algebraic smooth error vectors* arise. That means that while local errors are quickly reduced by iteration methods, the smooth error components reduce only relatively slowly. However, according to our multigrid approach, multilevel methods are outstandingly suitable for solution of these problems.

In contrast to the analytical multigrid, the proofs of the convergence speed are often incomplete in the algebraic case. In practice however AMG has proven itself very powerful in most cases.

In what follows, we shall provide a brief, but mathematically sound introduction to algebraic multigrid methods. To this end, we need to stipulate an algebraic notion of a grid and specify what is meant by an algebraically smooth error vector.

Consider a system of linear equations of the form

$$Ax = b, \tag{6.4}$$

where A is a non-singular real $N \times N$-matrix and b is a real vector of length n, and let (M, N) be a *splitting* of A, that is, $A = M–N$ and M is non-singular. It can easily be verified that the sequence of iterates generated by the update rule

$$x := x + M^{-1}(b - Ax). \tag{6.5}$$

converges to the solution of (6.4) for all initial guesses if and only if the *spectral radius*, that is, the largest modulus of the eigenvalues, of the iteration matrix $S := M^{-1}N$ is strictly smaller than 1. Moreover, the spectral radius of S coincides with the asymptotic rate of convergence (with respect to *any* norm).

Hence, if the spectral radius of S is close to 1, the method will exhibit slow convergence. In the above introduced setting where (6.4) is a finite differences discretization of a Poissonian boundary value problem, it holds that the spectral radius of the discretized system converges to 1 as the number of grid points increases. Hence, both the cost of one iteration and the number of iterations required to achieve a prescribed accuracy grow with the number of grid points. Expressing the error in terms of a basis of eigenvectors of the iteration matrix, one observes that the coefficients converge to 0 at an asymptotic rate given by the magnitude of the corresponding eigenvalue. In the PDE setting, this is the algebraic explanation for slow convergence of smooth errors. In the case where A is a discretization of a Laplacian differential operator, eigenvectors of the iteration matrix corresponding to eigenvalues close to 1 are geometrically *smooth*, whereas those corresponding to small eigenvalues are *oscillatory*. In a purely algebraic setting where there is no underlying differential equation, we retain this geometric intuition as a metaphor

and refer to error vectors that are dominated by contributions in directions of eigenvectors of the iteration matrix that correspond to small-magnitude eigenvalues as (algebraically) *smooth*. Similarly to the PDE setting, we would like to eliminate these smooth error components by means of a correction step on an algebraic coarse grid, the notion of which shall be specified subsequently.

In compliance with the notation introduced in foregoing chapters, we shall denote the set of the first n natural numbers by $\underline{n}$. Each element of this set is an index corresponding to an entry in of a vector of length n. With the graph-theoretic framework introduced in Sect. 3.9.2 in mind, we also refer to such an index as a node. So, a reasonable approach to mimicking the multi-scale framework in the continuous setting consists in considering a set of nodes, that is, a subset of $\underline{n}$, as an (algebraic) grid. The *fine grid*, then, is simply the set $\underline{n}$ itself, and a *coarse grid* is a suitably chosen subset $\underline{m}$ of $\underline{n}$ such that the number m of nodes therein is considerably smaller than n. Given a *coarse grid* $\underline{m}$, we construct, by means of a method yet to be specified, a restriction operator enabling to restrict a given residual

$$r := b - Ax$$

to the coarse grid and an interpolation operator allowing us to prolong the coarse grid correction to the fine grid. If we denote these operators by R and L, respectively, the algebraic version of the coarse grid correction step in the V-cycle (Algorithm 6.1) becomes

$$x := x + L\widetilde{x},$$

where $\widetilde{x}$ is the solution of the *coarse grid equation*

$$RAL\widetilde{x} = Rr. \tag{6.6}$$

Combining the above equations, summarize the coarse grid correction step in a single assignment:

$$x := x + L(RAL)^{-1}R(b - Ax). \tag{6.7}$$

To provide some evidence for the soundness of the approach, we shall, for now, assume that A be symmetric and positive definite and $R = L^T$. As it turns out, this case is easy to handle mathematically. The crucial fact is that a symmetric positive definite matrix induces the so-called energetic inner product $\langle \cdot, \cdot \rangle_A$ given by

$$\langle x, y \rangle_A := x^T Ay.$$

The catch consists in carrying out the error analysis in terms of the thereby induced norm

$$\|x\|_A := \sqrt{\langle x, x \rangle_A}.$$

Let $e := x - A^{-1}b$ denote the error vector of the current iterate before the coarse grid correction. Equation (6.7) reveals that the error vector immediately after the coarse grid correction is given by

$$e^{'} = \left(I - L\left(L^{T}AL\right)^{-1}RA\right)e.$$

But $I - L(L^{T}AL)^{-1}RA)$ is the orthogonal projector along the range of L in terms of the energetic inner product. Hence, the following relation holds:

$$\left\|e^{'}\right\|_{A} = \min_{z} \|e - Lz\|_{A}, \tag{6.8}$$

that is, the coarse grid correction is the best approximation to the smooth error vector in terms of the energetic inner product. In particular, this implies that

$$\left\|e^{'}\right\|_{A} \leq \|e\|_{A},$$

that is, the coarse grid correction can never increase the error, no matter which coarse grid and interpolation operator have been chosen. Furthermore, Eq. (6.8) gives a decisive hint as to how to establish the prolongation operator: to eliminate smooth error components, the range of L should be a good approximation to the space spanned by smooth eigenvectors of the iteration matrix S. In the geometric setting with a regular grid, this may be achieved by coarsening the grid and using, for example, piecewise linear interpolation.

In the algebraic setting, however, there remains the question of the construction of a suitable grid hierarchy and corresponding inter-level operators. Interestingly, by means of what we call *coarsening*, an interpolator can be extracted automatically from the Eq. (6.3) (and from this by the inverse the restrictor and by the Galerkin operator the stiffness matrix on the coarse grid). This might again sound like Baron Münchhausen, but we do exactly the same thing when solving most operator equations: their structure is analyzed in the preprocessing and the best solution method derived from it.

Apart from this, it is also possible to establish a grid hierarchy and inter-level operators intellectually by exploiting certain structural properties arising from the underlying model. We shall do so to devise AMG schemes dedicated to RL in the next section.

Sadly enough, the coefficient matrices arising from RL for recommendation matrices are typically not symmetric, let alone positive definite. As it lacks a notion of an energetic inner product, the nonsymmetric case defies a mathematical analysis as profound and simple as the above-outlined. Moreover, eigenvalues of nonsymmetric iteration matrices may be complex, there need not be a basis of eigenvectors, and, even if, this basis is generally not orthogonal in terms of any sensible inner product.

Nevertheless, if we can guarantee that the matrix of the coarse Eq. (6.6) be non-singular, the AMG scheme is at least well defined. Moreover, it is supported by numerical evidence, and, as we shall see in the next section, results on convergence rates, though not as neat as those for the symmetric case, may be obtained.

A step toward understanding convergence properties of the method in the nonsymmetric case is the insight that the operator $(I - L(R^TAL)^{-1}RA)$ transforming the error vector before to that after the coarse grid projection, that is,

$$e' = \left(I - L\left(R^TAL\right)^{-1}RA\right)e,$$

though not orthogonal in general, is always a projector (i.e., its square equals itself) along the range of the interpolation operator. Moreover, it can be shown that there is always an inner product in which the correction operator is an orthogonal projector (Proposition 3.6.2 in [Pap10]). The iteration matrices corresponding to standard splittings, however, are no contractions with respect to such an inner product in general. Hence, it is in some cases easier to analyze the asymptotic convergence rate of applying the V-cycle procedure in an iterative fashion.

For the sake of completeness, we should mention that it is possible to circumvent the nonsymmetric case by applying an AMG procedure to the equivalent system

$$A^TAx = A^Tb,$$

which has a symmetric and positive definite coefficient matrix if A is non-singular. This approach, however, brings along difficulties of its own. First, the condition of the symmetrized matrix A^TA is square of that of A, which renders the solution considerably more sensitive to perturbations in the data. Furthermore, many structural features of A, such as sparsity, are not inherited by the symmetrized system. Therefore, the symmetrized approach turns out to be unsatisfactory in most situations.

With this we come to the last point of this introduction to hierarchical methods for acceleration of convergence: multilevel splitting can be used in different ways: directly, as multigrid or as preconditioners, additively and multiplicatively, etc. It is beyond the scope of this study to present them all individually. We refer here in particular to the Abstract Schwarz Theory [Os94], which gives unified access via basis transformations to virtually all multilevel methods – including sparse grids. In the next sections we will concentrate primarily on the algebraic construction of the grid hierarchy, from which a wide variety of hierarchical approaches can be derived.

6.2 Multilevel Methods for Reinforcement Learning

We now come to the application of multilevel methods for RL. We consider the Bellman equation in the form (3.15). Let us now define (leaving out the policy notation)

$$A = I - \gamma P^{\pi} \tag{6.9}$$

$$b = r^{\pi} \tag{6.10}$$

so that Eq. (3.15) takes the standard form (6.4):

$$Av = b.$$

Now Ziv investigated the use of classic iteration methods such as the Richardson method for the solution of (6.4) with the operator (6.9) and the right-hand side (6.10). Using the eigenvalue properties of the transition probability matrix P^{π}, it was established by this means that algebraic smooth error vectors do arise. As was mentioned at the start, multilevel methods are outstandingly suitable for the solution of these problems. Of course everything we have described carries over to the case of action-value functions (3.17).

Furthermore. in [Ziv04] and [Pap10], the use of multilevel methods is investigated not only for the problems of dynamic programming (i.e., model based) but also for model-free methods, in particular for temporal-difference learning. For the latter, two multilevel methods are proposed, the first a multiplicative variant and the second an additive variant. The multiplicative variant is less convincing, and it cannot be fully proven to converge. In the following we will investigate the multigrid method for the model-based case and then move to Ziv's additive preconditioner for the model-free case.

6.2.1 Interpolation and Restriction Based on State Aggregation

We consider a hierarchy with the levels 0, . . ., L, where 0 is the finest grid which represents the current state values from $\boldsymbol{S}$ and L the coarsest grid. An interpolator I_{l+1}^{l} from level l+1 to level l shall be given. For the restriction I_{l}^{l+1} of level l+1 to the level l in general, the transpose or pseudo-inverse of I_{l+1}^{l} is used.

In [Ziv04] and [Pap10] the construction of the interpolator I_{l+1}^{l} was considered on an algebraic basis in the course of the algebraic multigrid. In particular the *state aggregation* was considered according to [BC89] that we will also apply. For this purpose the state space $\boldsymbol{S}$ is split into K disjoint groups G_{β}, $\beta = 1, \quad \ldots, \quad m$. The definition of the interpolator is

$$\left(I_{l+1}^{l}\right)_{i\beta} = \begin{cases} 1, & i \in G_{\beta} \\ 0, & \text{else} \end{cases} \tag{6.11}$$

and the restrictor is defined as its pseudo-inverse according to Moore-Penrose:

$$I_{l}^{l+1} = \left[\left(I_{l+1}^{l}\right)^{T} I_{l+1}^{l}\right]^{-1} \left(I_{l+1}^{l}\right)^{T}. \tag{6.12}$$

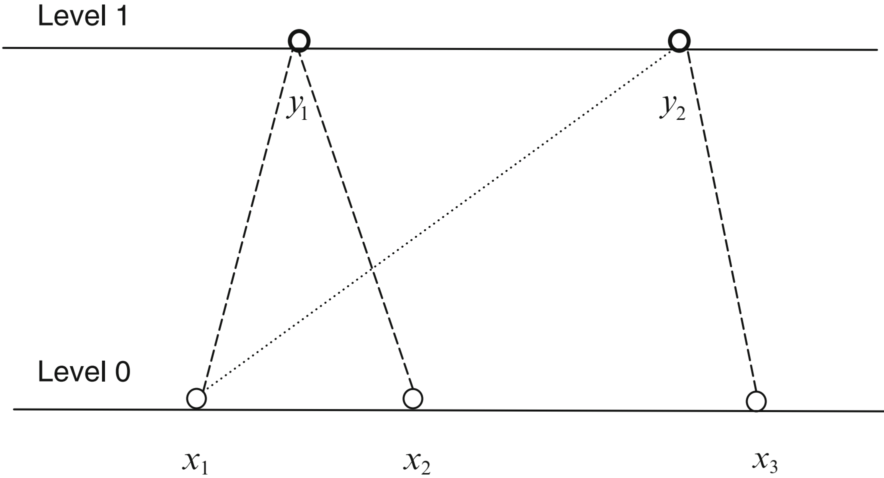


Fig. 6.5 Interpolation operator for state aggregations. The dashed line would remove the uniqueness of the assignment and is not permitted

Because of the simple structure of the interpolator (6.11), the Moore-Penrose pseudo-inverse simplifies to:

$$I_l^{l+1} = \left(|G_\beta|^{-1}\right)_{\beta \in \underline{m}} \left(I_{l+1}^l\right)^T, \tag{6.13}$$

that is, it equals to the transpose of the interpolator weighted by the reciprocal numbers of the states that merge in the corresponding aggregate.

Example 6.2 The state aggregation shall be illustrated by means of a simple example. For this let us consider an RE which delivers recommendations for just 3 products. We now obtain a level hierarchy with 3 nodes on the fine grid and 2 nodes on the coarse grid (Fig. 6.5).

The states are designated on the fine grid as x and on the coarse grid as y. This gives us the interpolator I_1^0 and restrictor I_0^1 as follows:

$$I_1^0 y = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}, \quad I_0^1 x = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

Obviously we can rewrite the restrictor I_0^1 in the simple form (6.13):

$$I_0^1 = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \left(|G_\beta|^{-1}\right)_{\beta \in \{1,2\}} \left(I_1^0\right)^T.$$

■

In [Ziv04] and [Pap10] the specification of the aggregate groups G_β was performed using AMG methods in combination with RL-specific extensions.

In contrast to this general algebraic case, in most cases product master data is available for recommendation engines, including their assignment to categories. This should be used for the construction of the hierarchies, since it contains important additional information. Thus, there are two sources for specification of hierarchies for REs:

1. Product hierarchies such as shop taxonomy or product groups
2. Product attributes such as manufacturer, brand, or color

The use of 1. seems obvious; however, in most cases it requires comprehensive preprocessing so that the pseudo-inverse (6.12) can be calculated. In the case of 2., this is automatically guaranteed, since every product can be assigned only to one parent state. Notice that only two-grid hierarchies can be constructed from 2., but in most cases this is sufficient.

6.2.2 *The Model-Based Case: AMG*

In the following, we shall investigate and discuss the algebraic multigrid method for solving the Bellman equation (3.15) or (3.17), respectively. The approach was proposed by Bertsekas and Castanon in [BC89]. Without loss of generality, we shall hold back on the 2-level method, that is, $l = 0$ is the fine grid and $l = 1$ the coarse grid.

Let $L = I_1^0$ be the aggregation prolongator according to (6.11). Moreover, let R be the restriction operator given by

$$R := \left(L^T W L^{-1}\right)^{-1} L^T W, \quad W := \operatorname{diag}(w), \tag{6.14}$$

(of which (6.12) is the special case where $W = I$), where $w \in \Re^n$ is (component-wise) positive and sums to 1. The matrix R is also referred to as the *Moore-Penrose inverse of L with respect to the w-weighted inner product*.

We now consider the 2-level-mutligrid procedure according to Algorithm 6.1 with a single smoothing step, that is, $\nu_1 = \nu_2 = 1$, by means of the Richardson iteration

$$x := IT(x, \ y) := x + y - Ax.$$

As we carry out only one smoothing step and y is taken to be the residual $Ax - \hat{y}$, the smoothing simplifies to

$$x := x + (Ax - \hat{y}) - Ax = x - \hat{y}.$$

Of course, this does no longer hold for multiple smoothing steps. For two smoothing steps, we would obtain

$$x := (x - \hat{y}\,) + (Ax - \hat{y}\,) - A(x - \hat{y}\,) = x - 2\hat{y} + A\hat{y}\,.$$

The resulting multigrid method is summarized in Algorithm 6.2. Hereupon, the upper index *1* has been omitted for the sake of readability.

Algorithm 6.2: Multigrid V-cycle

Input: matrix A, right-hand side b, prolongator L, restrictor R, initial iterate $x := x^0 \in \Re^n$

Output: approximate solution $\widetilde{x} \in \Re^n$ of (6.4)

1: **procedure** VCYCLE(y)
2: $\quad x := x - y$ ▷ pre-smoothing
3: $\quad y^1 := R(Ax - y)$ ▷ computing the residual
4: $\quad x^1 := (A^1)^{-1} y^1$ ▷ direct solver on the coarse grid
5: $\quad x := x + Lx^1$ ▷ coarse grid correction
6: $\quad x := x - y$ ▷ post-smoothing
7: $\quad$ return x
8: **end procedure**
9: **return** VCYCLE(b) ▷ initial call

The following rather technical convergence result has been established in [Pap10].

Theorem 6.1 (Theorem 3.7.1 in [Pap11]) *Let*

$$\widetilde{P} \in \Re^{n \times n}, \;\; \widetilde{p}_{ij} := \begin{cases} \alpha q_j^{(\beta)}, & i,j \in G_\beta \\ 0, & \text{else} \end{cases} \tag{6.15}$$

for $0 \leq \alpha \leq 1$, *and*

$$q_j^{(\beta)} \geq 0, \;\; j \in G_\beta, \;\; \sum_{j \in G_\beta} q_j^{(\beta)} = 1, \;\; \beta \in \underline{m}.$$

Moreover, we define

$$\begin{aligned} &A := I - \gamma P, \\ &Q := I - A, \\ &K := I - L(RAL)^{-1} RA, \\ &\varepsilon := \left\|P - \widetilde{P}\right\|_\infty^*, \\ &\hat{\varepsilon} := \|\alpha I - RPL\|_\infty^*, \textit{and} \\ &c := \min_{\beta,j} q_j^{(\beta)}. \end{aligned}$$

Let $\hat{\varepsilon} < \gamma^{-1}(1-\alpha\gamma)$. Then the asymptotic convergence rate of Algorithm 6.2 satisfies

$$\rho(KQ) \leq \|KQ\|_\infty^* \leq \frac{2(1-\alpha\gamma c)\gamma + \gamma^2}{1-\alpha\gamma}\varepsilon + \underbrace{\frac{\gamma^2\|A\|_\infty^*\hat{\varepsilon}}{\big(1-\gamma(\alpha+\hat{\varepsilon})\big)\big(1-\alpha\gamma\big)}}_{=:\delta}.$$

If the stronger condition $\varepsilon < \gamma^{-1}(1-\alpha\gamma)$ holds, we obtain

$$\delta \leq \frac{\gamma^2\|A\|_\infty^*\varepsilon}{\big(1-\gamma(\alpha+\varepsilon)\big)\big(1-\alpha\gamma\big)}.$$

Proof Let

$$\widetilde{A} := I - \gamma\widetilde{P}, \widetilde{K} := I - L\Big(R\widetilde{A}L\Big)^{-1}R\widetilde{A}, \widetilde{Q} := I - \widetilde{A}.$$

In the following, we shall establish an expression for KQ in terms of $\widetilde{K}$ and $\widetilde{Q}$. To this end, notice that we have for some $B \in \mathfrak{R}^{m\times m}$ and $\delta A \in \mathfrak{R}^{n\times n}$,

$$\begin{aligned} K &= I - L\Big(R\widetilde{A}L\Big)^{-1} + BRA \\ &= I - L\Big(R\widetilde{A}L\Big)^{-1}RA - \underbrace{LBR}_{=:\overline{B}}A \\ &= I - L\Big(R\widetilde{A}L\Big)^{-1}R\widetilde{A} - L\Big(R\widetilde{A}L\Big)^{-1}R\delta A - \overline{B}A \\ &= \widetilde{K}\underbrace{-11-\alpha\gamma LR\delta A - \overline{B}}_{=:\,\delta K}A. \end{aligned}$$

Moreover, we have

$$KQ = \underbrace{\widetilde{K}\widetilde{Q}}_{=O} + \widetilde{K}\delta Q + \delta KQ$$

where $\delta Q := Q - \widetilde{Q}$, and the identity $\widetilde{K}\widetilde{Q} = O$ follows from the fact that, as an immediate consequence of the definition of $\widetilde{P}$, $\widetilde{K}$ is an oblique projector along the range of $\widetilde{P}$ (cf. Lemma 3.2.7 in [Pap11]). Combining the above expressions, we obtain

$$KQ = \underbrace{\widetilde{K}\delta Q}_{=:\,E^{(1)}} \underbrace{-\frac{LR}{1-\alpha\gamma}\delta QQ}_{=:\,E^{(2)}} + \underbrace{\overline{B}AQ}_{=:\,E^{(3)}}.$$

The remainder of the proof is by bounding the additive terms on the right-hand side of the above equation separately:

$$\begin{aligned}\|E^{(1)}\|_\infty^* &\leq \left\| I - L\left(R\widetilde{A}L\right)^{-1} R\widetilde{A} \right\|_\infty^* \|\delta Q\|_\infty^* \\ &\leq \left(1 + \|L\|_\infty^* \left\| \left(R\widetilde{A}L\right)^{-1} \right\|_\infty^* \|R\|_\infty^* \left\|\widetilde{A}\right\|_\infty^* \right)\gamma\varepsilon \\ &= \left(1 + \frac{\|A\|_\infty^*}{1-\alpha\gamma}\right)\gamma\varepsilon.\end{aligned}$$

Moreover, row stochasticity of $\widetilde{P}$ implies that (cf. Eq. (3.2.4) in [Pap11])

$$\begin{aligned}\left\|I - \gamma\widetilde{P}\right\|_\infty^* &= \max_{i \in \underline{n}} \left(|1 - \gamma\widetilde{p}_{ii}| + \gamma \sum_{j \in \underline{n}\setminus i} \widetilde{p}_{ij} \right) \\ &= \max_{i \in \underline{n}} \left(|1 - \gamma\widetilde{p}_{ii}| + \gamma(1 - \widetilde{p}_{ii}) \right) \\ &= 1 + \gamma\left(1 - 2\min_{i \in \underline{n}} \widetilde{p}_{ii}\right),\end{aligned}$$

which, by definition of $\widetilde{A}$, gives rise to

$$\begin{aligned}\left\|\widetilde{A}\right\|_\infty^* &= \max_{k \in \underline{m}} \left\| I_{n_k} - \alpha\gamma\mathbf{1}\left(q^{(k)}\right)^T \right\|_\infty^* = 1 + (1-2c)\alpha\gamma \\ \|E^{(1)}\|_\infty^* &\leq \frac{2(1-\alpha\gamma c)}{1-\alpha\gamma}\gamma\varepsilon.\end{aligned}$$

As for $E^{(2)}$, we obtain

$$\|E^{(2)}\|_\infty^* \leq \frac{\|LR\|_\infty^*}{1-\alpha y} \|\delta Q\|_\infty^* \|Q\|_\infty^* = \frac{1}{1-\alpha\gamma}\gamma\varepsilon\gamma.$$

Finally, we bound $\|E^{(3)}\|_\infty^*$ as follows: the assumption that $\hat{\varepsilon} < \gamma^{-1}(1-\alpha\gamma)$ yields

$$\|R\delta AL\|_\infty^* \leq \|\delta A\|_\infty^* \leq \gamma\hat{\varepsilon} \leq 1 - \alpha\gamma = \frac{1}{\left\|\left(R\widetilde{A}L\right)^{-1}\right\|_\infty^*}.$$

Hence, RAL and $R\widetilde{A}L$ satisfy the hypothesis of Theorem 2.7.2 in [GVL96, pp. 80–86], and we may apply the perturbation provided therein to establish

$$
\begin{aligned}
\|B\|_\infty^* &\le \frac{\kappa_\infty\left(R\widetilde{A}L\right)\frac{\|\delta A\|_\infty^*}{\left\|R\widetilde{A}L\right\|_\infty^*}}{1-\kappa_\infty\left(R\widetilde{A}L\right)\frac{\|\delta A\|_\infty^*}{\left\|R\widetilde{A}L\right\|_\infty^*}}\left\|\left(R\widetilde{A}L\right)^{-1}\right\|_\infty^* \\
&= \frac{\frac{\|R\delta AL\|_\infty^*}{\left\|R\widetilde{A}L\right\|_\infty^*}}{1-\frac{\|R\delta AL\|_\infty^*}{\left\|R\widetilde{A}L\right\|_\infty^*}}\left\|\left(R\widetilde{A}L\right)^{-1}\right\|_\infty^* \\
&= \frac{\|R\delta AL\|_\infty^*}{\left\|R\widetilde{A}L\right\|_\infty^*-\|R\delta AL\|_\infty^*}\frac{1}{1-\alpha\gamma} \\
&= \frac{\|R\delta AL\|_\infty^*}{1-\alpha\gamma-\|R\delta AL\|_\infty^*}\frac{1}{1-\alpha\gamma} \\
&\le \frac{\gamma\hat{\varepsilon}}{1-\alpha\gamma-\gamma\hat{\varepsilon}}\frac{1}{1-\alpha\gamma},
\end{aligned}
$$

where we invoked upon the identity

$$\kappa_\infty\left(R\widetilde{A}L\right) = \kappa_\infty((1-\alpha\gamma)I) = 1.$$

Hence, we obtain

$$\left\|E^{(3)}\right\|_\infty^* \le \|LBR\|_\infty^*\|A\|_\infty^*\|Q\|_\infty^* \le \frac{\gamma^2\hat{\varepsilon}}{\left(1-(\alpha+\hat{\varepsilon})\gamma\right)\left(1-\gamma\right)}\|A\|_\infty^*.$$

If the stronger condition $\varepsilon < \gamma^{-1}(1-\alpha\gamma)$ holds, we attain the stronger bound

$$\left\|E^{(3)}\right\|_\infty^* \le \frac{\gamma^2\varepsilon}{\left(1-(\alpha+\varepsilon)\gamma\right)\left(1-\gamma\right)}\|A\|_\infty^*$$

from essentially the same calculation. □

The message of this result is as follows: the closer the transition probability matrix is to a block-diagonal matrix composed of rank-1-blocks corresponding to the classes of the partition, the faster converges the AMG procedure. With regard to recommendation engines, we may conclude that the method is sensibly applicable if a major part of the transitions is between products in the same class. Furthermore, the behavior within the classes should be almost memoryless, that is, the transition to a product within a class hardly depends on the previous product.

Example 6.3 For a shop with 9 products ($n = 9$) and using 3 partitions ($\beta = 3$) with 2, 4, and 3 products, respectively, the matrix $\widetilde{P}$ has the following structure:

$$\widetilde{P} = \begin{bmatrix} \begin{bmatrix} a & a \\ b & b \end{bmatrix} & 0 & 0 \\ 0 & \begin{bmatrix} c & c & c & c \\ d & d & d & d \\ e & e & e & e \\ f & f & f & f \end{bmatrix} & 0 \\ 0 & 0 & \begin{bmatrix} g & g & g \\ h & h & h \\ i & i & i \end{bmatrix} \end{bmatrix}. \quad \blacksquare$$

Obviously, the first assumption is realistic, since, in a typical shop, most transitions take place within product categories, provided that the latter have been chosen sensibly. The second assumption, however, is violated in most practical situations.

An alternative approach with favorable convergence properties in more general situations is *iterative aggregation-disaggregation (IAD)*. This method is obtained by replacing the Richardson sweep by a so-called additive algebraic Schwarz sweep. Having its origins in the field of partial differential equations, the latter is a domain decomposition-based procedure which can also be applied to systems of linear equations of the form (6.4). In this context, one speaks of *algebraic Schwarz methods*. We shall now provide a brief outline of the algebraic Schwarz sweep (details may be found in [Pap10]).

For each set in the partition, we restrict the residual and the corresponding matrix coefficients to the indices therein and add the solution of the thus obtained system to the corresponding entries of the current iterate. It may easily be verified that the algebraic Schwarz sweep yields the exact solution in case of a block-diagonal system. Due to continuity, we may conclude that – if applied in an iterative fashion – the method exhibits swift convergence if the system is almost block-diagonal.

From a practical point of view, this complies with the above-described situation where the major part of the state transitions takes place within the sets of the partition. Hence, we may expect the method to exhibit rapid convergence in a larger class of practical situations even without a coarse grid correction. Sadly enough, practical experience in [Pap10] reveal that even in cases where the second condition of the above is not satisfied, the aggregation method with a Richardson step outperforms the IAD method with respect to computation time, although the number of iterations is larger by 1 up to 2 orders of magnitude. This is due to the fairly greater computational intensity of the algebraic Schwarz sweep, which requires the solution of a multitude of smaller systems of linear equations, whereas the Richardson step consists of only one matrix–vector multiplication.

6.2.3 Model-Free Case: TD with Additive Preconditioner

We would now like to deploy the multilevel approach to accelerate the TD (λ)-method. To this end, we invoke the approach of a so-called preconditioner, which we shall present in the following.

We consider the inter-level operators I_{l+1}^{l}, I_{l}^{l+1} as described in Sect. 6.2.1. Moreover, let I_l^m be the prolongator from level l to level m defined as

$$I_l^m = I_{m-1}^m I_{m-2}^{m-1} \ldots I_l^{l-1}, \tag{6.16}$$

and I_l^l is stipulated to be the identity matrix. Let a preconditioner C_t^{-1} be given by

$$C_t^{-1} = \sum_{l=0}^{L} \beta_{l,t} I_l^0 I_0^l. \tag{6.17}$$

It is summarized in Algorithm 6.3.

Algorithm 6.3: BPX preconditioner (Ziv)

Input: residual y, number of grids L, interpolator I_{l+1}^{l}, restrictor I_l^{l+1}, coefficient vectors β_l

Output: new guess $x \in \Re^n$

1: **procedure** BPX(y)
2: $x^0 := y$
3: **for** $k = 1, \ldots, L$ **do**
4: $x^l := I_{l-1}^{l}\, x^{l-1}$ ▷ restriction
5: **end for**
6: **for** $k = 0, \ldots, L$ **do**
7: $x^l := \beta_l x^l$ ▷ scaling
8: **end for**
9: **for** $k = L, \ldots, 1$ **do**
10: $x^{l-1} := I_l^{l-1}\, x^l$ ▷ interpolation
11: **end for**
12: **return** x^0
13: **end procedure**

The preconditioner of Ziv (6.17) can be viewed as an algebraic counterpart to the BPX preconditioner [BPX90] well known in numerical analysis.

Then the preconditioned TD(λ)-method (for simplicity we avoid the iteration indexes)

$$w := w + \alpha_t C_t^{-1} z_t d_t \tag{6.18}$$

converges almost surely to the same solution as the TD(λ)-method (3.20).

The proof of convergence is essentially based on the subsequent theorem from [Ziv04], a further generalization of which has been devised in [Pap10].

Theorem 6.2 *Let the prerequisites for convergence of TD(λ) of Theorem 3.2 be satisfied. Moreover, let B^{-1} be a symmetric and positive definite (spd) $N \times N$-matrix. Then the preconditioned TD(λ)-method*

$$w := w + \alpha_t B^{-1} z_t d_t \tag{6.19}$$

converges as well.

Since C_t^{-1} is an spd $N \times N$-matrix, the hierarchically preconditioned TD(λ) converges.

The preconditioned TD(λ) algorithm (6.18), however, operates in terms of action values. Hence, the inter-level operators are needed for state-action pairs (s, a) rather than for single states. Yet how can we define hierarchies of actions? Since in the recommendation approach the spaces $\boldsymbol{S}$ and $\boldsymbol{A}$ are isomorphic (4.1), actions may be treated in the same way as states, and the same inter-level operators may be used for the former.

While the states in $\boldsymbol{S}$ correspond to products that are endowed with recommendations, the actions $\boldsymbol{A}$ correspond to the recommended products. Thus, similarly to (6.11), the following definition of the prolongator $\hat{I}_{l+1}^{l}$ suggests itself:

$$\left(\hat{I}_{l+1}^{l}\right)_{ij\beta\gamma} = \begin{cases} 1, & i \in G_\beta \wedge j \in H_\gamma(i) \\ 0, & \text{else.} \end{cases} \tag{6.20}$$

Here, the aggregations $H_\gamma(i)$, $\gamma = 1, \ldots, m_i$ refer to $A(s_i)$, that is, all actions executable in state s_i. Thus, the prolongation matrix is a block-diagonal matrix, where the blocks correspond to states and the block values to the actions.

Subsequently, we shall address a modification of the prolongator $\hat{I}_{l+1}^{l}$. This *weighted* prolongator is defined as follows:

$$\left(\overline{\hat{I}}_{l+1}^{l}\right)_{ij\beta\gamma} = \begin{cases} |A(s_i)|\,\left|A(a_j)\right|, & i \in G_\beta \wedge j \in H_\gamma(i) \\ 0, & \text{else} \end{cases}. \tag{6.21}$$

Here, $|A(s_i)|$ denotes the number of all actions in state s_i, that is, all rules for the corresponding product, and $|A(a_j)|$ the number of actions in the state associated with a_j, that is, the rules with the associated product for a conclusion. This weighted prolongator thus prefers rules with "strong" prerequisite or subsequent products, respectively.

In general, one can derive multiple hierarchies from the product specifications, for example, by means of shop hierarchies, commodity groups, and product attributes. Consequently, a corresponding preconditioner $\hat{C}_i^{-1}$ can be derived for each hierarchy according to (6.17). This gives rise to the question of whether preconditioners can also be applied in a combined fashion. Indeed, this is possible, for example, with respect to the preconditioner $\hat{C}_a^{-1}$:

$$\hat{C}_a^{-1} = \hat{C}_1^{-1} + \hat{C}_2^{-1} + \ldots + \hat{C}_n^{-1}, \tag{6.22}$$

where n denotes the number of all used hierarchies. Since all of the preconditioners $\hat{C}_i^{-1}$ are spd, so is $\hat{C}_a^{-1}$, and convergence of preconditioned TD(λ) follows from Theorem 6.2.

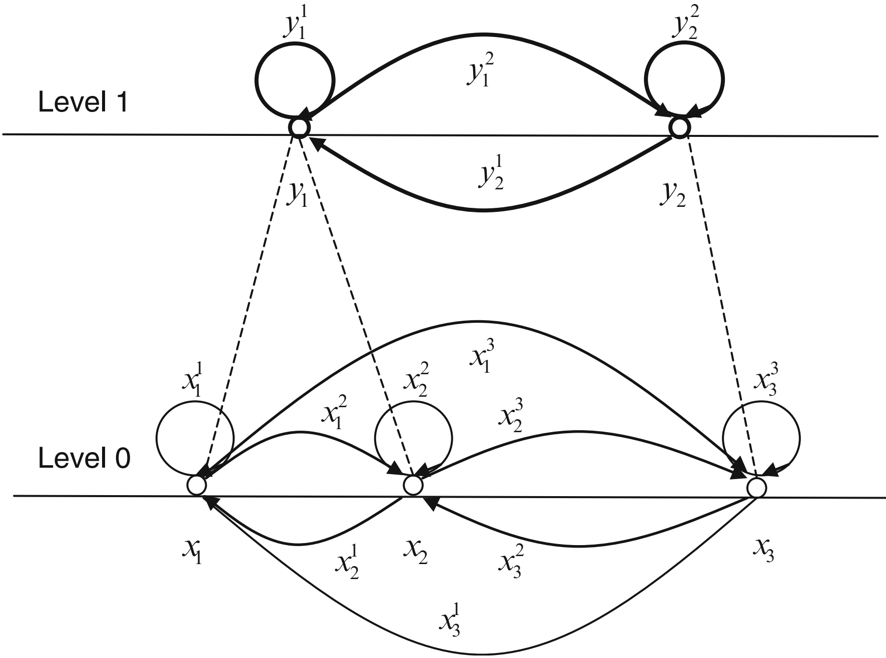


Fig. 6.6 Interpolation operator for state-action aggregations

Example 6.4 We consider the state space of Example 6.1 with the corresponding iterator I_1^0 and restrictor I_0^1. This gives the following preconditioner for the state-value function:

$$C^{-1}x = \begin{bmatrix} \widetilde{x}_1 \\ \widetilde{x}_2 \\ \widetilde{x}_3 \end{bmatrix} = \begin{bmatrix} 1+\frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 1+\frac{1}{2} & 0 \\ 0 & 0 & 1+1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

The example of the state aggregations should now be extended to include the associated actions, where initially all states are permissible as actions in all states. The result is shown in Fig. 6.6. For ease of reading, the actions are shown as x at level 0 and y at level 1, where the lower index represents the start nodes and the upper index the target nodes. For instance, x_1^2 is the recommendation of product 2 for product 1 at level 0.

Note that on the finest grid – that is, the product level – the reflexive relation x_i^i is practically meaningless, since a product cannot recommend itself. At levels > 0 these actions are meaningful, however, since they are a measure of the strength of product recommendations within the same group relative to one another.

The following interpolation and restriction matrix applies to the example under consideration:

$$\hat{I}_1^0\hat{y} = \begin{bmatrix} \begin{bmatrix} x_1^1 \\ x_1^2 \\ x_1^3 \end{bmatrix} \\ \begin{bmatrix} x_2^1 \\ x_2^2 \\ x_2^3 \end{bmatrix} \\ \begin{bmatrix} x_3^1 \\ x_3^2 \\ x_3^3 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} & 0 \\ \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} & 0 \\ 0 & \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \end{bmatrix} \begin{bmatrix} \begin{bmatrix} y_1^1 \\ y_1^2 \end{bmatrix} \\ \begin{bmatrix} y_2^1 \\ y_2^2 \end{bmatrix} \end{bmatrix}, \hat{I}_0^1\hat{x} = \begin{bmatrix} \begin{bmatrix} y_1^1 \\ y_1^2 \end{bmatrix} \\ \begin{bmatrix} y_2^1 \\ y_2^2 \end{bmatrix} \end{bmatrix}$$

$$= \begin{bmatrix} \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & 0 \\ 0 & 0 & \frac{1}{2} \end{bmatrix} & \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & 0 \\ 0 & 0 & \frac{1}{2} \end{bmatrix} & 0 \\ 0 \quad 0 & \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} & \end{bmatrix} \begin{bmatrix} \begin{bmatrix} x_1^1 \\ x_1^2 \\ x_1^3 \end{bmatrix} \\ \begin{bmatrix} x_2^1 \\ x_2^2 \\ x_2^3 \end{bmatrix} \\ \begin{bmatrix} x_3^1 \\ x_3^2 \\ x_3^3 \end{bmatrix} \end{bmatrix},$$

where the preconditioner $\hat{C}^{-1}$ is derived as follows:

$$\hat{C}^{-1}\hat{x} = \begin{bmatrix} \begin{bmatrix} \widetilde{x}_1^1 \\ \widetilde{x}_1^2 \\ \widetilde{x}_1^3 \end{bmatrix} \\ \begin{bmatrix} \widetilde{x}_2^1 \\ \widetilde{x}_2^2 \\ \widetilde{x}_2^3 \end{bmatrix} \\ \begin{bmatrix} \widetilde{x}_3^1 \\ \widetilde{x}_3^2 \\ \widetilde{x}_3^3 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} 1+\frac{1}{4} & \frac{1}{4} & 0 \\ \frac{1}{4} & 1+\frac{1}{4} & 0 \\ 0 & 0 & 1+\frac{1}{2} \end{bmatrix} & \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 \\ 0 & 0 & \frac{1}{2} \end{bmatrix} & \begin{matrix} 0 \\ 0 \end{matrix} \\ \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 \\ 0 & 0 & \frac{1}{2} \end{bmatrix} & \begin{bmatrix} 1+\frac{1}{4} & \frac{1}{4} & 0 \\ \frac{1}{4} & 1+\frac{1}{4} & 0 \\ 0 & 0 & 1+\frac{1}{2} \end{bmatrix} & \begin{bmatrix} 1+\frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 1+\frac{1}{2} & 0 \\ 0 & 0 & 1+1 \end{bmatrix} \\ 0 & 0 & \end{bmatrix} \begin{bmatrix} \begin{bmatrix} x_1^1 \\ x_1^2 \\ x_1^3 \end{bmatrix} \\ \begin{bmatrix} x_2^1 \\ x_2^2 \\ x_2^3 \end{bmatrix} \\ \begin{bmatrix} x_3^1 \\ x_3^2 \\ x_3^3 \end{bmatrix} \end{bmatrix}.$$

For simplicity's sake all scaling factors $\beta_{l,t}$ were set to a constant 1. We should now take a brief look at the method of operation of the preconditioner. Thus, an update via the action x_1^2 leads to an update of the actions x_1^2 themselves and also x_2^1 in accordance with:

$$\widetilde{x}_1^2 = \left(1+\frac{1}{4}\right)x_1^2, \ \ \widetilde{x}_2^1 = \frac{1}{4}x_1^2.$$

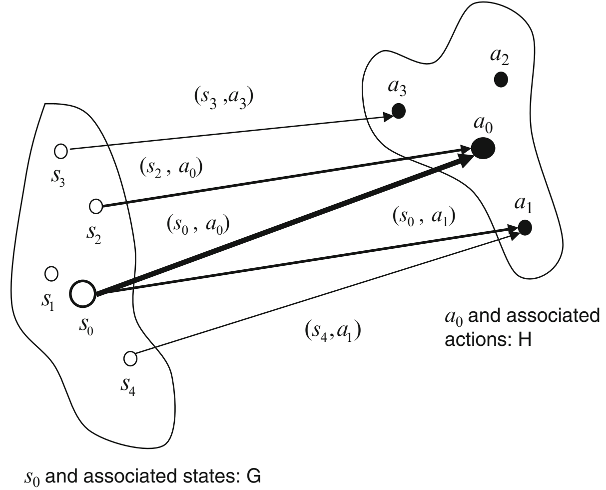


Fig. 6.7 Illustration of the update logic of the preconditioner $\hat{C}^{-1}$

This results from the reflexive coarse grid action y_1^1 of the group y_1 on itself. Of course the reflexive update for x_1^2 is especially strong here.

An update via the action x_1^3 leads to an update of the actions x_1^3 themselves and also x_2^3:

$$\widetilde{x}_1^3 = \left(1 + \frac{1}{2}\right)x_1^3, \quad \widetilde{x}_2^3 = \frac{1}{2}x_1^3.$$

It is the coarse grid action y_1^2 of the group y_1 on y_2 that is responsible for this. ■

Figure 6.7 illustrates the general logic of the updates using the example of the action (s_0, a_0).

An update of the rule (s_0, a_0) therefore leads not only to the update of the rule itself but also to the update of all rules in the same state group G of the initial product s_0 into the same action group H of the recommended product a_0.

From a technical point of view, there is another positive aspect: when the preconditioner $\hat{C}^{-1}$ updates an action value for the state-action pair (s, a), even though for (s, a) still no rule exists, it can be generated automatically. In this way the hierarchical preconditioner automatically generates new recommendations for products without recommendations (due to a lack or too little transaction history). We will also stress the subject into the next section.

6.3 Learning on Category Level

So far we have considered the hierarchical RL only under the aspect of the acceleration of convergence. However, as we mentioned at the end of the last section, it can also be used for a further task: raising the recommendation coverage.

Let us start with the information from Ziv [Ziv04]. A matrix A_l possesses a *Markov chain (MC) interpretation* if it can be written as

$$A_l = I_l - \gamma P_l \tag{6.23}$$

where I_l is the unit matrix of level l and P_l the matrix of the associated transition probabilities. Furthermore, we declare that the MC interpretation *is retained* under the precondition that A_l possesses an MC interpretation and the Galerkin operator $A_{l+1} = I_l^{l+1} A_l I_{l+1}^N$ also possesses an MC interpretation. Providing the MC interpretation is retained, the coarse grid problem can also be considered as an MDP on a reduced state space. This permits the policy search at a reduced resolution$_{j=1}^N$.

Now Ziv was able to prove that the state aggregations (6.11) and (6.12) retain the MC interpretation. The proof can be converted directly to our state-action aggregation (6.20). Alternatively we can also apply the learning simultaneously on the coarse grid $l + 1$ and obtain A_{l+1} directly. In this way everything that is described in Chaps. 4 and 5 is directly transferred to the category level. In particular we obtain explicitly in addition to the product rules $s \rightarrow s'$ the associated category rules $c_l \rightarrow c_l{}'$, which can also be saved in this form.

We can use the category rules to generate recommendations for products for which there is insufficient statistical mass. This follows the logic of the introduction of statistical, that is, averaged, characteristics: having convinced ourselves of the impossibility of obtaining good forecasts for the variable Y, we proceed from the failed prediction $C \rightarrow Y$ (C are the conditions) to the coarser prediction $C \rightarrow S_Y$ with a statistical characteristic S_Y. This stems from the natural pragmatic idea of "predict anything rather than nothing at all," in the hope that the prediction $C \rightarrow S_Y$ will prove stable and also in a certain sense useful.

So if we cannot generate stable product rules (or too few of them) $s \rightarrow s'$ for the product s (for instance, because s is a new product or so far has scarcely been visited), we move across to its parent category or categories c_l^s and use the strongest of those rules $c_l^s \rightarrow c'_l$. After this we break the recommended categories c'_l down into their products, for instance, by selecting the category top sellers, thereby obtaining our recommendations $s \rightarrow s'$. Generally the quality of these category recommendations is lower than of the pure product recommendations, but the conclusion stated above applies: most of the category recommendations are quite good, and they permit us to generate recommendations for virtually all long-tail products.

The use of these hierarchical recommendations together with the hierarchical preconditioners described above has been supported, for example, by the prudsys RDE for years and has proved to be very successful both for small web shops and also for shops that carry a vast range of products. In addition the use of hierarchical policies can be extended significantly, and this is the subject of current research [The12].

6.4 Summary

As we have seen so far, hierarchical methods are extremely important in many scientific areas. We have described that especially multilevel methods, which originate from numerical analysis, are in principle well suited to speed up the reinforcement learning. However, their application to RL is quite difficult. We used some specifics of recommendation engines to make the multilevel approach more applicable in this field. However, more research is required here.

We now come to the outlook for the future. We have the choice between the use of predefined hierarchies (as in the analytical case) or automatically generated hierarchies (coarsening, as in the algebraic case). Currently we are working with predefined hierarchies. Now these are hierarchies such as shop taxonomies or product groups specified by the shop or category manager and not primarily intended for the use in hierarchical preconditioners. Thus, they do not prove optimal for this. (In practice, they are usually subjected to a comprehensive preprocessing.) However, classic coarsening is also not the best possible method either, since of course the existing category information should be exploited for the hierarchies. It is therefore useful to employ a combination of the coarsening procedure, having regard to the product attributes and hierarchies. This work is in progress.

Chapter 7
Breaking Dimensions: Adaptive Scoring with Sparse Grids

Abstract We introduce the concept of a sparse grid and show how this powerful approach to function space discretization may be employed to tackle high-dimensional machine learning problems of regression and classification. In particular, we address the issue of incremental computation of sparse grid regression coefficients so as to meet the requirements of realtime data mining. Conclusively, we present experimental results on real-world data sets.

7.1 Introduction

In this chapter, we shall use hierarchical methods as introduced in the previous chapter to devise a powerful method for scoring – sparse grids. We first demonstrate how scoring can be used for calculating high-quality recommendations, although only for a limited number of products. Then we will introduce the sparse grid method and develop an adaptive sparse grid version. Finally, we present some numerical results.

We follow the approach of the papers [GG01a, GGT01] which describe sparse grids for classification. We develop an incremental sparse grid approach, i.e., incremental learning from new data points. Although sparse grids are quite complex, it turns out that extending them for this type of adaptivity is straightforward. This chapter addresses the mathematical inclined reader and requires solid knowledge of numerical analysis. Since it is not directly connected to the following chapters, it can also be skipped.

We start by mentioning that the term "scoring" is very general. Throughout this chapter we identify scoring with supervised learning which represents the common scoring domain.

In supervised learning we consider the given set of already classified data (training set)

$$S = \left\{ (\mathbf{x}_i, y_i) \in \Re^d \times \Re \right\}_{i=1}^{M},$$

A. Paprotny and M. Thess, *Realtime Data Mining: Self-Learning Techniques for Recommendation Engines*, Applied and Numerical Harmonic Analysis,
DOI 10.1007/978-3-319-01321-3_7, © Springer International Publishing Switzerland 2013

where $\mathbf{x}_i$ represents the data points in the attribute space and y_i the target attribute. Assume now that these data have been obtained by sampling of an unknown function f which belongs to some function space V defined over $\mathfrak{R}^d$. The sampling process was disturbed by noise. The aim is now to recover the function f from the given data as faithfully as possible. We distinguish between *classification*, where the target values y_i are from a discrete set of classes, e.g., from $\{-1, +1\}$ for binary classification, and *regression* where y_i are from a continuous spectrum. In what follows we mainly focus on classification having in mind that sparse grids can be used for regression, too [Gar06, Gar11]. In classification the function f is also called *classifier*.

Scoring is increasingly used for personalization and may also be applied to recommendations. An advantage of scoring is that we can include many attributes characterizing the user behavior in $\mathbf{x}_i$. This may be user-centric attributes like age and gender, transactional attributes like number of clicks or revenue, and many other attribute types like time, channel, or even weather. The disadvantage of scoring is the limited number of single attribute values it can handle in general. This renders a *direct* application of scoring for recommendations of many products virtually impossible.

There are different approaches to scoring-based recommendations. The most simple is to use the recommendations as target attribute, i.e., each recommended product corresponds to a target class. A more sophisticated approach is to use the success of the session (revenue or in case of classification indicator of orders in the session) as target attribute and the recommendation as a special set of *control* attributes. Thus, in each recommendation step, we select the control attributes to maximize $f(\mathbf{x})$. (Note that depending on the function class of the classifier f, this may result in a complex optimization problem. But this is not the main task of scoring and hence will not be considered here.)

Example 7.1 Consider a small web shop. Suppose we need to select one of the three on-site banners at each category and product page. Therefore, the banners represent the control attribute and thus the recommendations. We further assume that in each step of the session (product or category page view), a user is characterized by four attributes: age, gender, number of clicks in current session, and how many products are already in her/his basket. The target attribute is 0 if no order was placed within the session and 1 if something was ordered.

Table 7.1 shows three sample sessions. In the first step of session *A*, the user is considered to be unknown and hence his/her user-specific attributes age and gender have missing values (represented by character "?"). In the first step, the banner *b1* was recommended to his/her. We know from history that he/she has bought nothing in this session, so the target attribute is 0 in all steps of the session. The second step is very similar to the first one except that banner *b3* was recommended. In the third step, he/she added a product to his/her basket. In the fourth step, he/she signed in to the shop and now his/her age and gender are considered to be known.

Session *B* represents a registered user, who was already recognized at the beginning of the session, e.g., by a cookie. This user finally placed an order.

Table 7.1 Example of training set of three banner recommendations for classification

ID attribute	Input attributes				Control attribute	Target attribute
Session	Age	Gender	Clicks	Basket	Banner	Ordered
A	?	?	1	0	b1	0
A	?	?	2	0	b3	0
A	?	?	3	1	b2	0
A	24	f	4	1	b2	0
B	19	m	1	0	b3	1
B	19	m	2	0	b2	1
B	19	m	3	1	b2	1
B	19	m	4	2	b3	1
C	?	?	1	0	b2	0

The third session C represents an unknown user who clicked only once and then left the shop.

Based on such historic data, by means of a classification technique, we now can construct a classifier f that assigns a target value y to each attribute vector $\mathbf{x} \in \Re^5$ (the four input attributes and the control attribute), i.e., $y = f(\mathbf{x})$. The higher the y, the higher is the probability that an order will be placed inside the session. Since the input attributes are fixed, we maximize f in each step with respect to the control attribute.

To illustrate this procedure, consider a session step of a 56-year-old woman who has already done 3 clicks and added two products to the basket, i.e., $\mathbf{x} = (56, f, 3, 2, x_c)$, where x_c represents the value of the control attribute. Let $f(\mathbf{x})$ be 0.6 for $x_c = b_1$, 0.34 for $x_c = b_2$, and 0.85 for $x_c = b_3$. Then we would select banner b_3 as recommendation.

Thus, in each step of the session, we use the current values of the input attributes and find the optimal value of the control attribute. The corresponding banner is recommended.

The construction of the classifier, i.e., the learning, is performed either offline on historic data stored in the form of Table 7.1 or online, after each session terminated (e.g., by a timeout mechanism). ■

Although Example 7.1 is very simple, it reveals the power of the scoring approach for recommendations. Unlike basket analysis or collaborative filtering (which will be studied in the next chapter), it considers the recommendation task as control problem taking into account the effect of recommendations. As mentioned before, it also allows to include many different attributes into prediction. Scoring is used to calculate banner recommendations, special offers during the checkout process, and even for personalized navigation. Of course, it also bears some disadvantages: the control-theoretic approach is very limited and so is the ability to handle large numbers of recommendation items.

There exist many algorithms for classification and regression. Widely used approaches are nearest neighbor methods, decision tree induction, rule learning, and memory-based reasoning. There are also classification methods based on

adaptive multivariate regression splines, neural networks, support vector machines, and regularization networks. Interestingly, the latter techniques can be interpreted in the framework of regularization networks [GJP95]. With these techniques, it is possible to treat quite high-dimensional problems, but the amount of data is limited due to complexity reasons. This situation is reversed in many practical applications such as those of recommendations presented here where the dimension of the resulting problem is moderate but the amount of data is usually huge. Thus, there is a strong need for methods which can be applied in this situation as well.

We will see that sparse grids can cope with the complexity of the problem, at least to some extent. Moreover, sparse grids are perfectly suited for data adaptivity and show many other advantages. They represent a very modern approach to scoring.

7.2 The Sparse Grid Approach

Classification of data can be interpreted as traditional scattered data approximation problem with certain additional regularization terms. In contrast to conventional scattered data approximation applications, we now encounter quite high-dimensional spaces. To this end, the approach of regularization networks [GJP95] gives a good framework. The approach allows a direct description of the most popular neural networks, and it also allows for an equivalent description of support vector machines and n-term approximation schemes [EPP00, Gir98].

We start with the scoring problem for a given data set S described at the beginning of this chapter. This is clearly an ill-posed problem since there are infinitely many solutions possible. To get a well-posed, uniquely solvable problem, we have to assume further knowledge on f. To this end, regularization theory [TA77, Wah90] imposes an additional smoothness constraint on the solution of the approximation problem, and the regularization network approach considers the variation problem

$$\min_{f \in V} R(f)$$

with

$$R(f) = \frac{1}{M} \sum_{i=1}^{M} C(f(\mathbf{x}_i), y_i) + \lambda \Phi(f). \tag{7.1}$$

Here, $C(.,.)$ denotes an error cost function which measures the interpolation error, and $\Phi(f)$ is a smoothness functional which must be well defined for $f \in V$. The first term enforces closeness of f to the data, the second term enforces smoothness of f, and the regularization parameter λ balances between these two terms. Typical examples are

$$C(x, y) = |x - y| \quad \text{or} \quad C(x, y) = (x - y)^2,$$

and

$$\Phi(f) = \|Pf\|_2^2 \quad \text{with} \quad Pf = \nabla f \quad \text{or} \quad Pf = \Delta f,$$

with ∇ denoting the gradient and Δ the Laplace operator. The value λ can be chosen according to cross-validation techniques or to some other principle. Note that we find exactly this type of formulation in the case $d = 2, 3$ in many scattered data approximation methods (see [ADT95, HL92]), where the regularization term is usually physically motivated.

Now, we assume that we have a basis of V given by $\{\varphi_j(\mathbf{x})\}_{j=1}^{\infty}$. Let also the constant function be in the span of the functions φ_j. We then can express a function $f \in V$ as

$$f(\mathbf{x}) = \sum_{j=1}^{\infty} \alpha_j \varphi_j(\mathbf{x})$$

with associated degrees of freedom α_j. In the case of a regularization term of the type

$$\Phi(f) = \sum_{j=1}^{\infty} \frac{\alpha_j^2}{\lambda_j}$$

where $\{\lambda_j\}_{j=1}^{\infty}$ is a decreasing positive sequence, it is easy to show that independent of the function C, the solution of the variational problem (7.1) has always the form

$$f(\mathbf{x}) = \sum_{j=1}^{M} \alpha_j K\left(\mathbf{x}, \mathbf{x}_j\right).$$

Here, K is the symmetric kernel function

$$K(\mathbf{x}, \mathbf{y}) = \sum_{j=1}^{\infty} \lambda_j \varphi_j(\mathbf{x}) \varphi_j(\mathbf{y})$$

which can be interpreted as the kernel of a *Reproducing Kernel Hilbert Space* (RKHS). In other words, if certain functions $K(\mathbf{x},\mathbf{x}_j)$ are used in an approximation scheme which are centered in the location of the data points $\mathbf{x}_j$, then the approximation solution is a finite series and involves only M terms. Many approximation schemes like radial basis functions, additive models, several types of neural networks, and support vector machines (SVMs) can be derived by a specific choice of the regularization operator (see [EPP00, GJP93, GJP95]).

7.2.1 Discretization

We take the ansatz (6.1) of Sect. 6.1.1 (using a slightly different notation) and restrict the problem to a finite-dimensional subspace $V_N \in V$. The function f is then replaced by

$$f_N(\mathbf{x}) = \sum_{j=1}^{N} \alpha_j \varphi_j(\mathbf{x}). \tag{7.2}$$

Here the ansatz functions $\{\varphi_j\}_{j=1}^N$ should span V_N and preferably should form a basis for V_N. The coefficients $\{\alpha_j\}_{j=1}^N$ denote the degrees of freedom. Note that the restriction to a suitably chosen finite-dimensional subspace involves some additional regularization (regularization by discretization) which depends on the choice of V_N.

In the remainder of this chapter, we restrict ourselves to the choice

$$C(f_N(\mathbf{x}_i), y_i) = (f_N(\mathbf{x}_i) - y_i)^2$$

and

$$\Phi(f_N) = \|Pf_N\|_{L_2}^2 \tag{7.3}$$

for some given linear operator P. This way we obtain from the minimization problem a feasible linear system. We thus have to minimize

$$R(f_N) = \frac{1}{M} \sum_{i=1}^{M} (f_N(\mathbf{x}_i), y_i)^2 + \lambda \|Pf_N\|_{L_2}^2, \tag{7.4}$$

with f_N in the finite-dimensional space V_N. We plug (7.2) into (7.4) and obtain after differentiation with respect to α_k, $k = 1, \ldots, N$

$$0 = \frac{\partial R(f_N)}{\partial \alpha_k} = \frac{2}{M} \sum_{i=1}^{M} \left(\sum_{j=1}^{N} \alpha_j \varphi_j(\mathbf{x}_i) - y_i \right) \cdot \varphi_k(\mathbf{x}_i) + 2\lambda \sum_{j=1}^{N} \alpha_j \left(P\varphi_j, P\varphi_k\right)_{L_2}. \tag{7.5}$$

This is equivalent to $k = 1, \ldots, N$

$$\sum_{j=1}^{N} \alpha_j \left[M\lambda \left(P\varphi_j, P\varphi_k\right)_{L_2} + \sum_{i=1}^{M} y_i \varphi_j(\mathbf{x}_i) \cdot \varphi_k(\mathbf{x}_i) \right] = \sum_{i=1}^{M} y_i \varphi_k(\mathbf{x}_i). \tag{7.6}$$

In matrix notation we end up with the linear system

$$\left(\lambda C + B \cdot B^T\right) \alpha = By. \tag{7.7}$$

Here C is the $N \times N$ matrix with the entries $C_{j,k} = M \cdot \left(P\varphi_j, P\varphi_k\right)_{L_2}$, $j, k = 1, \ldots, N$, and B is a rectangular $N \times M$ matrix with entries $B_{j,i} = \varphi_j(\mathbf{x}_i)$, $i = 1, \ldots, M$, $j = 1, \ldots, N$. The vector y contains the data labels y_i and has length M. The unknown vector α contains the degrees of freedom α_j and has length N.

Depending on the regularization operator, we obtain different minimization problems in V_N. For example, if we use the gradient $\Phi(f_N) = \|\nabla f_N\|_{L_2}^2$ in the regularization expression in (7.1), we obtain a Poisson problem with an additional term which resembles the interpolation problem. The natural boundary conditions for such a partial differential equation are Neumann conditions. The discretization (7.2) gives us then the linear system (7.7) where C corresponds to a discrete Laplacian. To obtain the classifier f_N, we now have to solve this system.

7.2.2 *Grid-Based Discrete Approximation*

Up to now we have not yet been specific what finite-dimensional subspace V_N and what type of basis functions $\{\varphi_j\}_{j=1}^N$ we want to use. In contrast to conventional data mining approaches like radial basis approaches or SVMs, which work with ansatz functions associated to data points, we now use a certain grid in the attribute space to determine the classifier with the help of these grid points. This is similar to the numerical treatment of partial differential equations.

For reasons of simplicity, here and in the reminder of this chapter, we restrict ourselves to the case $\mathbf{x}_i = [0,1]^d$. This situation can always be reached by proper rescaling of the data space. A conventional finite element discretization would now employ an equidistant grid Ω_n with mesh size $h_n = 2^{-n}$ for each coordinate direction, where n is the refinement level. In the following we always use the gradient $P = \nabla$ in the regularization expression (7.3). Let $\mathbf{j}$ denote the multi-index $(j_1, \ldots, j_d) \in \mathbf{N}^d$. We now use piecewise d-linear, i.e., linear in each dimension, so-called hat functions (see also Fig. 6.2a) as test and trial functions $\varphi_{n,\mathbf{j}}(\mathbf{x})$ on grid Ω_n. Each basis function $\phi_{n,\mathbf{j}}(\mathbf{x})$ is thereby 1 at grid point $\mathbf{j}$ and 0 at all other points of grid Ω_n. A finite element method on grid Ω_n now would give

$$(f_N(\mathbf{x}) =) f_n(\mathbf{x}) = \sum_{j_1}^{2^n} \ldots \sum_{j_d}^{2^n} \alpha_{n,\mathbf{j}} \phi_{n,\mathbf{j}}(\mathbf{x}).$$

And the variational procedure (7.4), (7.5), and (7.6) would result in the discrete linear system

$$\left(\lambda C_n + B_n \cdot B_n^T\right) \alpha_n = B_n y \tag{7.8}$$

with the discrete $(2^n + 1)^d \times (2^n + 1)^d$ Laplacian

$$(C_n)_{\mathbf{j},\mathbf{k}} = M.\left(\nabla\phi_{n,\mathbf{j}}, \nabla\phi_{n,\mathbf{k}}\right),$$

j_1, $k_t = 0, \ldots, 2^n$, $t = 1, \ldots d$, the $(2^n + 1)^d \times M$ -matrix

$$(B_n)_{\mathbf{j},i} = \phi_{n,\mathbf{j}}(\mathbf{x}),$$

$j_t = 0, \ldots 2^{l_t}$, $t = 1, \ldots, d$, $i = 0, \ldots, M$, and the unknown vector $(\alpha_n)_j$, $j_t = 0, \ldots, 2^n$, $t = 1, \ldots, d$. Note that f_n lives in the space

$$V_n := \text{span}\left\{\phi_{n,\mathbf{j}}, j_t = 0, \ldots, 2^d, t = 1, \ldots, d\right\}.$$

The discrete problem (7.8) might in principle be treated by an appropriate solver like the conjugate gradient method, a multigrid method or some other suitable efficient iterative method. However, the direct application of a finite element discretization and the solution of the resulting linear system by an appropriate solver are clearly not possible for a d-dimensional problem if d is larger than four. The number of grid points is of the order $O(h_n^{-d}) = O(2^{nd})$, and in the best case, the number of operations is of the same order. Here we encounter the so-called curse of dimensionality: the complexity of the problem grows exponentially with d. At least for $d > 4$ and a reasonable value of n, the arising system cannot be stored and solved on even the largest parallel computers today.

7.2.3 Sparse Grid Space

However, there is a special discretization technique using so-called sparse grids which allow to cope with the complexity of the problem, at least to some extent. This method has been originally developed for the solution of partial differential equations [GSZ92, Zen91] especially by the group of Christoph Zenger and is now used successfully also for integral equations, interpolation and approximation, eigenvalue problems, and integration problems. In the information-based complexity community, it is also known as "hyperbolic cross points," and the idea can even be traced back to the Russian mathematician Alexey Smolyak [Smo63]. The application of sparse grids for classification and regression, described here, is a result of a long-standing cooperation of the prudsys AG (Michael Thess) with the University of Bonn (Jochen Garcke and Michael Griebel); see [GGT01].

For a d-dimensional problem, the sparse grid approach employs only $O(h_n^{-1}(\log(h_n^{-1}))^{d-1})$ grid points in the discretization process. It can be shown that an accuracy of $O(h_n^2 \log(h_n^{-1})^{d-1})$ can be achieved pointwise or with respect to the L_2- or L_∞ -norm provided that the solution is sufficiently smooth. Thus, in comparison to conventional full grid methods, which need $O(h_n^{-d})$ for an accuracy of $O(h_n^2)$, the sparse grid method can be employed also for high-dimensional problems. The curse of dimensionality of full grid methods affects sparse grids much less.

Now, with the multi-index $\mathbf{l} = (l_1, \ldots, l_d) \in \mathbf{N}^d$, we consider the family of standard regular grids

$$\left\{\Omega_\mathbf{l}, \mathbf{l} \in \mathbf{N}^d\right\} \tag{7.9}$$

on $\overline{\Omega}$ with mesh size $h_\mathbf{l} := \left(j_{l_1}, \ldots, j_{l_d}\right) := \left(2^{-l_1}, \ldots, 2^{-l_d}\right)$. That is, $\Omega_\mathbf{l}$ is equidistant with respect to each coordinate direction, but, in general, has different mesh sizes in the different coordinate directions. The grid points contained in a grid $\Omega_\mathbf{l}$ are the points

$$x_{\mathbf{l},\mathbf{j}} := \left(x_{l_1,j}, \ldots, x_{l_d}\right) \tag{7.10}$$

with $h_\mathbf{l} := \left(h_{l_1}, \ldots, h_{l_d}\right) := \left(2^{-l_1}, \ldots, 2^{-l_d}\right)$. On each grid $\Omega_\mathbf{l}$, we define the space $V_\mathbf{1}$ of piecewise d-linear functions

$$V_l := \operatorname{span}\left\{\phi_{\mathbf{1},\mathbf{j}}, j_t = 0, \ldots, 2^{l_t}, t = 1, \ldots, d\right\} \tag{7.11}$$

which is spanned by the usual d-dimensional piecewise d-linear hat functions

$$\phi_{\mathbf{1},\mathbf{j}}(\mathbf{x}) := \prod_{t=1}^{d} \phi_{l_t,j_t}(x_t). \tag{7.12}$$

Here, the one-dimensional functions $\phi_{l_t,j_t}(x_i)$ with support $\left[x_{l_t,j_t} - h_{l_t}, x_{l_t,j_t} + h_{l_t}\right]$ $= \left[(j_t - 1)h_{l_t}, (j_t + 1)h_{l_t}\right]$ (e.g., restricted to [0,1]) can be created from a unique one-dimensional mother function $\phi(x)$

$$\phi(x) := \begin{cases} 1 - |x| & x \in]-1, 1[, \\ 0 & \text{otherwise} \end{cases} \tag{7.13}$$

by dilatation and translation, i.e.,

$$\phi_{l_t,j_t}(x_t) := \phi\left(\frac{x_t - j_t \cdot h_{l_t}}{h_{l_t}}\right). \tag{7.14}$$

In the previous definition and in the following, the multi-index $\mathbf{l} \in \mathbf{N}^d$ indicates the level of a grid or a space or a function, respectively, whereas the multi-index $\mathbf{j} \in \mathbf{N}^d$ denotes the location of a given grid point $x_{\mathbf{l},\mathbf{j}}$ or of the respective basis function $\phi_{\mathbf{1},\mathbf{j}}(\mathbf{x})$, respectively.

Now, we can define the difference spaces

$$W_\mathbf{1} = V_\mathbf{1} / \bigcup_{t=1}^{d} V_{\mathbf{1}-\mathbf{e}_t}, \tag{7.15}$$

where $\mathbf{e}_1$ denotes the tth unit vector. To complete this definition, we formally set $V_\mathbf{1} = 0$ if $l_t = -1$ for at least one $t \in \{1, \ldots, d\}$. In other words, $W_\mathbf{1}$ consists of

the functions in $V_{\mathbf{1}}$ which are not in any $V_{\mathbf{1}-\mathbf{e}_t}$. These hierarchical difference spaces allow us the definition of a multilevel subspace splitting, i.e., the definition of the space V_n as a direct sum of subspaces,

$$V_n := \bigoplus_{l_1}^{n} \cdots \bigoplus_{l_d}^{n} W_{(l_1,\ldots,l_d)} = \bigoplus_{|1|_\infty \le n} W_1. \tag{7.16}$$

Here and in the following, let $\le$ denote the corresponding element-wise relation, and let $|\mathbf{l}|_\infty := \max_{l \le l_t \le d} l_t$ and $\|\mathbf{l}\|_1 := \sum_{t=1}^{d} l_t$ denote the discrete L_∞- and the discrete L_1 -norm of $\mathbf{l}$, respectively. As it can be easily seen from (7.11) and (7.15), the introduction of index sets $\mathbf{I}_1$,

$$\mathbf{I_1} := \left\{ (j_1,\ldots,j_d) \in \mathbf{N}^d, \begin{cases} j_t = 1,\ \ldots, 2^{l_t} - 1,\ j_t \text{ odd}, t = 1,\ \ldots, d, & \text{if} \quad l_t > 0, \\ j_t = 0, t = 1,\ \ldots, d, & \text{if} \quad l_t = 0, \end{cases} \right\} \tag{7.17}$$

leads to

$$W_{\mathbf{l}} = \operatorname{span}\left\{ \phi_{\mathbf{1},\mathbf{j}}, \mathbf{j} \in \mathbf{I}_1 \right\}. \tag{7.18}$$

Therefore, the family of functions

$$\left\{ \phi_{\mathbf{1},\mathbf{j}}, \mathbf{j} \in \mathbf{I}_1 \right\}_{\mathbf{0}}^{\mathbf{n}} \tag{7.19}$$

is just a hierarchical basis [Fab9, Ys86] of V_n that generalizes the one-dimensional hierarchical basis of [Fab9] to the d-dimensional case by means of a tensor-product approach. Note that the support of all basis functions $\phi_{\mathbf{1},\mathbf{j}}(\mathbf{x})$ in (7.18) spanning $W_{\mathbf{1}}$ is mutually disjoint.

Now, any function $f \in V_n$ can be splitted accordingly by

$$f(\mathbf{x}) = \sum_{\mathbf{1} \le \mathbf{n}} f_{\mathbf{1}}(\mathbf{x}), f_{\mathbf{1}} \in W_{\mathbf{1}} = \phi_{\mathbf{1},\mathbf{j}}(\mathbf{x}), \quad \text{and} \quad f_{\mathbf{1}}(\mathbf{x}) = \sum_{\mathbf{j} \in \mathbf{I}_1} \alpha_{\mathbf{1},\mathbf{j}}.\phi_{\mathbf{1},\mathbf{j}}(\mathbf{x}), \tag{7.20}$$

where $\alpha_{\mathbf{1},\mathbf{j}} \in \Re$ are the coefficient values of the hierarchical product basis representation.

It is the hierarchical representation which now allows to consider the following subspace $V_n^{(s)}$ of V_n which is obtained by replacing $|\mathbf{l}|_\infty \le n$ by $|\mathbf{l}|_1 \le n + d - 1$ (now with $l_t > 0$) in (7.16):

$$V_n^{(s)} := \bigoplus_{|\mathbf{l}| \le n+d-1} W_{\mathbf{1}}. \tag{7.21}$$

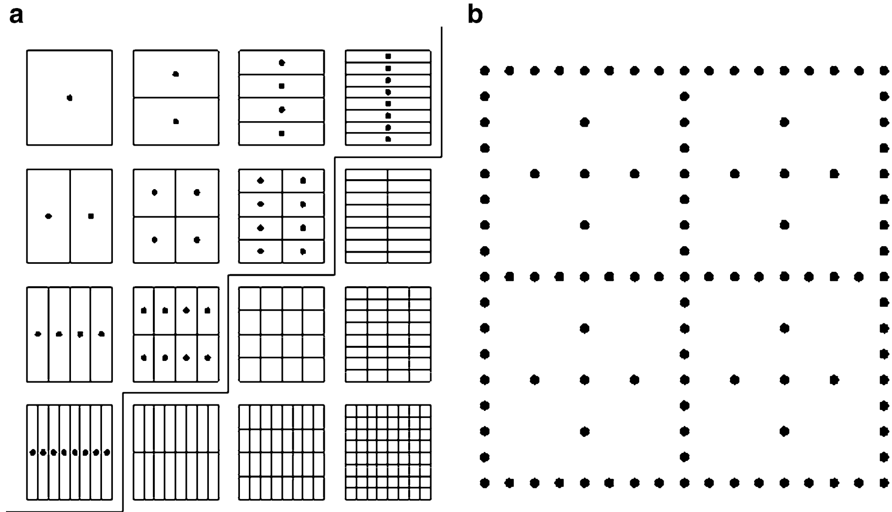


Fig. 7.1 The grids of the subspaces W_1 (**a**) and the corresponding sparse grid (**b**) for level 4 in two dimensions

Again, any function $f \in V_n^{(s)}$ can be splitted accordingly by

$$f_n^{(s)}(\mathbf{x}) \sum_{|\mathbf{l}|_1 \leq n+d-1} \sum_{\mathbf{j} \in \mathbf{I}_1} \alpha_{\mathbf{l},\mathbf{j}} \phi_{\mathbf{l},\mathbf{j}}(\mathbf{x}). \tag{7.22}$$

Definition 7.1 The grids corresponding to the approximation spaces $\mathrm{V}_n^{(s)}$ are called sparse grids.

Sparse grids have been studied in detail, e.g., in [Bun92, GMZ92, Zen91]. An example of a sparse grid for the two-dimensional case is given in Fig. 7.1b.

Now, a straightforward calculation shows that the dimension of the sparse grid space $V_n^{(s)}$ is of the order $O(n^{d-1}2^n)$. For the interpolation problem, as well as for the approximation problem stemming from second-order elliptic PDEs, it was proven that the sparse grid solution $f_n^{(s)}$ is almost as accurate as the full grid function f_n, i.e., the discretization error satisfies

$$\|f - f_n^{(s)}\|_{L_p} = O\left(h_n^2 \log\left(h_n^{-1}\right)^{d-1}\right),$$

provided that a slightly stronger smoothness requirement on f holds than for the full grid approach. Here, we need the seminorm

$$|f|_\infty := \left\| \frac{\partial^{2d} f}{\prod_{t=1}^{d} \partial x_t^2} \right\|_\infty \tag{7.23}$$

be bounded.

The idea is now to carry this discretization method and its advantages with respect to the degrees of freedom over to the minimization problem (7.1). The minimization procedure (7.4), (7.5), and (7.6) with the discrete function $f_n^{(s)}$ in $V_n^{(s)}$

$$f_n^{(s)}(\mathbf{x}) = \sum_{|\mathbf{l}|_1 \leq n+d-1} \sum_{\mathbf{j} \in \mathbf{I}_1} \alpha_{\mathbf{l},\mathbf{j}}^{(s)} \phi_{\mathbf{l},\mathbf{j}}(\mathbf{x}).$$

would result in the discrete system

$$\left(\lambda C_n^{(s)} + B_n^{(s)} . \left(B_n^{(s)}\right)^T\right) \alpha_n^{(s)} = B_n^{(s)} y \tag{7.24}$$

with

$$\left(C_n^{(s)}\right)_{(\mathbf{1},\mathbf{j}),(\mathbf{r},\mathbf{k})} = M \cdot \left(\nabla \phi_{n,(\mathbf{1},\mathbf{j})}, \nabla \phi_{n,(\mathbf{r},\mathbf{k})}\right) \text{ and } \left(B_n^{(s)}\right)_{(\mathbf{1},\mathbf{j}),i} = \phi_{n,(\mathbf{1},\mathbf{j})}(\mathbf{x}_i),$$

$|\mathbf{l}|_1 \leq n + d - 1$, $\mathbf{j} \in \mathbf{I}_1$, $|\mathbf{r}|_1 \leq n + d - 1$. $\mathbf{k} \in \mathbf{I_r}$, $i = 1, \ldots, M$, and the unknown vector $(\alpha_n^{(s)})_{(\mathbf{r},\mathbf{k})}$, $|\mathbf{r}|_1 \leq n + d - 1$, $\mathbf{k} \in \mathbf{I_r}$. The discrete problem (7.24) might in principle be treated by an appropriate iterative solver. Note that now the size of the problem is just of the order $O(n^{d-1}2^n)$. Here, the explicit assembly of the matrices $C_n^{(s)}$ and $B_n^{(s)}$ should be avoided. These matrices are more densely populated than the corresponding full grid matrices, and this would add further terms of complexity. Instead only the action on these matrices on vectors, i.e., a matrix–vector multiplication, should be performed in an iterative method like the conjugate gradient method or a multigrid method. For example, for $C_n^{(s)}$ this is possible in a number of operations which is proportional to the unknown only; see [Bun92]. However, the implementation of such a program is quite cumbersome and difficult. It also should be possible to avoid the assembly of $B_n^{(s)}$ and $B_n^{(s)} \cdot (B_n^{(s)})^T$ and to program the respective matrix–vector multiplications in $O(n^{d-1}2^n, M)$ operations. But this is complicated as well. Furthermore, the on-the-fly calculation of the action of these matrices scales with M which should be avoided for large M.

There also exists another variant of a solver working on the sparse grid, the so-called combination technique [GSZ92], which makes use of multivariate extrapolation. In the following, we apply this method to the minimization problem (7.1). It is much simpler to use than the Galerkin approach (7.24), it avoids the matrix assembly problem mentioned above, and it can be parallelized in a natural and straightforward way.

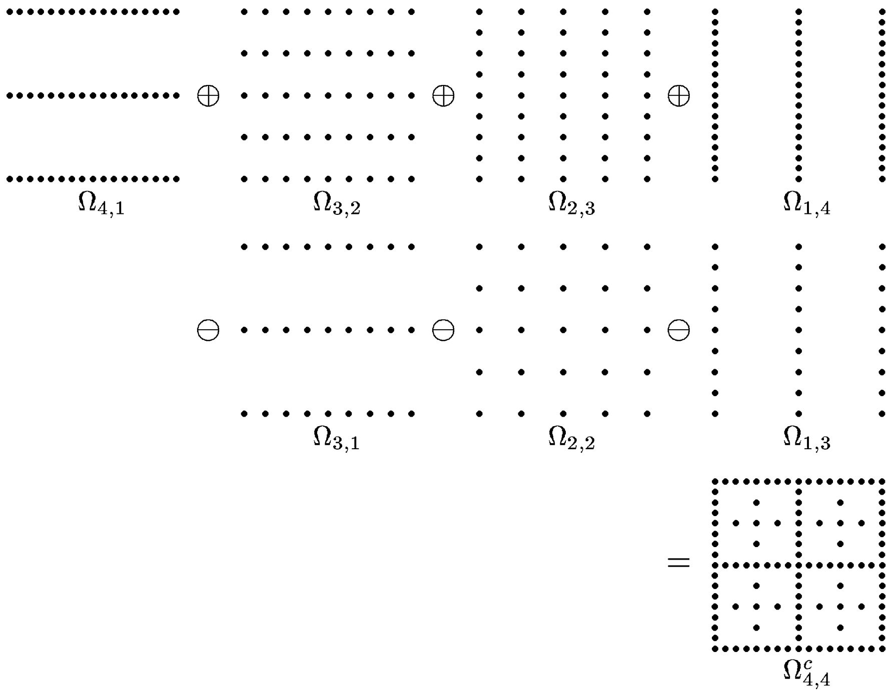


Fig. 7.2 Combination technique for level 4 in two dimensions

7.2.4 *The Sparse Grid Combination Technique*

For the sparse grid combination technique, we proceed as follows: we discretize and solve the problem on a certain sequence of grids $\Omega_{\mathbf{l}}$ with uniform mesh sizes $h_t = 2^{-l_t}$ in the tth coordinate direction. These grids may possess different mesh sizes for different coordinate directions. To this end, we consider all grids $\Omega_{\mathbf{l}}$ with

$$l_1 + \cdots + l_d = n + (d-1) - q, \quad q = 0, \ldots, d-1, l_t > 0. \tag{7.25}$$

In contrast to the definition (7.21), for reasons of efficiency, we now restrict the level indices to $l_t > 0$. For the two-dimensional case, the grids needed in the combination formula of level 4 are shown in Fig. 7.2. The finite element approach with piecewise d-linear test and trial functions $\varphi_{\mathbf{l},\mathbf{j}}(\mathbf{x})$ on grid $\Omega_{\mathbf{l}}$ now would give

$$f_{\mathbf{l}}(\mathbf{x}) = \sum_{j_1}^{2^{l_1}} \cdots \sum_{j_d}^{2^{l_d}} \alpha_{\mathbf{l},\mathbf{j}} \phi_{\mathbf{l},\mathbf{j}}(\mathbf{x})$$

and the variational procedure (7.4), (7.5), and (7.6) would result in the discrete system

$$\left(\lambda C_{\mathbf{l}} + B_{\mathbf{l}} \cdot B_{\mathbf{l}}^{T}\right)\alpha_{\mathbf{l}} = B_{\mathbf{l}} y \tag{7.26}$$

with the matrices

$$(C_{\mathbf{l}})_{\mathbf{j},\mathbf{k}} = M \cdot \left(\nabla \phi_{\mathbf{l},\mathbf{j}}, \nabla \phi_{\mathbf{l},\mathbf{k}}\right) \quad \text{and} \quad (B_{\mathbf{l}})_{\mathbf{j},i} = \phi_{\mathbf{l},\mathbf{j}}(\mathbf{x}_i),$$

$j_t, k_t = 0, \ldots, 2^{l_t}, t = 1, \ldots, d, i = 1, \ldots, M$, and the unknown vector $(\alpha_{\mathbf{l}})_{\mathbf{j}}, j_t = 0, \ldots, 2^{l_t}, t = 1, \ldots, d$. We then solve these problems by a feasible method. To this end we use here a diagonally preconditioned conjugate gradient algorithm. But also an appropriate multigrid method with partial semi-coarsening can be applied. The discrete solutions $f_{\mathbf{l}}$ are contained in the spaces $V_{\mathbf{l}}$ (see (7.11)) of piecewise d-linear functions on grid $\Omega_{\mathbf{l}}$.

Note that all these problems are substantially reduced in size in comparison to (7.8). Instead of one problem with the size $\dim(V_n) = O(h_n^{-d}) = O(2^{nd})$, we now have to deal with $O(dn^{d-1})$ problems of size $\dim(V_{\mathbf{l}}) = O(h_n^{-1}) = O(2^n)$. Moreover, all these problems can be solved independently which allows for a straightforward parallelization on a coarse grain level; see [Gri92]. Also there is a simple but effective static load balancing strategy available.

Finally, we linearly combine the results $f_{\mathbf{l}}(\mathbf{x}) = \sum_{\mathbf{j}} \alpha_{\mathbf{l},\mathbf{j}} \phi_{\mathbf{l},\mathbf{j}}(\mathbf{x}) \in V_{\mathbf{l}}$ from the different grids $\Omega_{\mathbf{l}}$ as follows:

$$f_n^{(c)}(\mathbf{x}) := \sum_{q=0}^{d-1} (-1)^q \binom{d-1}{q} \sum_{|\mathbf{l}|_1 = n+(d-1)-q} f_{\mathbf{l}}(\mathbf{x}). \tag{7.27}$$

The resulting function $f_n^{(c)}$ lives in the above-defined sparse grid space $V_n^{(s)}$ (but now with $l_t > 0$ in (7.21)).

The combination technique can be interpreted as a certain multivariate extrapolation method which works on a sparse grid space; for details see [GSZ92]. The combination solution $f_n^{(c)}$ is in general not equal to the Galerkin solution $f_n^{(s)}$, but its accuracy is usually of the same order; see [GSZ92]. To this end, a series expansion of the error is necessary. Its existence was shown for PDE-model problems in [BGRZ94].

Note that the summation of the discrete functions from different spaces $V_{\mathbf{l}}$ in (7.27) involves d-linear interpolation which resembles just the transformation to a representation in the hierarchical basis (7.19). However, we never explicitly assemble the function $f_n^{(c)}$ but rather keep the solutions $f_{\mathbf{l}}$ on the different grids $\Omega_{\mathbf{l}}$ which arise in the combination formula. Now, any linear operation F on $f_n^{(c)}$ can easily be expressed by means of the combination formula (7.27) acting directly on the functions $f_{\mathbf{l}}$, i.e.,

$$F\left(f_n^{(c)}\right) = \sum_{q=0}^{d-1} (-1)^d \binom{d-1}{d} \sum_{|\mathbf{l}| = n+(d-1)-q} F(f_{\mathbf{l}}). \tag{7.28}$$

Therefore, if we now want to evaluate a newly given set of data points $\{\widetilde{\mathbf{x}}_i\}_{i=1}^{\widetilde{M}}$ (the test or evaluation set) by

$$\widetilde{y}_i := f_n^{(c)}(\widetilde{\mathbf{x}}_i), \quad i = 1, \ldots, \widetilde{M},$$

we just form the combination of the associated values for $f_{\mathbf{l}}$ according to (7.30).

The learning on the training data is summarized in Algorithm 7.1. It consists of assembling the matrices $C_{\mathbf{l}}$ and $B_{\mathbf{l}}$ for the different grids $\Omega_{\mathbf{l}}$ and solving the corresponding discrete systems (7.26). The main results are the coefficient vectors $\alpha_{\mathbf{l}}$ for the grids. In the next section of adaptive learning, we will also need the matrices $C_{\mathbf{l}}$ and $B_{\mathbf{l}}$.

Algorithm 7.1: Computation of sparse grid classifier

Input: training data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^{M}$, regularization parameter λ

Output: coefficients $\alpha_{\mathbf{l}}$ (matrices $C_{\mathbf{l}}$ and $B_{\mathbf{l}}$)
  for $q = 0, \ldots, d-1$ **do**
    for $l_1 = 1, \ldots, n-q$ **do**
      for $l_2 = 1, \ldots, n-q-(l_1-1)$ **do**
        ...
          for $l_{d-1} = 1, \ldots, n-q-(l_1-1)-\ldots-(l_{d-2}-1)$ **do**
            $l_d = n-q-(l_1-1)-\ldots-(l_{d-2}-1)-(l_{d-1}-1)$
              assemble matrices $C_{\mathbf{l}}$ and $B_{\mathbf{l}}$
              solve the linear system $(\lambda C_{\mathbf{l}} + B_{\mathbf{l}} \cdot B_{\mathbf{l}}^T)\alpha_{\mathbf{l}} = B_{\mathbf{l}} y$
          end for
        ...
      end for
    end for
  end for

Algorithm 7.2 shows the application of the classifier (represented by the coefficients $\alpha_{\mathbf{l}}$) to the test data set $\{\widetilde{\mathbf{x}}_i\}_{i=1}^{\widetilde{M}}$ as described above.

Algorithm 7.2: Evaluation of sparse grid classifier

Input: test data set $\{\widetilde{\mathbf{x}}_i\}_{i=1}^{\widetilde{M}}$, coefficients $\alpha_{\mathbf{l}}$

Output: set of score values $\{\widetilde{y}_i\}_{i=1}^{\widetilde{M}}$
  $\widetilde{y}_i = 0, i = 1, \ldots, \widetilde{M}$
    for $q = 0, \ldots, d-1$ **do**
      for $l_1 = 1, \ldots, n-q$ **do**
        for $l_2 = 1, \ldots, n-q-(l_1-1)$ **do**
          ...

(continued)

Algorithm 7.2: (continued)

for $l_{d-1} = 1, \ldots, n - q - (l_1 - 1) - \ldots - (l_{d-2} - 1)$ **do**
$l_d = n - q - (l_1 - 1) - \ldots - (l_{d-2} - 1) - (l_{d-1} - 1)$
$\widetilde{y}_i := \widetilde{y}_i + (-1)^d \begin{pmatrix} d-1 \\ d \end{pmatrix} f_{\mathbf{l}}(\widetilde{\mathbf{x}}_i), i = 1, \ldots, \widetilde{M}$
end for
...
end for
end for
end for

The combination technique is only one of the various methods to solve problems on sparse grids. Note that there exist also finite difference and Galerkin finite element approaches which work directly in the hierarchical product basis on the sparse grid. But the combination technique is conceptually much simpler and easier to implement. Moreover, it allows to reuse standard solvers for its different subproblems and is straightforwardly parallelizable.

7.2.5 Adaptive Sparse Grids

We will now develop an adaptive version of Algorithm 7.1. Note that there are different types of adaptivity of sparse grids. Here we are interested in data adaptivity, i.e., adaptivity with respect to new data points.

We consider the initial data set $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^{M}$ and apply Algorithm 7.1 to it. As a result we obtain the solution coefficients $\alpha_{\mathbf{l}}$ of (7.26) as well as the matrices $C_{\mathbf{l}}$ and $B_{\mathbf{l}}$. Suppose now that we get new data points $\hat{S} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{\hat{M}}$ from the same distribution. We are looking for an efficient procedure to find $\overline{\alpha}_{\mathbf{l}}$ that solves (7.26) for all $\overline{M} = M + \hat{M}$ data points of $\overline{S} = S \cup \hat{S}$, and the solution shall be based on $\alpha_{\mathbf{l}}$ and may use $C_{\mathbf{l}}$ and $B_{\mathbf{l}}$.

We rewrite the system (7.26) as

$$\left(\lambda M C_{\mathbf{l}}^{'} + B_{\mathbf{l}} \cdot B_{\mathbf{l}}^{T}\right) \alpha_{\mathbf{l}} = B_{\mathbf{l}} y \tag{7.29}$$

with the matrices

$$\left(C_{\mathbf{l}}^{'}\right)_{\mathbf{j},\mathbf{k}} = \left(\nabla \phi_{\mathbf{l},\mathbf{j}}, \nabla \phi_{\mathbf{l},\mathbf{k}}\right) \text{and} \left(B_{\mathbf{l}}\right)_{\mathbf{j},i} = \phi_{\mathbf{l},\mathbf{j}}(\mathbf{x}_i).$$

Obviously, now the Laplacian $C'_{\mathbf{l}}$ does not depend on the data points $\{\mathbf{x}_i\}_{i=1}^{M}$ at all. Thus, we are looking for the solution of

$$\left(\lambda \overline{M} C_{\mathbf{l}}^{'} + \overline{B}_{\mathbf{l}} \cdot \overline{B}_{\mathbf{l}}^{T}\right) \overline{\alpha}_{\mathbf{l}} = \overline{B}_{\mathbf{l}} \overline{y} \tag{7.30}$$

in terms of that of (7.29). Concerning the matrix $B_{\mathbf{l}}$, it is obtained by just adding the new entries of the basis functions in the data points $\{\widetilde{\mathbf{x}}_i\}_{i=1}^{\widetilde{M}}$ to $B_{\mathbf{l}}$. Notice also that the number of unknown of (7.29) only depends on the grid points of the sparse grid and does not change for an increasing number of data points.

Of course, after adding the new data points of $\hat{S}$, we need to solve the complete equation system (7.30) in order to determine $\overline{\alpha}_{\mathbf{l}}$. However, as stated before, (7.29) has the same dimensionality as (7.30), and provided that $\hat{S}$ is not very large, the system (7.30) is very close to (7.29). So we can use $\alpha_{\mathbf{l}}$ as initial iterate in the iteration method of (7.30), and the solution will require, in general, only a few iterations.

Notice that there are different ways to assemble and solve the system (7.26); we did not discuss them in detail here. In general, we assemble $G_{\mathbf{l}} = B_{\mathbf{l}} \cdot B_{\mathbf{l}}^T$ and use it for multiplication with $\alpha_{\mathbf{l}}$ instead of multiplying with $B_{\mathbf{l}}^T$ and $B_{\mathbf{l}}$ directly. Since the dimension of $G_{\mathbf{l}}$ only depends on the number of grid points and, unlike as for $B_{\mathbf{l}}$, not on the number of data points M, it is especially suited for our adaptive approach. Therefore, we use $\overline{G}_{\mathbf{l}} = \overline{B}_{\mathbf{l}} \cdot \overline{B}_{\mathbf{l}}^T$ that can be incrementally calculated from $G_{\mathbf{l}}$. In the same way, we define $h_{\mathbf{l}} = B_{\mathbf{l}} y$ and also $\overline{h}_{\mathbf{l}}$ can be easily calculated from $h_{\mathbf{l}}$. Thus, we rewrite (7.30) as

$$\left(\lambda \overline{M} C'_{\mathbf{l}} + \overline{G}_{\mathbf{l}}\right)\overline{\alpha}_{\mathbf{l}} = \overline{h}_{\mathbf{l}}. \tag{7.31}$$

This way we arrive at the adaptive sparse grid Algorithm 7.3.

Algorithm 7.3: Adaptive computation of sparse grid classifier

---

Input: new training data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^{\hat{M}}$, λ, coefficients $\alpha_{\mathbf{l}}$, vectors $h_{\mathbf{l}}$, matrices $C'_{\mathbf{l}}$ and $G_{\mathbf{l}}$

Output: updated coefficients $\overline{\alpha}_{\mathbf{l}}$, updated vectors $\overline{h}_{\mathbf{l}}$ and matrices $\overline{G}_{\mathbf{l}}$

for $q = 0, \ldots, d-1$ **do**
  for $l_1 = 1, \ldots, n-q$ **do**
    for $l_2 = 1, \ldots, n-q-(l_1-1)$ **do**
      ...
        for $l_{d-1} = 1, \ldots, n-q-(l_1-1)-\ldots-(l_{d-2}-1)$ **do**
          $l_d = n-q-(l_1-1)-\ldots-(l_{d-2}-1)-(l_{d-1}-1)$
          update matrix $\overline{G}_{\mathbf{l}}$ from $G_{\mathbf{l}}$ and vector $\overline{h}_{\mathbf{l}}$ from $h_{\mathbf{l}}$
          solve the linear system $\left(\lambda \overline{M} C'_{\mathbf{l}} + \overline{G}_{\mathbf{l}}\right)\overline{\alpha}_{\mathbf{l}} = \overline{h}_{\mathbf{l}}$ starting with $\alpha_{\mathbf{l}}$
        end for
      ...
    end for
  end for
end for

For practical applications, we can start with Algorithm 7.1 on historic data S and then apply Algorithm 7.3 on new data points $\hat{s}$. This is again a nice example of combining offline and online learning.

7.2.6 *Further Sparse Grid Versions*

There are many different versions and modifications of the sparse grid approach. We want to explain some of them which may be relevant for scoring and reinforcement learning.

7.2.6.1 Simplicial Basis Functions

So far we only mentioned d-linear basis functions based on a tensor-product approach. But on the grids of the combination technique, linear basis functions based on simplicial discretization are also possible. Here, the so-called Kuhn triangulation for each rectangular block is used [Kuh60]. Now the summation of the discrete functions for the different spaces $V_{\mathbf{l}}$ in (7.27) only involves linear interpolation.

The theoretical properties of this variant of the sparse grid technique still have to be investigated in more detail. However, the results warrant its use. There are, if at all, just slightly worse results with linear basis functions than with d-linear basis functions, and we believe that this new approach results in the same approximation order.

Since in the new variant of the combination technique, the overlap of supports, i.e., the regions where two basis functions are both nonzero, is greatly reduced due to the use of a simplicial discretization, the complexities scale significantly better. By using simplicial basis functions, sparse grid classification can be performed with up to 20–22 dimensions on a conventional PC. For details about the simplicial basis functions for sparse grids, we refer to [GG01a].

7.2.6.2 Anisotropic Sparse Grids

Up to now we treated all attributes of the classification problem the same, i.e., we used the same mesh refinement level for all attributes. Obviously attributes have different properties, different number of distinct values, and different variances. For example, to discretize the range of a binary attribute, one does not need more than two grid points.

We can generalize our approach to account for such situations as well. We use different mesh sizes for each dimension along the lines of [GG98]. This results in a so-called anisotropic sparse grid. Now different refinement level n_j for each dimension $i, j = 1, \ldots, d$ can be given instead of only the same refinement level

n for the different dimensions. This extension of our approach can result in less computing time and better approximation results, depending on the actual data sets. For details, see [GG01a].

7.2.6.3 Dimension-Adaptive Sparse Grids

The combination technique presented so far can be used for a maximum of 20–25 dimensions. In order to advance into higher dimensions, the dimension-adaptive combination technique has been proposed [Gar11]. Based on error estimators, it automatically performs an adaptive refinement of an index set that represents the grids. Technically, problems with up to 100 dimensions can be handled by this technique. However, the development of the error estimator is difficult and a topic of current research.

7.2.6.4 Other Operator Equations

So far we have studied the minimization problem (7.4) using the gradient in the regularization expression, i.e.,

$$R(f_N) = \frac{1}{M}\sum_{i=1}^{M}(f_N(\mathbf{x}_i), y_i)^2 + \lambda\|\nabla f_N\|_{L_2}^2. \tag{7.32}$$

Of course, we can use many other operators P. For example, in [Pfl10] a much simpler functional has been used, exploiting the inherent smoothness of the hierarchical basis which is known to be spectrally close to the Laplacian. In fact, the Euclidean norm of the coefficient vector was used as regularization operator.

Now the minimization problem reads

$$R(f_N) = \frac{1}{M}\sum_{i=1}^{M}(f_N(\mathbf{x}_i), y_i)^2 + \lambda\sum_{i=1}^{N}\alpha_i^2 \tag{7.33}$$

and results in the linear system

$$\left(\lambda MI + B \cdot B^T\right)\alpha = By. \tag{7.34}$$

An advantage of the simple formulation (7.33) is that direct sparse grid discretization can be employed easily. Unlike as for the combination technique, where we can use dimension adaptivity only, for the direct sparse grid discretization, we can use spatial (local) adaptivity in a straightforward way. This allows to apply sparse grid classification of (7.33) also for very high dimensions. In [Pfl10] this sparse grid classification delivered good results for many high-dimensional data sets like character recognition in 64 dimensions and even a

music data set in 166 dimensions. However, the prediction quality of formulation (7.33) is somewhat lower compared to our more complex one (7.32).

Moreover, sparse grids are not restricted to regularization network problems (7.1) at all! In fact they can be used to a wide family of high-dimensional differential and integral equations. Of course, the function f needs to be smooth. This usually applies to most scoring problems. But even the non-smooth case can be handled, depending on the problem, using adaptive sparse grids. Also the concept of adaptive sparse grids can be generalized (see [Heg03]), arriving at the following core ideas: grids that are sparse, hierarchically organized, and adaptively refined. This results in a very powerful approximation approach which, however, requires much further research. At this, one of the most complex problems is the development of error estimation techniques required for grid refinement; see [Gar12b].

7.2.6.5 Sparse Grids for Regression in Reinforcement Learning

As stated in Sect. 6.1.1 and used in Chap. 10, for the representation of the state-value function v and action-value function q in reinforcement learning, often regression models are used which can handle large state spaces S and can also serve for regularization. Here, sparse grids are a good candidate, especially because they can solve complex operator equations on large data volumes and are well suited for adaptivity. For the continuous counterpart of the Bellman equation, the Hamilton-Bellman-Jacobi equation, promising results have been obtained recently [BGGK12].

7.3 Experimental Results

We now apply our approach to different data sets. Both synthetic and real data from practical data mining applications are used. All the data sets are rescaled to $[0,1]^d$. To evaluate our method, we give the correctness rates on testing data sets, if available, or the tenfold cross-validation results otherwise. For a critical discussion on the evaluation of the quality of classification algorithms, see [Diet98, Sal97].

The results are mostly based on the offline Algorithm 7.1 since the adaptive Algorithm 7.3 yields the same results. The equivalence of the results of both offline and online algorithms will be demonstrated in the last example.

7.3.1 Two-Dimensional Problems

Example 7.2 The first example is the spiral data set proposed by Alexis Wieland of MITRE Corp [Wie88]. Here, 194 data points describe two intertwined spirals; see Fig. 7.3. This is surely an artificial problem which does not appear in practical

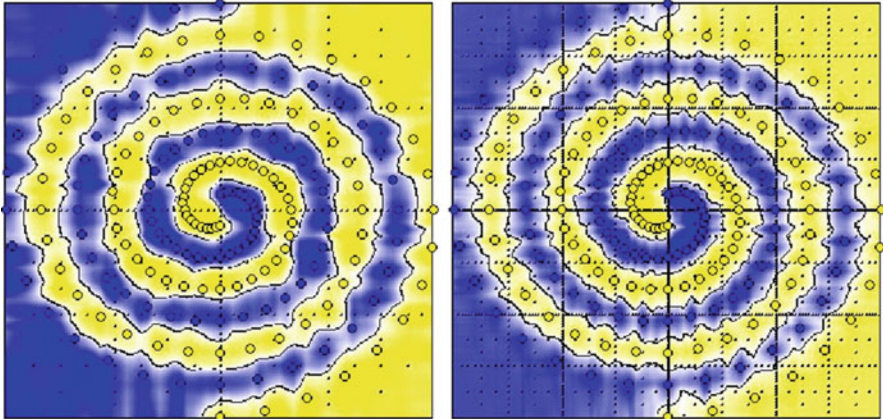


Fig. 7.3 Spiral data set, sparse grid with levels 5 (*left*) and 7 (*right*)

Table 7.2 Leave-one-out cross-validation results for the spiral data set

Level	λ	Training correctness (%)	Testing correctness (%)
4	0.00001	95.31	87.63
5	0.001	94.36	87.11
6	0.00075	100.00	89.69
7	0.00075	100.00	88.14
8	0.0005	100.00	87.63

applications. However, it serves as a hard test case for new data mining algorithms. It is known that neural networks can have severe problems with this data set and some neural networks cannot separate the two spirals at all. In Table 7.2 we give the correctness rate achieved with the leave-one-out cross-validation method, i.e., a 194-fold cross-validation. For the sparse grids, use the tensor-product basis functions as described in this chapter.

The best testing correctness was achieved on level 6 with 89.69 % in comparison to 77.20 % in [Sin98].

In Fig. 7.3 we show the corresponding results obtained with our sparse grid combination method for levels 5 and 7. With level 7 the two spirals are clearly detected and resolved. Note that here 1,281 grid points are contained in the sparse grid. ■

Example 7.3 This data set Ripley, taken from [Rip94], consists of 250 training data and 1,000 test points. It is shown in Fig. 6.4a. The data set was generated synthetically and is known to exhibit 8 % error. Thus no better testing correctness than 92 % can be expected. As before, we use tensor-product basis functions.

Since we now have training and test data, we proceed as follows: first, we use the training set to determine the best regularization parameter λ. The best test correctness rate and the corresponding λ are given for different levels n in the

Table 7.3 Results for the Ripley data set

	Tenfold			Best	
Level	Tenfold testing (%)	λ	On test data (%)	λ	Testing (%)
1	84.8	0.01005	89.8	0.00370	90.3
2	85.2	0.000001	90.4	0.00041	90.9
3	88.4	0.00166	90.6	0.00370	91.2
4	87.6	0.00248	90.6	0.01500	91.2
5	87.6	0.01005	90.9	0.00673	91.1
6	86.4	0.00673	90.8	0.00673	90.8
7	86.4	0.00075	88.5	0.00673	91.0
8	88.0	0.00166	89.7	0.00673	91.0
9	88.4	0.00203	90.9	0.00823	91.0
10	88.4	0.00166	90.6	0.00452	91.1

first two columns of Table 7.3. With this λ, we then compute the sparse grid classifier from the 250 training data. Column 3 of Table 7.3 gives the result of this classifier on the (previously unknown) test data set. We see that our method works well. Already level 5 is sufficient to obtain results of 90.0 %. We also see that there is not much need to use any higher levels. The reason is surely the relative simplicity of the data. Just a few hyperplanes should be enough to separate the classes quite properly. This is achieved with the sparse grid already for a small number n.

Additionally, we give in Table 7.3 the testing correctness which is achieved for the best possible λ. To this end we compute for *all* (discrete) values of λ the sparse grid classifier from the 250 data points and evaluate them on the test set. We then pick the best result. We clearly see that there is not much of a difference. This indicates that our approach to determine the value of λ from the training set by cross-validation works well. Note that a testing correctness of 90.6 % was achieved with neural networks in [Rip94]. ■

7.3.2 High-Dimensional Problems

Example 7.4 The 10-dimensional data set *ndcHArd* consists of two million instances synthetically generated and was first used in [MM01]. Here, the main observations concern the run time.

In Table 7.4 we give the results using the combination technique with simplicial basis functions as described in Sect. 7.2.6. More than 50 % of the run time is spent for the assembly of the data matrix. The time needed for the data matrix scales linearly with the number of data points. The total run time seems to scale even better than linearly. Already at level 1, we get 84.9 % testing correctness, and no improvement with level 2 is achieved. Notice that with support vector machines, correctness rates of 69.5 % were reported in [FM01].

Table 7.4 Results for the ndcHArd data set

	Number of points	Training correctness (%)	Training correctness (%)	Total time [s]	Data matrix time [s]	Number of iterations
Level 1	20,000	86.2	84.2	6.3	0.9	45
	200,000	85.1	84.8	16.2	8.7	51
	2 million	84.9	84.9	114.9	84.9	53
Level 2	20,000	85.1	83.8	134.6	10.3	566
	200,000	84.5	84.2	252.3	98.2	625
	2 million	84.3	84.2	1,332.2	966.6	668

Table 7.5 Results for the web shop data set with a block size of 10,000 data points

Blocks	Correctness (%)	Total time [s]	Blocks	Correctness (%)	Total time [s]
1	44.79	12.05	10	62.94	96.22
2	53.61	20.52	11	63.32	106.82
3	57.53	29.06	12	63.54	117.51
4	59.29	38.33	13	63.59	128.61
5	60.71	47.50	14	63.88	139.98
6	61.60	56.76	15	64.15	151.32
7	61.97	66.13	16	64.29	163.04
8	62.36	75.83	17	64.51	174.88
9	62.72	85.78	18	64.67	184.72

Example 7.5 Now we will compare the results of the offline Algorithm 7.1 with that of Algorithm 7.3, its adaptive counterpart. We use a data set suitable for scoring-based recommendations as they have been described in the introduction, especially in Example 7.1. The data is from a large web shop and each data point represents a certain transaction in a web session, such as a click, basket event, or order. The data is from 1 day. There are 15 attributes from three data sources:

- Session specific: number of clicks and basket products in current session
- User specific: gender, customer value, etc. (only for recognized users)
- External from host: step in checkout process, availability of products, etc.

As in Example 7.1, the target attribute is 0 if no order was placed within the session and 1 if something was ordered. The task is to find a good classification function that predicts at each step of a web session if the user will finally place an order.

Thus, the data set has 15 dimensions. The number of data points is 177,907. For simplicity, we use the training set as test set, too. As in the previous example, we again apply simplicial basis functions which are computationally cheaper the tensor-product ones. By using the offline Algorithm 7.1, for level 1 we obtain a training correctness of 67.90 % and the run time is 283 s. Now we apply the adaptive Algorithm 7.3 to the same problem. For a block size $\hat{M} = 10{,}000$, the results are shown in Table 7.5.

The classification rate of the resulting classifier applied to the full data set if 67.91 % and thus almost the same as that obtained by the offline algorithm. Concerning the total time of 185 s, the adaptive version is even faster than the offline one. However, when we further reduce the block size, the situation is changing. For the block size $\hat{M} = 1{,}000$, the run time of the adaptive version increases to 443 s suffering from the overhead of each update.

7.4 Summary

In this chapter we have studied sparse grid approximation for scoring, i.e., classification and regression. We first demonstrated how scoring can be efficiently used for generating recommendations.

We discussed shortcomings of current classification methods. We then introduced sparse grids and demonstrated how they can overcome many of these problems. Especially, sparse grids scale linearly with the number of data points and thus can handle huge data sets. Data adaptivity can directly be applied to sparse grid algorithms. Another advantage is that sparse grid classifiers can be interpreted and manipulated in a spectral sense as it is known from signal processing. Moreover, sparse grids can in general be applied to wide classes of operator equations. In summary, sparse grids represent a new quality of data analysis based on hierarchical decomposition of the attribute space.

On the other hand, sparse grids are very complex in theory and application. There exist many versions of sparse grid techniques. Much further research is required to extend the classical concepts of the numerical analysis of PDEs to the high-dimensional case in order to use them for sparse grids.

Chapter 8
Decomposition in Transition: Adaptive Matrix Factorization

Abstract We introduce SVD/PCA-based matrix factorization frameworks and present applications to prediction-based recommendation. Furthermore, we devise incremental algorithms that enable to compute the considered factorizations adaptively in a realtime setting. Besides SVD and PCA-based frameworks, we discuss more sophisticated approaches like non-negative matrix factorization and Lanczos-based methods and assess their effectiveness by means of experiments on real-world data. Moreover, we address a compressive sensing-based approach to Netflix-like matrix completion problems and conclude the chapter by proposing a remedy to complexity issues in computing large elements of the low-rank matrices, which, as we shall see, is a recurring problem related to factorization-based prediction methods.

In conventional modeling, the states correspond to the products being viewed. As we saw in Chap. 5, this assumption essentially complies with the Markov property of most RE applications. It would of course be better, though, to gather more information in each state. This applies mainly to previous transactions, but other dimensions such as user, price, and channels with their various attributes may also be useful. Thus, we will drop the considered Markov property and describe the corresponding procedure in this chapter.

This is where the realtime approach described in this book coincides with the complex analysis models on which most RE researchers are currently working (Chap. 2). While we have so far concentrated only on the simplest analysis scenario, namely, product rules in the form $s \to s'$, albeit in a modern realtime analytical context, the latest analytical approaches can already achieve good predictions, sometimes using multiple dimensions, but only in a static analysis context. So the obvious solution is to combine the two approaches. In principle it makes no difference which route is followed: expanding RL to include more extensive state definitions or adding control functions to conventional approaches. Given the general focus of this book, we will concentrate on the first route: expanding the definition of states in RL.

A. Paprotny and M. Thess, *Realtime Data Mining: Self-Learning Techniques for Recommendation Engines*, Applied and Numerical Harmonic Analysis,
DOI 10.1007/978-3-319-01321-3_8, © Springer International Publishing Switzerland 2013

Rather than only considering states of single products p_i where $s_i = \{p_i\}$, as before, the simplest approach is to switch to short product sequences of l previous products (or, more precisely, product views) in the episode and to include these as equally valid states $s_j = \{p_j, p_{j-1}, \ldots, p_{j-l+1}\}$ in the state space $\boldsymbol{S}$. Because of its complexity, this approach, which can also be found in the literature (for RL in [SHB05]), is of course very limited and can only reasonably be used for small values of l (usually 2 or 3). At most it represents a small expansion of the existing approach, but it does not solve the crux of the problem.

Probably the most promising route is to factorize the action-value and state-value functions and in the model-based case additionally the transition probabilities or rewards. This tensor factorization not only allows a theoretically unlimited number of new dimensions to be included but also makes it possible to regularize transition probabilities in particular. We proceed as follows: in this chapter we will introduce the tensor factorization, especially in its adaptive form, and combine it with reinforcement learning in Chap. 10.

8.1 Matrix Factorizations in Data Mining and Beyond

In classical data mining, approaches based on matrix factorization are ubiquitous. Typically, they arise as mathematical core problems in *collaborative filtering* (CF). A classical application of CF to recommendation engineering is the prediction of product ratings by users. Unlike in classical CF, we shall use sessions instead of users (from a mathematical point of view, this does not make any difference) for consistency reasons in the following. Like RL, CF is behavioristic in the sense that no background information with respect to neither of users nor products is involved.

Instead, we associate with each session a list assigning to each product the rating given by the user. These ratings may be explicit, e.g., users may be prompted to rate each visited product on a scale from 1 to 5, or, more commonly, implicit (as before in this book). As for the latter, one may, for instance, endow each type of customer transaction with a score value, say, 1 for a click, 5 for an "*add to cart*" or "*add to wish list,*" and 10 for actually buying the product. We consider this list as a signal or, simply, a vector. Inspired by noise reduction and deconvolution techniques in signal processing, most CF approaches are based on the assumption that the thus arising data are noise-afflicted observations of intrinsically low-dimensional signals generated by some unknown source. How shall we proceed to formalize the situation statistically?

The decisive obstacle is the mathematical treatment of the unknown values. Basically, this may be surmounted in two different manners: the ostensibly more sophisticated approach consists in modeling the unknown ratings as hidden variables, which need to be estimated along with the underlying source. Putting it in terms of signal processing, this gives rise to a problem related to the reconstruction of a partially observed signal. Dealing with hidden variables in

Table 8.1 Example of a session matrix of a web shop

	Session 1	Session 2	Session 3	Session 4
Product 1	0	1	10	5
Product 2	1	5	1	1
Product 3	0	5	0	1

statistical models, however, entails some major computational impediments, including intractable integrals and non-convex optimization, making a realtime implementation very difficult. The alternative approach consists in treating all variables as observed by assigning the value 0 to the unknown ratings. Although this may appear somewhat helter-skelter at the first glance, it may be rationalized by considering *not visiting a product* as a transaction corresponding to the lowest possible rating. Assuming, furthermore, that many of the zero entries are due to noise rather than intrinsic, we may put the approach on a sound footing. We will return to this discussion in Sect. 8.5.

Now we consider a matrix of rewards $A \in \Re^{n_p \times n_s}$ with n_s being the current number of sessions and n_p the number of different products over all sessions observed so far. Neither the order of sessions nor the order of products within the sessions is taken into account.

Example 8.1 As an example, we consider a web shop with 3 products and 4 sessions, i.e., $n_p = 3$ and $n_s = 4$. The session values are displayed in Table 8.1.

In terms of the reward assignment described above, this means, e.g., for session 2, product 1 has been clicked, whereas products 2 and 3 have moreover been added to the basket. In session 3, product 1 has been purchased, product 2 has been clicked, and product 3 has been skipped. ■

Mathematically, the matrix factorization problems arising in CF are of the form

$$\min_{X \in C_1 \subset \Re^{n_p \times r},\, Y \in C_2 \subset \Re^{r \times n_s}} f(A, XY). \tag{8.1}$$

The rank r is usually chosen to be considerably smaller than n_p. The function f is referred to as the cost function of the factorization and, more often than not, is chosen to be a metric. It stipulates a notion of quality of a factorization. The sets C_1, C_2 determine the parameter space. In terms of our signal processing metaphor, the factor X characterizes the source, which is restricted to be a low-dimensional subspace, and the columns Y are the intrinsic low-dimensional parameter vectors determining the signals given by the corresponding columns of A.

To put it even simpler, we approximate the matrix A by the product of two smaller matrices X and Y. The cost function stipulates a notion of "closeness," i.e., distance, of two matrices. Since the rank r is typically much smaller than n_p and n_s, the representation in terms of X and Y is much more compact than an explicit representation of the entries of A.

Example 8.2 Let us consider the following factorization for Example 8.1 with $r = 1$:

$$\underbrace{\begin{pmatrix} 1 \\ 0.2 \\ 0.1 \end{pmatrix}}_{X} \underbrace{\begin{pmatrix} 0.23 & 2.76 & 9.65 & 5.17 \end{pmatrix}}_{Y} = \underbrace{\begin{pmatrix} 0.23 & 2.76 & 9.65 & 5.17 \\ 0.05 & 0.55 & 1.93 & 1.03 \\ 0.02 & 0.28 & 0.97 & 0.52 \end{pmatrix}}_{\widetilde{A}}$$

$$\approx \underbrace{\begin{pmatrix} 0 & 1 & 10 & 5 \\ 1 & 5 & 1 & 1 \\ 0 & 5 & 0 & 1 \end{pmatrix}}_{A}$$

While the initial matrix A consists of 12 elements, the factors X and Y taken together contain only 7 elements. Now we have to assess whether our rank-1 approximation $XY = \widetilde{A}$ is a sufficiently good approximation to A. If not, we may increase the rank r, which, of course, entails an increase of complexity of the factors. It is obvious, however, that for large n_p and n_s and a moderate rank, the factorized representation is by orders of magnitude more compact than the explicit one. ■

In terms of CF, a commonly encountered intuitive interpretation is as follows: the matrix Y maps the sessions to their virtual profiles, the number of which is given by the rank r, and the matrix X maps profiles to their products of reference.

It is also noteworthy that optimal factors are almost never unique, even if their product is. This is only a minor difficulty since we are eventually interested in the latter and may choose among the corresponding optimal factors arbitrarily.

Please note that the framework stipulated by (8.1) is of profound generality. It encompasses a vast majority of commonly deployed factorization models. In particular, we stress that the computational complexity of a factorization (8.1) depends crucially on the choice of f and C_1, C_2. For example, the factorization model related to PCA, which we shall focus on in what follows, may be reduced to a rather "simple" algebraic problem capable of being solved optimally by algorithms of polynomially bounded complexity. On the other hand, it is possible to state the well-known clustering or vector quantization problem in terms of the above framework. This problem, however, is NP-hard.

As opposed to the control theoretic varieties discussed in the foregoing chapters, REs based on CF are "naïve-old fashioned." Why, you may ask yourself, after so keenly campaigning for the latter, do we suddenly address so unsophisticated and outdated approach? The reasons for doing so are as follows:

- In recent research, we have found a way to perform PCA-based CF in a realtime adaptive fashion. Since this book is intended to be about adaptive rather than only about the smaller class of control theoretic recommendation systems, this fits well into the framework.
- We are currently working on higher-order (i.e., tensor) generalization of PCA-based CF. This enables to deploy CF in a less behavioristic fashion.

It turns out that the adaptive algorithms for the matrix case carry over rather smoothly to the higher-order setting.

- Most importantly, our research in Chap. 10 will focus on a combination of adaptive CF and RL. Specifically, we are heading to apply (tensor) CF to the transition probability matrices (or tensors) of Markov decision processes as a means of approximation and regularization to render model-based methods tractable for large state spaces.

In what follows, the above points will be discussed in more detail.

8.2 Collaborative Filtering

Let us, for the sake of completeness, first address classical collaborative filtering. The first ever CF system was the *Information Tapestry Project* by *Xerox PARC* in the early 1990s [GNBT92]. In the further development, the research group *GroupLens* played an important part.

In classical CF, an identical "decomposition" of the matrix is carried out in (8.1), that is, X is taken to be A itself and Y to be the identity matrix. Thus, the "low-dimensional" subspace is given by the data space. Intuitively, there is no decomposition whatsoever – all sessions are stored as they are.

Having clarified the factorization, we shall briefly address how it is used for computation. Let $N(p)$ be the set of all sessions t with the considered product p (i.e., p has been clicked at least once). Then the predicted reward value $\hat{a}_{ps}$ of a product p not observed in the session is computed as follows:

$$\hat{a}_{ps} = b_{ps} + \frac{\sum_{t \in N(p)} s_{st}\left(a_{pt} - b_{pt}\right)}{\sum_{t \in N(p)} s_{st}},$$

where b_{ps} is the baseline prediction for a_{ps} (e.g., the mean value of the rewards of all products in the session) and s_{st} a measure of similarity between the sessions s and t. Frequently employed measures of similarity are the *cosine measure* and the *Pearson correlation*; see below.

What does this formula mean? For the prediction of product $\hat{a}_{ps}$, we choose all previously observed sessions that contain product p and compute the weighted mean of their rewards for the product p, where the weights are given by the similarities to the session s. That is, if a session u is closer to session s than another session t, then the reward a_{ps} makes a greater contribution than the reward a_{pt}.

In practice, the described CF approach yields fairly good results, since, in particular, it takes the entire history of sessions into account. An essential drawback, however, is the poor scaling with respect to computing time, as well as memory space. Indeed, we have to keep the entire matrix A in memory, and the

computation of each possible recommendation requires the computation of all similarities to all sessions in *N(s, p)*. Hence, classical CF can be employed for small problems only. Due to the lack of generalization, the quality of the predictions, too, is amendable.

We finish the short introduction to CF with some technical remarks. Instead of considering similarities of sessions, we can also look for similarity of products. Let *N(s)* be the set of all products q of a given session s. Now the predicted reward value $\hat{a}_{sp}$ of a product p not observed in the session is computed as follows:

$$\hat{a}_{sp} = b_{sp} + \frac{\sum\limits_{q \in N(s)} s_{pq}\left(a_{sq} - b_{sq}\right)}{\sum\limits_{q \in N(s)} s_{pq}},$$

where b_{sp} is the baseline prediction for a_{sp} (e.g., the mean value of the rewards of the product over all sessions) and s_{pq} a measure of similarity between the products p and q using the same similarity measures.

By only considering similarity of products s_{pq} for recommendations, especially applying the cosine similarity measure, we arrive at the popular *item-to-item collaborative filtering* [LSY03]. That is, for a product p, we simply recommend the products q of maximum similarity values s_{pq}. This simple static type of recommendations has proved to be very robust and useful in practice (see also discussion in Sect. 1.6).

As we already mentioned, different similarity measures can be used for s_{pq}; they can all be viewed as variants on the inner product. The cosine measure between two vectors x and y of length n is defined as

$$\cos(x, y) = \frac{< x, y >}{\|x\| \cdot \|y\|} = \frac{\sum\limits_{i=1}^{n} x_i \cdot y_i}{\sqrt{\sum\limits_{i=1}^{n} x_i^2}\sqrt{\sum\limits_{i=1}^{n} y_i^2}}.$$

For similarity of products s_{pq}, x can be considered as binary vector of the occurrence of product p over all n sessions and, similarly, y as binary vector of the occurrence of product q. Thus, we can identify x with p and y with q. This leads to an interesting observation.

Since the components x_i and y_i are binary values, i.e., 0 or 1, by introducing the support $\text{supp}(x) = \sum\limits_{i=1}^{n} x_i$, we can rewrite the cosine measure as

$$\cos(x, y)^2 = \frac{\text{supp}(x \wedge y) \cdot \text{supp}(x \wedge y)}{\text{supp}(x) \cdot \text{supp}(y)} = \frac{\text{supp}(x \wedge y)}{\text{supp}(x)} \cdot \frac{\text{supp}(y \wedge x)}{\text{supp}(y)} = \widetilde{p}_{xy}\widetilde{p}_{yx}.$$

Here $\widetilde{p}_{xy}$ is the confidence from x to y, i.e., it corresponds to our transition probability p_{xy} but without attention of the sequential order. This yields

$$\cos(x, y) = \sqrt{\widetilde{p}_{xy}\widetilde{p}_{yx}},$$

and the cosine measure can be interpreted as nonsequential counterpart to the transition probabilities. In other words, the transition probabilities in both directions between p and q are multiplied.

For factorization, we later will need the following relation. In order to calculate the cosine measure similarities between all n_p products of our transaction matrix A, we introduce the matrix $\overline{A}$ by normalizing A along all n_s columns a_i. Thus, $A = \left[a_1 \middle| \ldots \middle| a_{n_s}\right]$ and

$$\overline{A} = \left[\frac{a_1}{\|a_1\|}\middle| \ldots \middle|\frac{a_{n_s}}{\|a_{n_s}\|}\right].$$

Now the similarity matrix $\overline{S} \in \Re^{n_p x n_p}$ between all products can be simply expressed as

$$\overline{S} = \overline{A}\,\overline{A}^T.$$

8.3 PCA-Based Collaborative Filtering

8.3.1 *The Problem and Its Statistical Rationale*

In what follows, we shall introduce the factorization problem underlying PCA-based CF along with a rather intuitive geometric rationale for the procedure. Subsequently, we shall provide a statistical interpretation of the approach. The latter is rather technical and may safely be skipped by a less mathematically inclined reader.

Before plunging into the matter, we need to stipulate some basic mathematical concepts. We assume that the reader brings along basic knowledge of linear algebra at the level of an undergraduate introductory class.

The fundamental notion is that of a *linear submanifold*. Informally, a linear submanifold of $\Re^{n_p}$ is a shifted subspace. Specifically, it is a set

$$M := b + \mathrm{X} = \left\{b + x \,\middle|\, x \in \mathrm{X}\right\},$$

where X denotes a subspace of $\Re^{n_p}$ of dimension d. Given a basis $x_1, x_2, \ldots, x_d$ of V, a vector $x \in \Re^{n_p}$ lies in M if and only if

$$x = b + y_1 x_1 + \ \ldots \ + y_d x_d$$

for some real coefficients $y_1, y_2, \ldots, y_d$. In matrix notation, this corresponds to

$$x = b + Xy = [X, b]\left[y^T, 1\right]^T,$$

where $X := [x_1, x_2, \ldots, x_d]$, $y := [y_1, y_2, \ldots, y_d]^T$. A linear manifold is thus completely characterized by the matrix $[X, b]$.

We endow $\mathfrak{R}^{n_p}$ with the canonical inner product $<\cdot,\cdot>$, which is defined as

$$< p, q > := \sum_{i=1}^{n_p} p_i q_i, \quad p, q \in \mathfrak{R}^{n_p}$$

and which induces the norm

$$\|x\| := \sqrt{< x, x >}, \ \ x \in \mathfrak{R}^{n_p}.$$

Introducing a norm, which, in turn, induces a metric, gives rise to a criterion to distinguish the quality of an approximation to a given vector.

In particular, the problem of finding the best approximation to $v \in \mathfrak{R}^{n_p}$

$$\min_{m \in M} \|m - v\|$$

has a unique optimizer given by

$$\hat{m} := b + P_X(v - b),$$

where $P_X \in \mathfrak{R}^{n_p \times n_p}$ denotes the *orthogonal projector* onto X. This projector is given by

$$P_X = XX^+, \tag{8.2}$$

where

$$X^+ := \left(X^T X\right)^{-1} X^T \tag{8.3}$$

denotes the Moore-Penrose pseudo-inverse of X, which has already been used in (6.12). If it holds that

$$X^T X = I. \tag{8.4}$$

the orthogonal projector (8.2) simplifies to

$$P_X = XX^T. \tag{8.5}$$

As we have already seen in Chap. 6, the orthogonal projector plays a crucial part for many subspace decompositions and will be frequently encountered in what follows.

Geometrically, Principal Component Analysis is a linear dimensionality reduction tool. Given a set of high-dimensional data, the goal is to project the data orthogonally onto a linear manifold with a prescribed dimension, which is illustrated by Fig. 8.1.

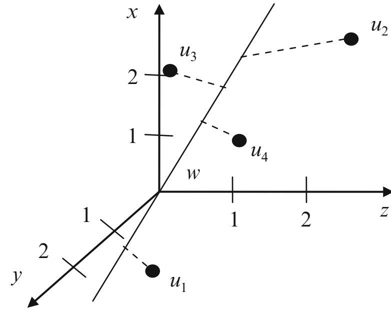


Fig. 8.1 The best approximating one-dimensional subspace (*solid line*) to a set of data residing in $\Re^3$. Projections are indicated by *dotted lines*

This manifold is chosen such that the mean-squared error resulting from the projection is minimal among all possible choices. Mathematically, the problem may be stated as follows:

$$\min_{X \in \Re^{n_p \times d},\ b \in \Re^{n_p},\ y_1,\ldots,y_{n_s} \in \Re^d} \sum_{j=1}^{n_s} \left\| a_j - X y_j - b \right\|^2, \tag{8.6}$$

where $a_1, \ldots, a_{n_s} \in \Re^{n_p}$ denote the given data. A straightforward argument reveals that b is always given by the centroid of the data, i.e., $b := n_s^{-1} \sum_{j=1}^{n_s} a_j$. Hence, assuming without loss of generality that the data are mean centered (which may always be achieved by replacing our data by $a_j - \hat{b},\ j = 1, \ldots, n_s$), the translation b may always be taken to be 0. We may thus restrict ourselves to the problem of finding the best approximating *subspace* to a set of mean-centered data:

$$\min_{X \in \Re^{n_p \times d},\ y_1,\ldots,y_{n_s} \in \Re^d} \sum_{j=1}^{n_s} \left\| a_j - X y_j \right\|^2. \tag{8.7}$$

The *Frobenius norm* is defined as

$$\|A\|_F^2 := \sum_{i=1,j=1}^{m,n} \left| a_{ij} \right|^2,\ A \in \Re^{m \times n}.$$

Summarizing our data and intrinsic variables in matrices,

$$A := [a_1, \ldots, a_{n_s}],\ Y := \left[y_1, \ldots, y_{n_s} \right],$$

we may cast (8.7) equivalently as the matrix factorization problem

$$\min_{X \in \Re^{n_p \times d}, Y \in \Re^{d \times n_s}} \|A - XY\|_F^2. \tag{8.8}$$

Recalling the general framework stipulated in (8.1), (8.8) may be stated in terms of the former by assigning $f(E, F) := \|E - F\|_F^2,\ C_1 := \Re^{n_p \times d},\ C_2 := \Re^{d \times n_s}$.

Readers familiar with optimization theory will notice that the objective of (8.8), although convex in each of the decision variables X and Y, is non-convex. Therefore, a global solution by means of optimization algorithms is hard, if not impossible. As foreshadowed in the introduction, however, (8.8) is equivalent to a well-studied algebraic problem, namely, that of a *spectral decomposition*.

Proposition 8.1 *Let* $A \in \Re^{n \times n}$ *be symmetric,* i.e., $A = A^T$. *Then there is a unique real sequence* $\lambda_1 \geq \ldots \geq \lambda_n$ *such that*

$$A = U \Lambda U^T, \tag{8.9}$$

where $\Lambda_{ij} := \delta_{ij}\lambda_i$ *and* U *is unitary (*i.e., $U^T U = UU^T = I$*). The values* λ_i *are called* eigenvalues *or* spectrum *of* A*, the corresponding columns of* U eigenvectors, *and the factorization* (8.9) eigenvalue *or* spectral decomposition *of* A.

(Proofs as well as more detailed renditions of this result may be found in any textbook on linear algebra. See, e.g., Chap. 1 of [HJ85].)

Given a matrix $A \in \Re^{m \times n}$, the corresponding *Gram matrix* is defined as

$$G := A^T A.$$

Since

$$x^T G x = x^T A^T A x = \|Ax\|^2 \geq 0,$$

G is positive semidefinite, which, as is well known in linear algebra, implies that its spectrum is nonnegative. The same holds for the *covariance matrix*

$$C := AA^T.$$

Moreover, both the Gram as well as the covariance matrices are symmetric.

Now let a spectral decomposition of G be given by

$$G = V \Lambda V^T, \tag{8.10}$$

and define

$$Z := AV.$$

Then we obtain

$$Z^T Z = V^T A^T A V = \Lambda.$$

Since Λ is a diagonal matrix of the eigenvalues of G, we may write

$$Z = US, \text{ i.e. } AV = US \tag{8.11}$$

for some unitary $U \in \Re^{m \times m}$ and $S \in \Re^{m \times n}$ defined as

$$S_{ij} := \delta_{ij} s_j, \tag{8.12}$$

where

$$s_j := \sqrt{\lambda_j}, j = 1, \ldots, n$$

are the *singular values* of A. We have thus derived the well-known *singular value decomposition* (SVD).

Proposition 8.2 (cf. Lemma 7.3.1 in [HJ85]) *Let* $A \in \Re^{m \times n}$. *Then there is a unique sequence* $s_1 \geq \ldots \geq s_m$ *such that*

$$A = USV^T, \tag{8.13}$$

where S *is as defined in (8.12), for some unitary matrices* $U \in \Re^{m \times m}$, $V \in \Re^{n \times n}$. *The values* s_j *are referred to as* singular values *of* A, *the columns of* U *as* left, *and those of* V *as* right singular vectors of A.

Example 8.3 For our Example 8.1 of a web shop with matrix

$$A = \begin{pmatrix} 0 & 1 & 10 & 5 \\ 1 & 5 & 1 & 1 \\ 0 & 5 & 0 & 1 \end{pmatrix},$$

we approximately obtain the following SVD:

$$S = \begin{pmatrix} 11.49 & 0 & 0 & 0 \\ 0 & 6.88 & 0 & 0 \\ 0 & 0 & 0.77 & 0 \end{pmatrix}, U = \begin{pmatrix} 1 & 0.3 & 0.1 \\ 0.2 & -0.7 & -0.7 \\ 0.1 & -0.7 & 0.7 \end{pmatrix},$$

$$V = \begin{pmatrix} 0.02 & -0.1 & -0.91 & -0.39 \\ 0.24 & -0.96 & 0.07 & 0.09 \\ 0.84 & 0.24 & 0.02 & -0.06 \\ 0.45 & -0.02 & 0.41 & -0.94 \end{pmatrix}.$$

■

Labeling the left and right singular vectors as $U =: [u_1, \ldots, u_m]$, $V =: [v_1, \ldots, v_n]$, we obtain a polyadic representation

$$A = \sum_{j=1}^{r} s_j u_j v_j^T,$$

where $r := \max_k s_k > 0$. It may easily be verified that $r = rank\ A$, i.e., equal the minimum number terms in a polyadic representation of A or, equivalently, the dimension of the range of A. It is thus also obvious that $\{u_1, \ldots, u_r\}$ is an

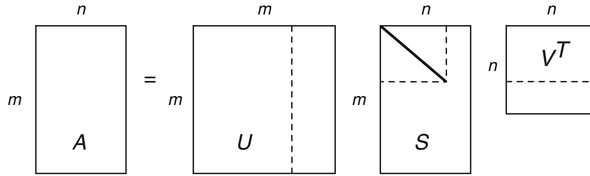


Fig. 8.2 Illustration of the singular value decomposition of a matrix A. The *dotted lines* indicate the borders of the submatrices corresponding to a truncated SVD under the assumption that A be rank deficient. Please note that all entries of S other than those indicated by the diagonal *solid line* are zero

orthogonal basis for the range of A, and $\{u_{r+1}, \ldots, u_m\}$ for the orthogonal complement thereof. Likewise, $\{v_{r+1}, \ldots, v_n\}$ span the null space of A, and the remaining right singular vectors its orthogonal complement.

With regard to the solution of (8.8), the decisive tool is the *truncated singular value decomposition*. It can be shown that the rank-k matrices

$$A_k := \sum_{j=1}^{k} s_j u_j v_j^T = [u_1, \ldots, u_k]\left(\delta_{ij} s_j\right)[v_1, \ldots, v_k]^T =: U_k S_k V_k^T, k = 1, \ldots, r \tag{8.14}$$

provide optimal rank-k approximations to our matrix A in terms of $\|\cdot\|_F$. In fact, the subsequent stronger result obtains.

Theorem 8.1 (cf. Theorem 7.4.51 and Example 7.4.52 in [HJ85]) *Let* $\|\cdot\|$ *be a unitarily invariant norm,* i.e., $\|A\| = \|QAT\|$ *for any* A *and unitary* Q,T. *Then we have*

$$\min_{B \in \Re^{m \times n}, \, rank \, B = k} \|A - B\| = \|A - A_k\|.$$

In particular, this holds for $\|\cdot\|_F$ *and* $\|\cdot\|_2^*$, i.e., *the matrix norm induced by the Euclidean norm.*

As an immediate consequence, this insight provides us with a solution of the approximation problem at hand.

Corollary 8.1 *An optimal solution of* (8.8) *is given by* $X := U_d$ *and* $Y := S_d V_d^T$.

The (truncated) SVD is illustrated by Fig. 8.2.

Example 8.4 For our web shop example 8.1, we obtain for a rank-1 approximation immediately from our SVD

$$X = \begin{pmatrix} 1 \\ 0.2 \\ 0.1 \end{pmatrix}, Y = (11.49)\begin{pmatrix} 0.02 \\ 0.24 \\ 0.84 \\ 0.45 \end{pmatrix}^T = \begin{pmatrix} 0.23 & 2.76 & 9.65 & 5.17 \end{pmatrix},$$

which coincides with the factorization in Example 8.2.

We obtain for a rank-2 approximation

$$X = \begin{pmatrix} 1 & 0.3 \\ 0.2 & -0.7 \\ 0.1 & -0.7 \end{pmatrix},$$

$$Y = \begin{pmatrix} 11.49 & 0 \\ 0 & 6.88 \end{pmatrix} \begin{pmatrix} 0.02 & -0.1 \\ 0.24 & -0.96 \\ 0.84 & -0.02 \\ 0.45 & -0.02 \end{pmatrix}^T = \begin{pmatrix} 0.23 & 2.76 & 9.65 & 5.17 \\ -0.69 & -6.6 & 1.65 & -0.14 \end{pmatrix}$$

and

$$A_k = XY = \begin{pmatrix} 0.02 & 0.78 & 10.14 & 5.13 \\ 0.53 & 5.17 & 0.78 & 1.13 \\ 0.51 & 4.9 & -0.19 & 0.62 \end{pmatrix}$$

already provides a fairly good approximation to A. ■

This terrific result tells us that the computation of a solution of (8.8) may be reduced to computing a truncated singular value decomposition of A. It should be clear from the above derivation that this problem, in turn, may be essentially solved by computing a truncated spectral decomposition of the Gram matrix A^TA. Calculation of spectral decompositions of symmetric and positive definite matrices, fortunately, is a well-understood domain of numerical linear algebra. In particular, this problem may be solved within polynomially bounded time in terms of dimensionality and desired accuracy.

The bad news is that the complexity of state-of-the-art solvers increases with the number of columns of A. Hence, these methods are not suitable for realtime computation.

8.3.2 Incremental Computation of the Singular Value Decomposition

In what follows, we shall present an approach to incremental computation of the SVD. In a nutshell, we shall address the following question: given a matrix A and a vector a, how can we express the SVD of $[A,a]$ in terms of that of A? The subsequent lemma, which summarizes the reasoning of Brand [Bra06, Bra03, Bra02, GE94], is crucial.

Lemma 8.1 *Let $A \in \Re^{m \times n}, a \in \Re^m$. Furthermore, let $A = U_r S_r V_r^T$, where $r \geq \operatorname{rank} A$, be a full-rank truncated SVD. Let $U := U_r$, $S := S_r$, and $V := V_r$. Then we have*

$$[A, a] = \left[U, \frac{a_\perp}{\|a_\perp\|}\right] \begin{bmatrix} S & U^T a \\ 0^T & \|a_\perp\| \end{bmatrix} \begin{bmatrix} V^T & 0 \\ 0^T & 1 \end{bmatrix} \tag{8.15}$$

where $a_\perp := (I - UU^T)a$.

The proof is by a straightforward evaluation of the right-hand side of (8.15).

Without loss of generality, we shall henceforth assume that $a \neq 0$. The simplest case obtains if $U^T a = 0$, that is, a lives in the orthogonal complement of the range of A. Then, up to a permutation, (8.15) is already an SVD of $[A,a]$ and we are done. Now let us assume that $U^T a \neq 0$. For the sake of simplicity, we may safely neglect the case where $a_\perp = 0$ since a slight and obvious modification of the subsequent argumentation will do the trick. Under this assumption, both of the matrices

$$\widetilde{U} := \left[U, \frac{a_\perp}{\|a_\perp\|}\right], \widetilde{V} := \begin{bmatrix} V^T & 0 \\ 0^T & 1 \end{bmatrix}$$

have orthogonal columns. Hence, if we are given a full-rank truncated SVD

$$\widetilde{S} := \begin{bmatrix} S & U^T a \\ 0^T & \|a_\perp\| \end{bmatrix} = \overline{U}\,\overline{S}\,\overline{V}^T$$

then the matrices $\hat{U} := \widetilde{U}\overline{U}$ and $\hat{V} := \widetilde{V}\overline{V}$ have orthogonal columns. Thus, the desired full-rank truncated SVD is given by

$$[A, a] = \hat{U}\overline{S}\hat{V}^T$$

and we are done. Therefore, it all comes down to computing a full-rank truncated SVD of $\widetilde{S}$. Unfortunately, the efficient computation of an SVD of $\widetilde{S}$ is left open in Brand's papers. We therefore outline the approach devised by Paprotny [Pap09].

Fortunately, it turns out that the special structure of this matrix may be exploited for efficient computation: we have

$$\widetilde{S}\widetilde{S}^T = \begin{bmatrix} SS^T & 0 \\ 0^T & 0 \end{bmatrix} + zz^T,$$

where

$$z := \begin{bmatrix} U^T a \\ \|a_\perp\| \end{bmatrix}.$$

Hence, $\widetilde{S}\widetilde{S}^T$ is a rank-1 modification of a diagonal matrix. Again, the computation of a spectral decomposition of matrices of this type is well understood in numerical linear algebra. A sophisticated approach presented in [GE94] relies crucially on the solution of a so-called secular equation. One of the most fundamental insights of

finite-dimensional spectral theory is that the eigenvalues of a matrix A are, respecting multiplicities, precisely the roots of the *characteristic polynomial*

$$\chi_B(\lambda) := \det(A - \lambda I).$$

It turns out that the characteristic polynomial of a rank-1 modification of a diagonal matrix has a closed-form representation.

Proposition 8.3 Let $\Lambda := \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$ be a real diagonal matrix and $x \in \Re^n$. Then the characteristic polynomial of $\Lambda + xx^T$ is given by

$$\det\left(\Lambda + xx^T - \lambda I\right) = \det(\Lambda - \lambda I)\left(1 + x^T(\Lambda - \lambda I)^{-1}x\right)$$
$$= \left(\prod_{i=1}^{n}(\lambda_i - \lambda)\right)\left(1 + \sum_{i=1}^{n}\frac{x_i^2}{\lambda_i - \lambda}\right).$$

A proof may be found in, e.g., [GE94].

For the sake of simplicity, our discussion is restricted to the case where:

1. $\lambda_1, \ldots, \lambda_n$ are distinct.
2. The spectra of Λ and $\Lambda + xx^T$ are disjoint.

For a more general treatment, please consult [BNS78].

If the second of the above conditions holds, then $\widetilde{\lambda}$ is an eigenvalue of $\Lambda + xx^T$ if and only if

$$1 + x^T\left(\Lambda - \widetilde{\lambda}I\right)^{-1}x = 0. \tag{8.16}$$

Hence, the eigenvalues of the rank-1 modification may be obtained by solving the secular equation (8.16). To do so, we may exploit the following insight.

Proposition 8.4 (cf. [GE94]) The eigenvalues $\widetilde{\lambda}_1 \geq \ldots \geq \widetilde{\lambda}_n$ of $\Lambda + xx^T$ satisfy the interlacing property

$$\widetilde{\lambda}_i \geq \lambda_i \geq \widetilde{\lambda}_{i-1}, \;\; i = 2, \ldots, n.$$

By virtue of this observation, we are given intervals in which precisely one eigenvalue is located. How should we proceed to solve (8.16)? The most straightforward way consists in deploying a bisection method. Since Proposition 8.4 provides suitable initializations, such a method is guaranteed to converge. The rate, however, is only q-linear. A method exhibiting q-quadratic convergence is Newton–Raphson. Unfortunately, we are not in a position to guarantee convergence thereof because it may "leap" over the singularities and thus out of the search intervals in an early stage. A more sophisticated method, again, has been proposed in [BNS78]: the rational function on the right-hand side of (8.16) is iteratively approximated by low-degree rational functions the roots of which are available in

closed form. The method can be shown to converge locally q-quadratically if the initialization overestimates the sought-after root. To obtain such an initial estimate, one may perform a few steps of bisection.

The naïve way to compute the corresponding eigenvectors is by solving the sequence of null space problems

$$\left(\Lambda + xx^T - \widetilde{\lambda}\right)u_i = 0, \|u_i\| = 1, i = 1, \ldots, d.$$

Since the computed eigenvalues are inexact, this method is numerically unstable. In particular, it may lead to severe loss of orthogonality. A more sophisticated alternative is based on the following result, which is due to Löwner.

Proposition 8.5 [GE94] *Let $\lambda_1, \ldots, \lambda_n, d_1, \ldots, d_n \in \Re$ satisfy the interlacing property*

$$\lambda_{i+1} > d_{i+1} > \lambda_i, i \in \underline{n-1}.$$

Furthermore, let $D :=$ diag$(d_1, \ldots, d_n)$. Then $\lambda_1, \ldots, \lambda_n$ are the eigenvalues of

$$D + bb^T$$

for all $b \in \Re^n$ satisfying

$$|b_j| = \prod_{k=1}^{n} (\lambda_k - d_j) \Big/ \prod_{k=1, k\neq j}^{n} (d_k - d_j), j \in \underline{n}.$$

We proceed by considering the approximate eigenvalues as exact eigenvalues of a slightly perturbed problem. The above result enables to compute the eigenvectors thereof analytically.

For a detailed description of the entire procedure, we refer the reader to [Pap09].

The main SVD procedure is summarized in Algorithm 8.1. We leave out the calculation of the SVD of $\widetilde{S}$ since it is quite complex and there exist different approaches, one of which was presented here.

Algorithm 8.1: SVD update

Input: matrices U and V of left and right singular vectors of A, matrix S of singular values, new vector a

Output: updated matrices $\hat{U}$ and $\hat{V}$ of left and right singular vectors, matrix of singular values $\overline{S}$ of $[A,a]$

1: Calculate SVD of $\widetilde{S} := \begin{bmatrix} S & U^T a \\ 0^T & \|a_\perp\| \end{bmatrix} = \overline{U}\,\overline{S}\,\overline{V}^T$

2: Calculate $\hat{U} := \left[U, \frac{a_\perp}{\|a_\perp\|}\right]\overline{U}$ and $\hat{V} := \begin{bmatrix} V^T & 0 \\ 0^T & 1 \end{bmatrix}\overline{V}$

Now *experience* has shown that the update algorithm 8.1 in general can also be applied successfully for truncated SVDs of any rank r including small ones. However, unlike as for the truncated full-rank SVD presented there, the resulting SVD may not be optimal, i.e., it may lead to decompositions different from (8.14).

8.3.3 Computing Recommendations

There remains the question how to compute the recommendations via the truncated singular value decomposition (8.14). Let $A_k = U_k S_k V_k^T$ be the rank-k SVD of the previous session data and $a \in \Re^m$ be the current session vector, i.e., the vector containing the rewards of the products of the current session.

One way is to proceed as follows. In each step of the session, we add the current session vector $a \in \Re^m$ to the (fixed) SVD $A_k = U_k S_k V_k^T$ and execute one incremental step of the singular value decomposition as described in the previous section, i.e.,

$$\widetilde{A} := [A_k, a] \approx \widetilde{U}_k \widetilde{S}_k \widetilde{V}_k^T = \left[A_k^{'}, a_k\right] =: \widetilde{A}_k. \tag{8.17}$$

Now we use the updated session vector $a_k \in \Re^m$ and recommend the products of the row numbers with the largest entries of a_k, provided they are not already included in a. After the termination of the current session, we set $A_k := [A_k^{\prime}, a_k]$. Then we proceed in the same way with the next session vector.

Thus, in each step we conduct a low-rank approximation of the whole data matrix and use its generalization for prediction in the current session – by means of the last column. For the special case of a full-rank SVD, i.e., $k = \text{rank}\, A$, we would always obtain $a_k = a$ and would not be able to provide a meaningful prediction.

The described procedure essential complies with the typical approach used in literature about factorization for recommendations, although there learning and evaluation are mostly carried out separately, i.e., offline. At this, the transactions of each session (in literature mostly users) are subdivided into two disjoined training and test sets such that for each session vector, it holds that $a = a_{train} + a_{test}$. This way the data matrix A is split into a training matrix A_{train} and a test matrix A_{test}. Next a factorization is applied to A_{train} and thereafter evaluated on A_{test}. The adaptive behavior of our incremental SVD, in contrast, allows the more flexible procedure described above, which, however, is still computational intensive. Thus, we are looking for further alternatives.

The aforementioned "classical" offline approach has in practice the additional disadvantage that it can only be applied to existing users (in our case even session!) which already have a transaction history. To overcome this problem, the following approach is pursued, especially in older literature. First, for a the feature vector,

$$f_k^a = S_k^{-1} U_k^T a$$

is computed. Now in the feature space, we are searching for that feature vector f_k^b which is closest to f_k^a and recommend the highest-rewarded products of its

associated session vector b. This approach basically corresponds to classical CF, with the only difference that for the storage and evaluation of user profiles instead of the m-dimensional initial space of our products, now the synthesized k-dimensional feature space is used. This reduces complexity on the one hand, and at the same time a generalization of profiles with the associated increase of quality takes place. Nevertheless, this approach is still complicated and mathematically difficult to handle.

A better approach is based on projections. Therefore we first want to compute the incremental ansatz (8.17) in a more efficient way. It can be shown that the following holds:

$$U_k U_k^T A V_k V_k^T = U_k S_k V_k^T = A_k. \tag{8.18}$$

Because of the elementary relations $V_k^T V_k = I$ and $U_k^T U_k = I$, it follows from (8.5) and (8.4) that both $V_k V_k^T$ as well as $U_k U_k^T$ are orthoprojectors into the space of their corresponding singular vector bases. Consequently, the rank-k SVD can be represented as a concatenated projection into the spaces of left and right singular vector bases.

The projection (8.18) can be accomplished even easier.

Proposition 8.6 *The following properties hold:*

$$A_k = A V_k V_k^T = U_k U_k^T A$$

Proof For $A_k = A V_k V_k^T$, we just need to replace $A V_k$ by $U_k S_k$ according to (8.11):

$$A V_k V_k^T = U_k S_k V_k^T = A_k.$$

The deduction of $A_k = U_k U_k^T A$ is again based on (8.11), in conjunction with (8.10) for the decomposition of the Gram matrix by the right singular values, which constitute its eigenvectors:

$$U_k U_k^T A = U_k S_k^{-1} V_k^T A^T A = U_k S_k^{-1} V_k^T V S^2 V = U_k S_k V_k^T = A_k \qquad \square$$

From Proposition 8.6, it follows that by means of the *right-projection approximation*, A_k can be computed solely via the right singular values:

$$A_k = A V_k V_k^T. \tag{8.19}$$

Similarly, we can apply the *left-projection approximation* to compute A_k only via the left singular values:

$$A_k = U_k U_k^T A. \tag{8.20}$$

Beyond its simplicity the left-projection approximation has another advantage. Since the orthoprojector $U_k U_k^T$ is multiplied from left, due to

$$\left[A_k^{'}, a_k\right] = \widetilde{A}_k = \widetilde{U}_k \widetilde{U}_k^T \widetilde{A} = \widetilde{U}_k \widetilde{U}_k^T [A_k, a] = \left[\widetilde{U}_k \widetilde{U}_k^T A_k, \widetilde{U}_k \widetilde{U}_k^T a\right].$$

the property

$$a_k = \widetilde{U}_k \widetilde{U}_k^T a,$$

holds. This means that for the calculation of the updated session vector a_k, only the current session vector a is required. Thus, we can use this approach for arbitrary session vectors without updating the left singular vector $\widetilde{U}_k$ each time for a. This enables us to use an existing rank-k SVD without updating, i.e., without learning, for the prediction of new sessions.

Therefore, we now generally want to apply left-projection approximation for SVD-based calculation of recommendations. We get

$$a_k = U_k U_k^T a \tag{8.21}$$

and thus recommend the highest-rewarded products of the session vector a_k. It is so easy!

We will give a descriptive interpretation: the transposed left singular vector matrix U_k^T provides a mapping into the k-dimensional feature space resulting in a profile vector of our session. Then it is mapped by U_k back into the product space. For the special case of a full-rank SVD, i.e., $k = \text{rank}\, A$, the left singular vector matrix $U_k = U$ is unitary, and thus we get again

$$a_k = UU^T a = a,$$

what, of course, would be little helpful. The essence behind the projection approach is that we map our session vector by a low-rank approximation onto its "generalized profile" and then assign "characteristic rewards" to this profile. Hence, this procedure corresponds to the previous one but is much easier.

In a nutshell, (8.21) allows the direct computation of recommendations for arbitrary sessions. It is noteworthy that here the matrices of the singular values S_k and right singular vectors V_k are not required at all! This makes our approach in every aspect more computational efficient than the truncated SVD.

Finally we mention that the truncated SVD also gives rise to a nice factorized version of the item-to-item collaborative filtering described in Sect. 8.2. Thus, we are looking for a factorized version $\overline{S}_k$ of the similarity matrix $\overline{S} = \overline{A}\,\overline{A}^T$ over all products.

Obviously, it is obtained by $\overline{S}_k = \overline{A}_k \overline{A}_k^T$ where $\overline{A}_k$ is the rank-k SVD $A_k = U_k S_k V_k^T$ normalized along all of its columns. Introducing the factor matrix $L_k := U_k S_k$, we can express the inner products $A_k A_k^T$ through L_k as $A_k A_k^T = U_k S_k V_k^T V_k S_k U_k^T$

Table 8.2 Comparison of prediction qualities: adaptive SVD with variable rank

Rank	Clicks (%)	Baskets (%)	Buys (%)
5	6.3	11.14	12.07
10	7.59	12.1	14.22
14	8.07	13.72	15.35
15	8.09	13.37	15.39
16	7.98	13.72	15.44
18	8.38	13.78	15.33
20	7.67	13.06	15.28
25	7.78	12.63	14.77

$= (U_k S_k)(U_k S_k)^T = L_k L_k^T$. Again, normalizing L_k along its columns leads to $\overline{L}_k$, and we can calculate $\overline{S}_k$ very easy without requiring the right singular vectors V_k:

$$\overline{S}_k = \overline{L}_k \overline{L}_k^T.$$

Here, for the full-rank SVD, we arrive at the original ITI-CF similarity matrix $\overline{S}$. The efficient computation of maximum values of a low-rank matrix is described in Sect. 8.6.

Example 8.5 For the assessment of the prediction quality, we use the methodology of Sect. 4.4 with one difference: a product is counted as correctly predicted not only when it directly follows a recommendation but also if it appears in the remaining course of the session. This follows the logic of the prediction method because for transactions within a session, their sequential order is ignored.

The transaction log file used contains 695,154 transactions. For the test all sessions with less than 3 transactions have been removed, and only products of the core shop have been considered (because the products of the remaining assortment only rarely occur in the transactions). This resulted in 23,461 remaining sessions. The number of different products is 558. We use the following mapping:

$$click \mapsto 1, basket \mapsto 10, order \mapsto 20$$

for a reward function.

In Table 8.2 the results of the simulation with adaptive singular value decomposition of Algorithm 8.1 for variable ranks on the described data of one day are summarized.

From Table 8.2 it follows that rank 16 delivers the best prediction quality concerning the orders, whereas baskets and clicks are slightly better predicted by a model with rank 18.

Figure 8.3, which is a graphical interpretation of Table 8.2, clearly shows the expected graphs of under- and over-fitting. If the rank is too small, the model does not approximate the data good enough, and this results in a lower prediction rate (under-fitting); a rank that is too high approximates the noise too exactly and also reduces the prediction rate (over-fitting).

The results may suggest good chances for the application of the method in REs although we will see later that reality is more complex. ■

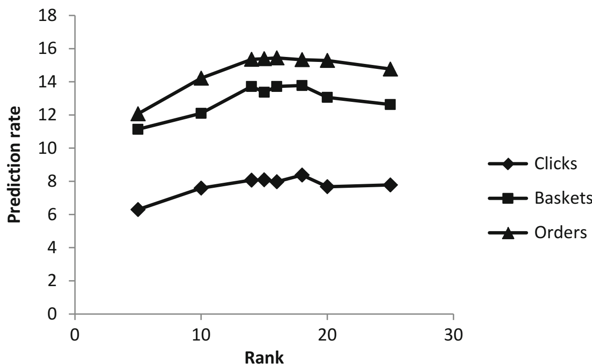


Fig. 8.3 Graphical comparison of prediction rates: adaptive SVD with variable rank

8.4 More Matrix Factorizations

8.4.1 Lanczos Methods

Lanczos methods are projection methods and are types of Krylov space methods. Although they are not adaptive, we shall yet address them in more detail, as they are, arguably, the most efficient solvers for eigen- and singular value decompositions and, apart from that, may provide an alternative to the projection approximations from Sect. 8.3.3, which is even more efficiently computable. Furthermore, the Lanczos process may be deployed for efficient computation of an initial rank-k SVD on historical data, which may then serve as a starting point for our adaptive SVD. There is a vast amount of Lanczos-type methods. We shall restrict ourselves to the symmetric Lanczos algorithm.

As we have shown in Sect. 8.3.1, for an arbitrary matrix A, we may establish the Gramian matrix $G = A^T A$, which is symmetric and positive semidefinite, so as to compute an SVD (8.13) of A through an eigenvalue decomposition (EVD, i.e., spectral decomposition (8.9)) of G by means of (8.11) and (8.12). This, of course, similarly applies to the truncated SVD (8.14). Thus, we need an efficient method for the computation of a truncated EVD of symmetric positive semidefinite matrices at the core. This requirement is best met by the symmetric Lanczos algorithm, as it is especially suitable for large eigenvalues, which are needed for the truncated SVD.

Let $G \in \mathrm{S}^{n \times n}_{\succ = o}$ be a symmetric positive definite matrix of order n. Then we seek after an optimizer of

$$\max_{X \in \Re^{n \times k}, X^T X = I} tr\left(X^T G X\right). \tag{8.22}$$

It is well known that V_k is an optimizer of (8.22) if and only if $ranV_k$, i.e., the *range* or *image* of V_k is an invariant subspace of G with respect to its k-largest eigenvalues. Thus, we solve (8.22) to determine a matrix the columns of which are eigenvectors of G corresponding to its k-largest eigenvalues. This matrix coincides

with the desired matrix V_k of right singular vectors. As described at the outset, given this matrix, the corresponding matrices of singular values S_k and left singular vectors U_k may easily be established.

A *Galerkin method* yields an approximate solution of (8.22) by imposing the additional constraint $ranX \subset \mathrm{Q} \subset \Re^{n}$ for a suitable subspace Q satisfying $k \leq \dim Q = l << n$. It can be shown that this amounts to solving

$$\max_{Y \in \Re^{l \times n},\, Y^T Y = I} tr\left(Y^T Q^T G Q Y\right) \tag{8.23}$$

for some Q satisfying $ran\ Q = \mathrm{Q}$, $Q^T Q = I$.

A *Krylov space method* recursively computes nested orthogonal bases $Q_1, \ldots, Q_l$ of subspaces $\mathrm{Q}_1 \subset \ldots \subset \mathrm{Q}_l$ yet to be specified and a sequence of approximate solutions X_l, $k \leq l \leq n$ of (8.22) given by $X_l := Q_l Y_l$, where Y_l is an optimizer of (8.23) for $Q = Q_l$. For an initial vector q_1, the subspaces are established as follows:

$$\mathrm{Q}_l := K_l(G, q_1), l \in \{1, \ldots, n\},$$

where

$$K_l(G, q_1) = range\left\{q_1, Gq_1, G^2 q_1, \ldots, G^{l-1} q_1\right\}$$

is called lth *Krylov space* of G and q_1. The *Lanczos process* is a recursive procedure for the computation of the basis vectors q_l, which are also referred to as *Lanczos vectors*.

Algorithm 8.2 Lanczos process

Input: G, q_1, l

Output: $q_1, \ldots, q_{l+1}$, $\alpha_i' s$, $\beta_i' s$

1: $\beta_1 := 0$, $q_0 = 0$
2: **for** $i = 1, \ldots, l$ **do**
3: $w_i := Gq_i - \beta_i q_{i-1}$
4: $\alpha_i := \langle w_1, q_i \rangle$
5: $w_i := w_i - \alpha_i q_i$
6: $\beta_{i+1} := \|w\|_2$
7: **if** $\beta_{i+1} = 0$ **then**
8: **stop**
9: **end if**
10: $q_{i+1} := w_i / \beta_{i+1}$
11: **end for**

With $Q_l := [q_1, \ldots, q_l] \in \Re^{n \times l}$, it may easily be shown that

$$Q_l^T G Q_l = T_l = \begin{bmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_{l-1} & \alpha_{l-1} & \beta_l \\ & & & \beta_l & \alpha_l \end{bmatrix}. \tag{8.24}$$

An eigenvalue of T_l is called *Ritz value,* and, for a corresponding eigenvector y_l, $x_l := Q_l y_l$ is called *Ritz vector*. With l increasing, more and more Ritz values and vectors converge to eigenvalues and vectors of G. Thus, the eigenvectors y_l yield the desired solution of (8.23), i.e., $Y_l := [y_1, \ldots, y_l]$, from which we obtain the approximate solution of (8.22). Due to the simplicity of the tri-diagonal matrix T_l, the eigenproblem (8.23) may be solved efficiently, e.g., by simple subspace iteration procedures.

Although the demonstrated Lanczos method looks fairly simple, it bears several difficulties. These include the problem of orthogonality. In practice (i.e., finite precision arithmetic), the theoretical orthogonality of the computed Lanczos vectors q_l is lost at an early stage due to inevitable rounding errors. Hence, many approaches to *re-orthogonalization* of Lanczos vectors have emerged. The easiest approach consists in orthogonalizing each newly obtained vector with respect to the previous basis. This is accomplished by adding the following line immediately below line 5 of the pseudo-code:

$$w_i := w_i - \sum\nolimits_{j=1}^{i-1} < w_i, q_j > q_j.$$

This additional step of complete re-orthogonalization increases the computational load by $O(l^2 n)$, but ensures that all Lanczos vectors be numerically mutually orthogonal and thus all subsequent processes be stable. Since, with regard to our factorization, we are interested only in the $k \ll n$ largest eigenvalues, l is not too large in practice (at most some 100). Hence, complete re-orthogonalization does not cause any trouble at all.

Another difficulty of the Lanczos method is that it may not find all eigenvalues, even if the computation is carried out in exact arithmetic. This problem occurs predominantly for small eigenvalues. To remedy this problem as well, many elaborate approaches have been developed. With respect to large eigenvalues, however, which we are predominantly interested in, the Lanczos process works in a fairly stable fashion. Altogether, we see that for our application, it is not necessary to put much effort into stabilization.

Thus, we may use the Lanczos method to compute the truncated SVD, which, in turn, provides the right- or left-projection approximation (8.19) or (8.20), respectively, for determining the recommendations.

Even more, in [CS09], Chen and Saad have shown how the Lanczos method may be used for an even easier computation of the projections (8.19) and (8.20). To this

end, we resort to the matrix of Lanczos vectors Q_k and consider their left-projection approximation

$$A_k^L = Q_k Q_k^T A.$$

Correspondingly, if using the covariance (rather than the Gramian) matrix $C = AA^T$ and its matrix of Lanczos vectors $\overline{Q}_k$, we obtain the associated right-projection approximation

$$\overline{A}_k^L = A\overline{Q}_k \overline{Q}_k^T.$$

Now Chen and Saad were able to show that for arbitrary vectors $b \in \Re^m$, the matrix–vector operations $s_k := A_k^L b$ and $t_k := \overline{A}_k^L b$ provide good approximations to $A_k b$, since their sequences $\{s_i\}$ and $\{t_i\}$ exhibit rapid convergence to Ab in terms of the leading left singular directions of A. As the goal of data compression consists precisely in preservation of accuracy with respect to these directions, the Lanczos projection provides a good alternative to singular value projection.

As for computation of the updated session vector through left-projection approximation by means of Lanczos vectors, we thus obtain

$$a_k = Q_k Q_k^T a. \tag{8.25}$$

Compared with its counterpart for singular vectors (8.21), (8.25) is even easier to compute, since we may abstain from a Ritz step, i.e., the solution of the eigenproblem (8.24), outright. As we shall see in numerical tests, the practical results, too, of approximation by means of Lanczos vector projection are hardly worse than those of singular vector projection.

8.4.2 RE-Specific Requirements

As regards usage for reinforcement learning, of course, the factorization of transition probabilities $p_{ss'}^{(a)}$ is of particular interest. Here, we shall ignore the actions a by considering either only the unconditional probabilities $p_{ss'}$ or, according to Assumption 5.2, only the conditional probabilities of a transition to the recommended product $p_{ss_a}^a$. For the sake of simplicity, we shall identify the two cases with $p_{ss'}$. The reason is that matrix factorization makes sense only for two indices s and s'. The case of an additional factorization with respect to the actions a, that is, $p_{ss'}^a$, will be treated later in the scope of tensor factorization.

Thus, we would like to factorize the matrix of transition probabilities $P = \left(P_{ss'}\right)_{s,s' \in S}$.

Theorem 8.1 states that the truncated SVD provides the best matrix factorization with respect to quality of approximation. In general, however, this goes along with a loss of stochasticity of the matrix of transition probabilities $\widetilde{P} := P_k$.

Hence, we require, on one hand, that the factorized transition probabilities satisfy

$$0 \leq \widetilde{p}_{ss'} \leq 1, \forall s, s' \in S. \tag{8.26}$$

On the other hand, the truncated SVD also violates the row sum condition (3.2). Thus, we furthermore require

$$\sum_{s'} \widetilde{p}_{ss'} = 1 \, \forall s \in S. \tag{8.27}$$

It is possible to impose the conditions (8.26) and (8.27) by subsequent normalization of the factorization (8.14). Although this may look somewhat artificial at the first glance, one must take into account that violation of stochasticity by a rank-k approximation has no deeply rooted causes with respect to content. And the truncated SVD is still the best approximation.

An alternative remedy consists in using specific nonnegative matrix factorizations, which we shall consider in the next section.

8.4.3 Nonnegative Matrix Factorizations

In a nonnegative matrix factorization (NMF), we consider the problem (8.1) for a nonnegative matrix A and require that the factor matrices $X, Y,$ as well, be nonnegative. With regard to the Frobenius norm, we obtain the following problem statement:

$$\min_{X \in \Re^{m \times k}, \, Y \in \Re^{k \times n}} \|A - XY\|_F^2, X, Y \geq 0. \tag{8.28}$$

As compared with the SVD-based approach, this gives rise to certain advantages for nonnegative A. Most importantly, the matrices X,Y have a clear interpretation – as opposed to the SVD. If we consider the columns of the left factor X as basis vectors, i.e.,

$$X = [x_1| \ldots |x_k],$$

then the columns of A,

$$A = [a_1| \ldots |a_m],$$

are approximately expressed in terms of the basis $B_X := \{x_1, \ldots, x_k\}$:

$$a_j \approx x_1 y_{1j} + \ldots + x_k y_{kj}, 1 \leq j \leq n.$$

Due to nonnegativity of X,Y, large x_{ij}'s indicate that the jth basis vector essentially represents the value of the ith element of a_j. Correspondingly, large values of y_{ij}'s indicate a major contribution of the ith basis vector to the element a_j.

As compared with SVD, of course, we need to trade off the advantages against the disadvantages: NMF typically exhibits a worse rank-k approximation than SVD. Moreover – as opposed to SVD – the approximation is not unique. While the left singular vectors of SVD are mutually orthogonal, this does not hold for the basis vectors x_i of NMF in general.

A major drawback is the lack of efficient computational methods for NMF that are suitable for high-dimensional problems. While there are technically mature standard procedures for SVD computation, like the Lanczos method, as well as adaptive procedures like that of Brand's, and their convergence is proven, methods for NMF are still in their infancy and poorly understood.

Even though the problem (8.28) is convex in each of X and Y, it is not convex in both variables taken together, which renders determining a globally optimal solution difficult. For most existing algorithms, convergence to a locally optimal solution has been shown at best.

Most NMF optimization procedures follow the EM (*expectation-maximization*) principle in alternatingly fixing one of the variables and obtaining a new iterate by optimizing with respect to the other variable.

A typical procedure is the ALS algorithm (*alternating least squares*), which is considered to be an important practical tool for computation of an NMF. It should be noted that we may introduce a column normalization immediately after step 6, so as to ensure that the column sums up to 1. With regard to the case of a factorization of transition probabilities discussed in the previous section, we thus automatically fulfill the conditions (8.26) and (8.27). Hence stochasticity of the factorized matrix is preserved.

Algorithm 8.3 ALS

Input: positive matrix $A \in \Re^{m \times n}$

Output: positive factor matrices $X \in \Re^{m \times k}, Y \in \Re^{k \times n}$
1: initialize X with small random values
2: **repeat**
3: $\quad Y := (X^T X)^{-1} X^T A = X^+ A$
4: $\quad Y := [Y]_+$
5: $\quad X := AY^T (YY^T)^{-1} = AY^+$
6: $\quad X := [X]_+$
7: **until** convergence

Despite its good practical performance, we cannot guarantee even local convergence of ALS. There is a vast variety of further algorithms for NMF. At present, researchers are working flat out to make these methods suitable for high-dimensional problems. The results, however, are still hard to assess.

Table 8.3 Comparison of prediction qualities and error norms of SVD, LVP, and NMF with variable rank

	SVD				LVP				NMF			
k	p_1	p_3	e_F	t	p_1	p_3	e_F	t	p_1	p_3	e_F	t
2	1.98	4.47	4.64	41	0.77	2.59	4.73	0	1.88	4.28	4.64	1
5	2.51	5.61	4.50	48	2.43	5.31	4.59	0	2.22	6.37	4.51	2
50	5.35	9.55	3.44	199	5.14	9.54	3.74	1	5.14	8.56	3.52	26
100	5.75	10.06	2.73	378	5.69	10.00	3.13	2	5.50	9.22	2.98	64
200	6.27	10.69	1.88	757	6.12	10.49	2.19	7	6.09	10.31	2.36	185
500	6.35	10.32	0.74	5168	6.35	10.35	0.75	38	6.35	10.51	2.53	752
558 (full)	6.37	10.33	0									

8.4.4 Experimental Results

In the following, we shall apply the above-described factorization procedures to predict product transitions and evaluate the results. To be able to employ the nonadaptive procedures from this chapter, we divide each of the data sets into a training and a test set.

We shall first address factorization of the transition probabilities P. To this end, we first compute the matrix P from the training data. Then, we factorize it and subsequently use the factorized matrix P_k to predict the considered products of the test set. To do so, we traverse all sessions of the test set step by step and recommend for each considered product s the product with the highest transition probability in P_k, i.e., $\arg\max_{s'} (p_k)_{ss'}$, which we compare with the immediate successor product. Correspondingly, we also recommend the three strongest products to inquire the case of multiple recommendations. This so corresponds to the Markov approach from the previous chapters.

Example 8.6 We consider the shop from Example 8.5, while taking all transactions and its entire product assortment (2671 products) into account. The training and test set both consist of 134,832 sessions.

For the factorization, we use each of the three methods presented in this section: the truncated SVD (8.14), computed by means of the Lanczos Algorithm 8.2 with Ritz projections; the Lanczos vector projection (LVP), i.e., Algorithm 8.2 with left-projection approximation (8.25); and the NMF according to the ALS Algorithm 8.3. The latter has always been carried out with 100 iterations.

The result is displayed in Table 8.3. Here, k denotes the rank, p_1 and p_3 are the rates of correct prediction for 1 and 3 recommendations, respectively, $e_F := \|P - P_k\|_F$ the Frobenius norm of the approximation error, and t the computing time (in seconds). The last row contains the results for usage of the complete matrix P.

The result is rather disappointing and may be summarized as follows: none of the three approaches to factorization of transition probabilities is really sensible. A ridiculously high rank is necessary to achieve a quality of prediction that is comparable with that attained when using the nonfactorized matrix P.

Table 8.4 Comparison of prediction rates of an LVP with variable rank with respect to the directly succeeding product and the remainder of the session

	Immediate		Remainder of the session	
k	p_1	p_3	p_1	p_3
1	0.001	0.003	0.47	0.76
5	0.55	2.05	1.28	2.50
50	0.61	2.07	1.48	2.98
100	0.60	1.89	1.40	2.62
200	0.67	1.51	1.37	2.20
500	0.64	1.32	1.34	1.79

Further results are the factorization is at least a useful tool for generating new recommendations. Indeed, the better results of three recommendations as compared to one recommendation simply result from the low-rank approximation's generating more recommendations. Moreover, the SVD turns out to be the best procedure, with respect to not only approximation error but also quality of prediction. The expectation that the nonnegativity inherent to NMF give rise to better prediction results turns out to be false. The deterioration of approximation quality of NMF with increasing rank (at a constant number of iterations) also reveals that the number of iterations of ALS must increase with a growing problem size, which results in bad scaling properties of the method, as the iteration steps themselves are computationally expensive. At the end of the day, LVP turns out to be the best choice, since it exhibits considerably better scaling properties – at an only slightly large approximation error. ■

Example 8.7 Next, we would like to return to computing recommendations according to the profile-based approach from this chapter, in particular from Example 8.1, hence based upon all transactions of the session before the prediction. We shall, however, again, restrict ourselves to prediction of product transitions – consequently, all clicks are endowed with the reward 1, the remainder with 0.

Thus, the training data act as the matrix A. We use the Lanczos vector projection, which has turned out to be very efficient in Example 8.6. Thus, we use Algorithm 8.2 to compute the Lanczos vectors Q_k from A and the left-projection approximation (8.25) to compute recommendations for the test data set.

We use the same data set as in Example 8.6. We again compute one or three recommendations, respectively, and evaluate their prediction rate. Here, we evaluate the recommendations with respect to the immediately following product, i.e., in analogy to Example 8.6, and, additionally, with respect to the entire remainder of the session, in analogy to Example 8.5. The result is displayed in Table 8.4.

The prediction rates of direct product acceptance may be compared to those from Table 8.3. Although we see that the low-rank approximation works in principle (with an optimal rank of approximately 50), the prediction rates are so low that the approach turns out to be practically irrelevant. This is simply due to the fact that few sessions are sufficiently long for the prediction to work well. With respect to the entire remainder of the session, the results are, according to nature, better, but still poor. The reason for the poor prediction rate as compared to Example 8.5 is that

therein, all short sessions have been removed and fewer products (namely, those of the core assortment) are considered. ■

Despite the disappointing results, the question arises of whether the two approaches to factorization, i.e., the Markov chain-based approach and collaborative filtering according to the two previous examples, may be combined in a meaningful way. This, indeed, is possible and will be studied in the course of the chapter in connection with tensor factorization.

Another way is to use techniques different from matrix factorization, like hierarchical decompositions, in order to exploit the structure of the probability matrix and develop corresponding representations based on a small number of parameters. Interesting combinations of both approaches are *hierarchical matrices* (***H**-matrices*) introduced by Wolfgang Hackbusch which rely on local low-rank approximations, i.e., blocks of the matrix are represented in factorized formats. H-matrices are a very powerful technique for matrix compression and a topic of current research. Since it is not a direct factorization technique, we will no further delve into this topic but refer to the literature [Beb08, GH03, Ha99, HK00].

However, there is still another interesting aspect of matrix factorization important for recommendation engines – incomplete data. This brings us to the task of exact matrix completion that has become recently very popular because of some outstanding and surprising results that have been achieved. Motivated by the revolutionary work on *compressed sensing* [CRT06, Don06], a signal processing technique for efficiently reconstructing a signal, some of its pioneers, especially Emmanuel Candes, Benjamin Recht, and Terence Tao, have leveraged basic ideas to the problem of exact matrix completion. The matrix completion problem will be discussed in the next section.

8.5 Back to Netflix: Matrix Completion

In many practical applications, one would like to recover a matrix from a sample of its entries. In case of recommendation engines, the best known example is the Netflix price. Users are given the opportunity to rate movies, but users typically rate only very few movies so that there are very few scattered observed entries of the data matrix. Yet one would like to complete the matrix so that Netflix might recommend titles that any particular user is likely to be willing to order. In the Netflix competition, for each of the users under consideration, a part of her/his ratings was provided in the training set. For evaluation, the remaining movies the user has rated were provided, and the task was to guess her/his actual ratings. The Netflix price was awarded to the recommendation solution of highest prediction rate on the test set. So the Netflix competition constitutes a classical matrix completion problem.

In mathematical terms, the matrix completion problem may be formulated as follows: we again consider a data matrix $A \in \Re^{m \times n}$ which we would like to know as precisely as possible. Unfortunately, the only information about A is a sampled set

of entries A_{ij}, $(i, j) \in \Omega$, where Ω is a subset of the complete set of entries $\underline{m} \times \underline{n}$. Clearly, this problem is ill posed in order to guess the missing entries without making any assumptions about the matrix A.

Now we suppose that the unknown matrix A has low rank. In [CR08], Emmanuel Candes and Benjamin Recht showed that this assumption radically changes the problem, making the search for solutions meaningful. We follow the guidelines of [CR08, CT10].

For simplicity, assume that the rank-r matrix A is $n \times n$. Next, we define the orthogonal projection $P_\Omega : \Re^{n\times n} \to \Re^{n\times n}$ onto the subspace of matrices vanishing outside of Ω as

$$P_\Omega(X)_{ij} = \begin{cases} X_{ij}, & (i,j) \in \Omega, \\ 0, & \text{otherwise}, \end{cases} \tag{8.29}$$

so that the information about A is given by $P_\Omega(A)$. We want to recover the data matrix by solving the optimization problem

$$\begin{array}{ll} \text{minimize} & \operatorname{rank}(X) \\ \text{subject to} & P_\Omega(X) = P_\Omega(A) \end{array}, \tag{8.30}$$

which is, in principle, possible if there is only one low-rank matrix with the given entries. Unfortunately, (8.30) is difficult to solve as rank minimization is in general an NP-hard problem for which no known algorithms are capable of solving problems in practical time for (roughly) $n \geq 10$.

Candes and Recht proved in [CR08] that, first, the matrix completion problem is not as ill posed as previously thought and, second, that exact matrix completion is possible by convex programming. At this, they proposed to replace (8.30) by solving the nuclear norm problem

$$\begin{array}{ll} \text{minimize} & \|X\|_* \\ \text{subject to} & P_\Omega(X) = P_\Omega(A) \end{array}, \tag{8.31}$$

where the *nuclear norm* $\|X\|_*$ of a matrix X is defined as sum of its singular values:

$$\|X\|_* := \sum_i s_i(X). \tag{8.32}$$

Candes and Recht proved that if Ω is sampled uniformly at random among all subset of cardinality p and A obeys a low coherence condition than with large probability, the unique solution to (8.31) is exactly A, provided that the number of samples is

$$p \geq C n^{6/5} r \log n. \tag{8.33}$$

In [CT10] the estimate (8.33) is further improved toward the limit $nr \log n$.

Why is the transition to formulation (8.31) so important? Whereas the rank function in (8.30) counts the number of nonvanishing singular values, the nuclear norm sums their amplitude and, in some sense, is to the rank functional what the convex l_1 norm is to the counting l_0 norm in the area of sparse signal recovery. The main point here is that the nuclear norm is a convex function and can be optimized efficiently via semidefinite programming.

When the matrix variable X is symmetric and positive semidefinite, the nuclear norm of X is the sum of the (nonnegative) eigenvalues and thus equal to the trace of X. Hence, for positive semidefinite unknown, (8.31) would simply minimize the trace over the constraint set

$$\begin{array}{ll} \text{minimize} & \operatorname{trace}(X) \\ \text{subject to} & P_\Omega(X) = P_\Omega(A), \\ & X \succeq 0 \end{array}$$

which is a semidefinite program. Recall that an $n \times n$ matrix A is called *positive semidefinite*, denoted by $A \succeq 0$, if

$$x^T A x \geq 0$$

for all vectors x of length n. For an introduction to semidefinite programming, see, e.g., [VB96].

For a general matrix A which may be not positive semidefinite and even not symmetric, the nuclear norm heuristic (8.31) can be formulated in terms of semidefinite programming as being equivalent to

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}(\operatorname{trace}(W_1) + \operatorname{trace}(W_2)) \\ \text{subject to} & P_\Omega(X) = P_\Omega(A) \\ & \begin{bmatrix} W_1 & X \\ X^T & W_2 \end{bmatrix} \succeq 0 \end{array} \tag{8.34}$$

with additional optimization variables W_1 and W_2. To outline the analogy (strongly simplified; for details, see [RFP10]), we consider the singular value decomposition of X

$$X = USV^T$$

and of the block matrix

$$\begin{bmatrix} W_1 & X \\ X^T & W_2 \end{bmatrix} = \begin{bmatrix} U \\ V \end{bmatrix} S \begin{bmatrix} U^T & V^T \end{bmatrix}$$

leading to $W_1 = USU^T$ and $W_2 = VSV^T$. Since the left and right singular vector matrices are unitary, the traces of W_1 and W_2 are equal to the nuclear norm of X.

By defining the two factor matrices $L = US^{1/2}$ and $R = VS^{1/2}$, we easily observe that trace(W_1) + trace(W_2) = $\|L\|_F^2 + \|R\|_F^2$ and finally arrive at the optimization problem

$$\begin{aligned} &\text{minimize} \quad \frac{1}{2}\left(\|L\|_F^2 + \|R\|_F^2\right) \\ &\text{subject to} \quad P_\Omega\left(LR^T\right) = P_\Omega\left(A\right) \end{aligned}. \tag{8.35}$$

(Please consult [CR08] for a proof of equivalence of (8.34) an (8.35).)

Since in most practical applications data is noisy, we will allow some approximation error on the observed entries, replacing (8.35) by the less rigid formulation

$$\begin{aligned} &\text{minimize} \quad \frac{1}{2}\left(\|L\|_F^2 + \|R\|_F^2\right) \\ &\text{subject to} \quad \left\|P_\Omega\left(LR^T\right) - P_\Omega(A)\right\|_F^2 \le \sigma \end{aligned} \tag{8.36}$$

for a small positive σ. Thus, in Lagrangian form, we arrive at $-$ the formulation

$$\text{minimize} \ \lambda\frac{1}{2}\left(\|L\|_F^2 + \|R\|_F^2\right) + \left\|P_\Omega\left(LR^T\right) - P_\Omega(A)\right\|_F^2, \tag{8.37}$$

where λ is the regularization parameter that controls the noise in the data. Formulation (8.37) is also called *maximum-margin matrix factorization* (MMMF) and goes back to Nathan Srebro in 2005 [RS05]. Related work was also done by the group of Trevor Hastie [MHT10].

Problem (8.37) can be solved, e.g., by using a simple gradient descent method. Interestingly, the first SVD solution submitted to Netflix (by Simon Funk [Fun06], in 2006) used exactly this approach and achieved considerable progress. The final price was a mixture of hundreds of models with the SVD playing a crucial role.

After all, one may ask: what is the difference of the SVD (8.37) to the truncated SVD (8.14) that we have considered before? This brings us back to the introductory discussion in Sect. 8.1 about unknown values. The answer is that in (8.37), we assume only the entries A_{ij}, $(i,j) \in \Omega$ to be known. In contrast, in our previous formulation, we consider all entries A_{ij}, $(i,j) \notin \Omega$ to be zero. The Netflix competition, where all entries of A on a test set (of actual ratings of the users) had to be predicted, is obviously a classic matrix completion problem and hence (8.37) is the certainly the right approach.

In case of matrix factorization for an actual recommendation task, like that of a user or session matrix in Example 8.1, or even the probability matrix P, the discussion is more complex. In fact, we may view all non-visited entries of the matrix to be zero since the user didn't show any interest in them. This justifies our

Table 8.5 Comparison of prediction qualities and error norms for different regularization parameter values and with variable rank

	$\lambda = 0.1$				$\lambda = 0.01$				$\lambda = 0.001$			
k	p_1	p_3	e_F	e_Ω	p_1	p_3	e_F	e_Ω	p_1	p_3	e_F	e_Ω
2	1.18	2.11	13.22	3.62	0.20	1.22	34.52	3.29	0.02	0.32	55.98	3.28
5	2.66	4.83	11.78	3.33	0.96	3.66	29.77	2.93	2.05	3.66	41.52	2.89
50	5.74	9.16	8.13	1.94	5.77	8.21	20.47	0.63	5.41	7.84	27.02	0.55
100	6.13	9.75	7.32	1.80	6.29	9.84	17.62	0.28	6.15	9.12	21.82	0.12
200	6.09	9.86	7.11	1.79	6.32	10.02	16.64	0.28	6.32	9.93	18.06	0.12

previous approach. But, on the contrary, we may also argue that there is too little statistical volume and the user cannot view all products that he/she is potentially interested in, simply because there are too many of them. This would suggest the matrix completion approach. So there are pros and cons for both assumptions.

Example 8.8 We next repeat the test of Example 8.6 with the factorization according to formulation (8.37). Instead of a gradient descent algorithm, we used an ALS algorithm as described in [ZWSP08] which is more robust.

The results are contained in Table 8.5 whose structure basically corresponds to that of Table 8.3. Instead of the time, we have included the error norm e_Ω which corresponds to the Frobenius norm e_F but is calculated only on the given entries $(i,j) \in \Omega$. Thus, e_Ω is equal to the root-mean-square error (RMSE) multiplied by the square root of the number of entries $\sqrt{|\Omega|}$. Additionally, we compare different values of the regularization parameter λ.

From Table 8.5 we see that for increasing rank the RMSE is strongly declining, and we capture the given probabilities on Ω almost perfectly. A rank of 50–100 is already sufficient to bring the RMSE close to zero, and higher ranks also do not improve the prediction rate significantly. Unlike in Table 8.3, where we needed almost full rank to zero the approximation error, this is because here we just have to approximate the unknown entries.

In contrast to Table 8.3, the overall error e_F is slowly decreasing and remains very high. This is again because we do not approximate the zero values outside Ω. The prediction rate is comparable to Table 8.3. This indicates that for the probability matrix P, the approach to consider all non-visited entries to be zero is equally reasonable like assuming them to be unknown. ■

The result of Example 8.8 does not mean that the matrix completion approach is outright useless for the recommendation engine task. In fact, it could be, e.g., used to complete the matrix of transactions or transitions before it is further processed.

8.6 A Note on Efficient Computation of Large Elements of Low-Rank Matrices

All of the above-described factorization-based approaches to recommendation leave us with the computational burden of determining the largest entries of each column (or row) of a huge low-rank matrix. Interestingly enough, the authors did not found any publications about this fundamental problem!

A naïve approach would involve computing an explicit representation of the matrix entries, which incurs $O(mn)$ floating point operations as well as a storage complexity of mn, and then running a sorting algorithm for each column (or row), which takes at least another m operations for each row. This amounts to an overall quadratic complexity, which renders the naïve approach unsuitable for realtime recommendation applications, where m and n, commonly denoting numbers of products in a shop or numbers of customers or sessions, respectively, are notoriously huge.

Hence, if we are to deploy factorization-based prediction in a realtime RE, we must think of a more efficient method, or even settle for a heuristic algorithm. So how is this to be attained?

Apparently, to achieve a subquadratic complexity, we must somehow manage to avoid computing all of the matrix elements to begin with. In an analytical application, where the columns are discretized versions of continuous functions with a certain structure, the complexity issue might simply be remedied by computing only a few entries of each column and then using an interpolation method along with a continuous optimization procedure so as to obtain a fair estimate of the largest entries. Since in our setting, though, there is no continuous framework whatsoever, an interpolation-based remedy is not an option.

Let us consider a low-rank matrix $A := X^T Y$, where X and Y are matrices of dimensionality $r \times m$ and $r \times n$, respectively. Each of the entries a_{ij} of A is given by the inner product $x_i^T y_j$ of the corresponding columns of X and Y. Let us consider one single column

$$a := x^T y = \left(x_i^T y\right)_{i \in \{1,\ldots,m\}}$$

of A, where y denotes the corresponding column of Y and indices have been omitted for the sake of simplicity. Inspired by the interpolation approach, we would like to estimate the largest entries of a without computing all of the inner products. Since there are no structural clues like continuity at hand, we are left to our own devices to discover some exploitable structure. Yet for the whole thing to pay off, the cost of the discovery process must not exceed its benefits.

In the following, we shall describe a tentative remedy that is based on recursively *clustering* the columns of X. The so-called k-means clustering, sometimes referred to as *vector quantization* in signal processing and related communities, splits a set of m vectors into k disjoint clusters such that

$$\min_{C_1,\ldots,C_k,C_1,\ldots,C_k} \sum_{l=1}^{k} \sum_{i\in C_l} ||x_i - c_l||_2^2 \tag{8.38}$$

where $C_1, \ldots, C_k \subseteq \{1, \ldots, m\}$ denote the clusters and $c_1, \ldots, c_k \in \mathbb{R}^r$ the cluster centers to be minimized. It is straightforward to verify that if we are given clusters and keep them fixed in the optimization, corresponding optimal cluster centers are given by the centroids of the clusters. Conversely, given cluster centers c_l, $l \in \{1, \ldots, k\}$, corresponding optimal cluster may be obtained by assigning each x_i, $i \in \{1, \ldots, m\}$, to its nearest neighbor among the c_l. Hence, either the clusters or the centers may be eliminated from the objective, giving rise to an equivalent problem. A major drawback is that since (8.38) is not convex, we must settle for a local optimization. Yet, as described above, given a specific choice of either clusters or centers, the resulting partial optimization problem is straightforward to solve. This insight provides us with a fairly easy local optimization method based on alternating partial optimization.

Interestingly enough, (8.38) may be equivalently rewritten as the matrix factorization problem

$$\min_{C\in\mathbb{R}^{r\times k},B\in\mathbb{R}^{m\times k}} \left\|X - CB^T\right\|_F^2 \; s\cdot t\cdot b_{ij} \in \{0,1\}, \sum_{s=1}^{k} b_{is} = 1 \forall i \in \forall \{1,\ldots,m\}, j \in \{1,\ldots,k\}. \tag{8.39}$$

(By virtue of the Moore-Penrose inverse of B, it also becomes clear why partially optimal cluster centers are given by the centroids of the clusters.)

Not only does this enable to express the entire procedure in terms of linear algebra, but also, for the generic case that the desired number of clusters be smaller than the rank of X, does it provide us with a clue as to how to find a reasonable initial guess by virtue of relaxation: omitting the constraints yields

$$\min_{C\in\mathbb{R}^{r\times l},B\in\mathbb{R}^{m\times l}} \|X - CB^T\|_F^2, \tag{8.40}$$

which is nothing but an ordinary low-rank factorization problem that may be solved by means of SVD.

Now, given an optimizer $\widetilde{C}, \widetilde{B}$ of (8.40), how are we to construct an initial guess C_0, B_0 for (8.39)? A straightforward way that bears the advantage of circumventing normalization issues consists of taking B_0 to a feasible matrix that best approximates $\widetilde{B}$, which is obtained by taking each row of B_0 to be zero everywhere except for the element at which the corresponding row of $\widetilde{B}$ attains its largest value (the nongeneric case of a draw is handled by picking one of the possibilities at random), where we assign the value 1.

Herein, the circumstance that in many of the application previously described in this chapter, X has orthonormal rows comes in particularly handy since this spares

us from computing an SVD of X in order to solve (8.40): we may simply take B^T to be composed of the first k rows of X and C to be composed of the first k column unit vectors. In other words, the desired initial guess is available right away through "rounding" X itself to a feasible solution.

For more details about the connection of k-means and SVD and nonnegative matrix factorization, we refer to the work of Chris Ding [DHS05, DLJ10].

But how exactly may clustering figure in estimating the largest entries of a low-rank matrix? The procedure we would like to propose here is as follows: the column set of X is to be divided into k clusters, and each of the clusters is to be k-clustered itself, and so on in a recursive fashion until we arrive at a partitioning each set of which contains fewer than k elements. For simplicity's sake, let us take $k := 2$. Imagine the recursive cluster centers to be arranged in a tree to which an auxiliary root node connected to the two top-level cluster centers has been added. Starting at the root, we proceed by computing the inner product of y and the cluster centers corresponding to the children of the current node and move on to the child node where the inner product is largest. Applying this criterion recursively, we arrive at a leaf node that corresponds to a specific column x of X. Arguably, $x^T y$ should be fairly close to the greatest $x_i^T y$, $i \in \{1, \ldots, m\}$. Having deleted the node pertaining to x from our tree, we apply the same procedure to obtain an estimate of the second largest entry of a and repeat this procedure until we have gathered a sufficient amount of large elements.

Please note that, apart from the computational cost of clustering, which needs to be carried out only once, though, estimating the largest entry of a in the above-described fashion would require only $O(\log_k m)$ rather than $O(m)$ operations, which makes it an appealing candidate.

By means of linear algebra, it is even possible to assess the quality of our estimates by providing error bounds and provide a rigorous framework for the above procedure.

Lemma 8.1 *Let* $(V, \langle\cdot,\cdot\rangle)$ *be an inner product space. Then, for all* $c, v, w \in V$,

$$\langle c, w\rangle - \|w\|\|c - v\| \leq \langle v, w\rangle \leq \langle c, w\rangle + \|w\|\|c - v\|,$$

where $\|.\| := \sqrt{\langle\cdot,\cdot\rangle}$, i.e., *the norm induced by the inner product.*

Proof The statement is an immediate consequence of the Cauchy-Schwarz inequality: it holds that

$$|\langle v, w\rangle - \langle c, w\rangle| = |\langle v - c, w\rangle| \leq \|v - c\|\|w\|,$$

which, together with the simple fact that

$$-|\langle v, w\rangle - \langle c, w\rangle| \leq \langle v, w\rangle - \langle c, w\rangle \leq |\langle v, w\rangle - \langle c, w\rangle|,$$

yields the desired result. □

Proposition 8.7 *Let* $(V,\langle\cdot,\cdot\rangle)$ *be an inner product space and* $S \subseteq V$. *Then, for all* c, $w \in V$,

$$\langle c, w\rangle - \|w\| \min_{v\in S} \|v - c\| \leq \max_{v\in S} \langle v, w\rangle \leq \langle c, w\rangle + \|w\| \max_{v\in S} \|v - c\|$$

Proof Taking maximums on both sides of the rightmost inequality of the above Lemma yields

$$\begin{aligned} \max_{v\in S} \langle v, w\rangle &\leq \max_{v\in S} (\langle c, w\rangle + \|w\|\|v - c\|) \\ &\leq \max_{v\in S} \langle c, w\rangle + \max_{v\in S} (\|w\|\|v - c\|) \\ &= \langle c, w\rangle + \|w\| \max_{v\in S} \|v - c\|. \end{aligned}$$

Similarly, by taking maximums in the left inequality, we obtain

$$\begin{aligned} \max_{v\in S} \langle v, w\rangle &\leq \max_{v\in S} (\langle c, w\rangle + \|w\|\|v - c\|) \\ &\leq \max_{v\in S} \langle c, w\rangle + \min_{v\in S} (\|w\|\|v - c\|) \\ &= \langle c, w\rangle + \|w\| \min_{v\in S} \|v - c\|. \end{aligned}$$ □

This result equips us with a means to enclose the exact maximum inner product between a vector in the considered cluster and a given vector, while the only information that is required is the inner product between the given vector and the cluster center and the maximum distance from the cluster center. The latter, emerging as a side product of the foregoing clustering procedure, may safely be regarded as readily available.

By virtue of our bound, we may retrofit our branching heuristic into a full-blown branch-and-bound procedure which computes the exact result, though forfeiting a large degree of efficiency. To balance the tradeoff between fast computation and accuracy, we introduce a parameter $h \in [0;1]$ to relax the bound.

Let T denote the refinement tree obtained from recursive k-means clustering of the x_i, and let *children*(.) denote the functions that map a node of T to the set of its children. We define for $t \in T$

$$u_h(t) := \langle c_t, y\rangle + h\|y\| \max_{i\in C_t} \|x_i - c_t\|, l_h(t) := \langle c_t, y\rangle - h\|y\| \min_{i\in C_t} \|x_i - c_t\|,$$

where C_t, c_t denote the cluster and cluster center corresponding to t, respectively. Moreover, in the below-described algorithm, *select_node*$(\cdot)$ denotes some function which judiciously picks the next node to be refined from a list of candidate nodes (one may, e.g., choose the node t with the greatest upper bound $u_h(t)$ among all nodes in the list).

Algorithm 8.4 Branch-and-bound method for estimating $argmax_{i \in \underline{m}} \langle v_i, w \rangle$

Input: refinement tree T (obtained from recursive k-means clustering), relaxation parameter $h \in [0;1]$

Output: list (approximate) maximizers and the corresponding value

1: $L := \{root(T)\}$ ▷ initialize list of candidate nodes with the root of the refinement tree
2: **repeat**
3: $t_1 := select_node\ (L)$ ▷ select node to be branched
4: $t_1, \ldots, t_k := children(t)$
5: $L := L \backslash \{t\}$
6: **for** $i = 1, \ldots, k$ **do**
7: **if** $u_h(t_i) \geq \max_{s \in L} l_h(s)$ **then**
8: $L := L \cup \{t_i\}$
9: **if** $l_h(t_i) \geq \max_{s \in L} l_h(s)$ **then**
10: **for** $s \in L$ **do**
11: **if** $u_h(s) < l_h(t_i)$ **then**
12: $L := L \backslash \{s\}$ ▷ discard node if necessary
13: **end if**
14: **end for**
15: **end if**
16: **end if**
17: **end for**
18: **until** $\max_{s \in L} l_h(s) = \max_{s \in L} u_h(s) \wedge L$ contains a leaf node of T
19: **return** L, $\max_{s \in L} l_h(s)$

If h is taken to be zero, this procedure comes down to the heuristic method outlined in the beginning of this section, whereas $h = 1$ gives rise to an exact though by far less efficient procedure. To attain a satisfactory compromise between accuracy and efficiency, the parameter h must be adjusted from experience with respect to the nature of the inputs under consideration.

Since it would exceed the scope of this chapter, we leave a final assessment of the effectiveness of the above-devised procedure for future research. In first applications it proved to be very effective, where h was about 0.95.

8.7 Summary

We have seen that matrix factorization is a potentially valuable instrument of approximation with regard to devising recommendation engines. It offers different advantages. It may reduce the complexity of data representation (with respect to models of the environment or the cost function) and thus render the practical usability of the data feasible in the first place. Moreover, it allows for a

regularization with a resulting increase of quality by removing noise from the data. It is also possible to develop incremental factorization models such as the Brand's incremental SVD.

At the same time, the difficulties related to factorization for recommendation engines have clearly emerged. First, it has unequivocally turned out that the SVD-based factorization of the matrix of transition probabilities is effective with respect to neither of data compression nor increase of prediction rates. The only advantage consists in the opportunity of generating new recommendations by virtue of the low-rank approximation. A factorization of transactions over the sessions has turned out to be sensible, especially with regard to an increase of the prediction rate, but the yield was rather poor and, moreover, required longer sessions, which rarely occur in practice.

Modified formulations did not seem to help very much. Especially the nonnegative matrix factorization and the one based on Lanczos vectors did (in line with theory) lead to even worse prediction results than the SVD. Whereas the Lanczos vector calculation is at least cheaper than the SVD, for the NMF no comparable standard algorithms exist. In practice, here the ALS turns out to be most powerful. We also studied and validated the matrix completion approach, which considers all non-observed transitions to be unknown instead of zero. However, practical results turned out to be even worse.

All in all, we must conclude that the direct matrix factorization barely gives rise to a significant improvement of the prediction rate of the recommendation models – a circumstance which (as applied to prediction methods in general) has already become clear in the Netflix contest mentioned in Chap. 2.

Of course, this does not imply that devising better factorization models with regard to prediction rate is impossible in principle. One way is to incorporate additional RE-related assumptions into the factorization model. Another way is to include further dimensions into the factorization. This will be studied in the next chapter.

Chapter 9
Decomposition in Transition II: Adaptive Tensor Factorization

Abstract We consider generalizations of the previously described SVD-based factorization methods to a tensor framework and discuss applications to recommendation. In particular, we generalize the previously introduced incremental SVD algorithm to higher dimensions. Furthermore, we briefly address other tensor factorization frameworks like CANDECOMP/PARAFAC as well as hierarchical SVD and Tensor-Train-Decomposition.

9.1 Beyond Behaviorism: Tensor-PCA-Based CF

9.1.1 What Is a Tensor?

Historically, the concept of a tensor originated in differential geometry as a calculus for dealing with multilinear forms on manifolds. In recent years, thought, tensor-based approaches have made their way into numerical analysis of partial differential equations to model highly multivariate functions and, more importantly, into data mining as formal frameworks for multimodal data. We shall address the latter in more detail in the subsequent section after introducing the basic notions and notations related to the concept in the following.

If we conceive of a matrix as a two-dimensional array, then, in a nutshell, a tensor is a generalized matrix in that it may be thought of as a d-dimensional array, where d may be an arbitrary natural number. A more formal version of this definition suffices for the purpose pursued herein.

Definition 9.1 A (real) d-mode *tensor* of dimensionality $(n_1, \ldots, n_d) \in N^d$ is a sequence of real numbers indexed by the set $\underline{n_1} \times \ldots \times \underline{n_d}$. We denote the set of d-mode tensors of dimensionality $(n_1, \ldots, n_d) \in N^d$ by $\Re^{n_1, \ldots, n_d}$.

Multi-indexes are somewhat cumbersome to deal with. Thus, to achieve a clear representation, we introduce the following notation.

A. Paprotny and M. Thess, *Realtime Data Mining: Self-Learning Techniques for Recommendation Engines*, Applied and Numerical Harmonic Analysis,
DOI 10.1007/978-3-319-01321-3_9, © Springer International Publishing Switzerland 2013

Notation 9.1 *Multi-indexes are denoted by bold small letters,* i.e., $(i_1, \ldots, i_d) \in N^d$ *is denoted by* $\mathbf{i}$. *Furthermore, the multi-index set* $\underline{n_1} \times \ldots \times \underline{n_d}$, *induced by* $\mathbf{n} = (n_1, \ldots, n_d)$, *is denoted by* $\underline{\mathbf{n}}$. *Given multi-indexes* $\mathbf{m} = (m_1, \ldots, m_{d_1})$, $\mathbf{n} = (n_1, \ldots, n_{d_2})$, *their concatenation is denoted by*

$$(\mathbf{m}, \mathbf{n}) := (m_1, \ldots m_{d_1}, n_1, \ldots, n_{d_2}).$$

We state the following fundamental operations with tensors:

1. **Addition/subtraction**

$$\begin{aligned}&\Re^{\mathbf{m}} \times \Re^{\mathbf{m}} \to \Re^{\mathbf{m}},\\ &\left((a_i)_{\mathbf{i}\in\underline{\mathbf{m}}}, (b_j)_{\mathbf{j}\in\underline{\mathbf{m}}}\right) \mapsto (a_{\mathrm{i}}) \pm (b_{\mathrm{j}}) := (a_{\mathrm{i}} \pm b_{\mathrm{j}})_{\mathbf{i}\in\underline{\mathbf{m}}}.\end{aligned}$$

2. **Contraction**
   Let $\mathbf{p} = (p, q)$, $\mathbf{n} = (1, \ldots, m_{p-1}, m_{p+1}, \ldots, m_{q-1}, m_{q+1}, \ldots, m_d)$

$$\Re^{\mathbf{m}} \to \Re^{\mathrm{n}}$$

$$\left(\left(a_{(\mathbf{i},\mathbf{k})}\right)_{(\mathbf{i},\mathbf{k})\in\underline{(\mathbf{n},\mathbf{p})}}\right) \mapsto (a_{\mathrm{i}}) := \left(\sum_{\mathbf{k}\in\underline{\mathbf{p}}} a_{(\mathbf{i},\mathbf{k})}\right)_{\mathbf{i}\in\underline{\mathbf{n}}}.$$

3. **Inner product**

$$\Re^{\mathbf{m}} \times \Re^{\mathbf{m}} \to \Re,$$

$$\left((a_{\mathbf{i}})_{\mathbf{i}\in\underline{\mathbf{m}}}, (b_{\mathbf{j}})_{\mathbf{j}\in\underline{\mathbf{m}}}\right) \mapsto \left\langle (a_{\mathbf{i}}), (b_{\mathbf{j}})\right\rangle := \sum_{\mathbf{i}\in\underline{\mathbf{m}}} a_{\mathrm{i}} b_{\mathrm{i}}.$$

4. **Outer product** (tensor product)

$$\begin{aligned}&\Re^{\mathbf{m}} \times \Re^{\mathbf{n}} \to \Re^{(\mathbf{m},\mathbf{n})},\\ &\left((a_{\mathbf{i}})_{\mathbf{i}\in\underline{\mathbf{m}}}, (b_{\mathbf{j}})_{\mathbf{j}\in\underline{\mathbf{n}}}\right) \mapsto (a_{\mathbf{i}}) \otimes (b_{\mathbf{j}}) := (a_{\mathbf{i}} b_{\mathbf{j}})_{(\mathbf{i},\mathbf{j})\in\underline{(\mathbf{m},\mathbf{n})}}.\end{aligned}$$

5. **Contracted product** (multilinear product)
   Let $\mathbf{m} \in \mathbf{N}^{\mathrm{d}_1}, \mathbf{n} \in \mathbf{N}^{\mathrm{d}_2}, \mathbf{p} \in \mathbf{N}^{\delta}$.

$$\otimes_\delta : \Re^{(\mathbf{m},\mathbf{p})} \times \Re^{(\mathbf{n},\mathbf{p})} \to \Re^{(\mathbf{m},\mathbf{n})},$$

$$\left(\left(a_{\mathbf{i},\mathbf{k}}\right)_{(\mathbf{i},\mathbf{k})\in\underline{(\mathbf{m},\mathbf{p})}}, \left(b_{(\mathbf{j},\mathbf{k})}\right)_{(\mathbf{j},\mathbf{k})\in\underline{(\mathbf{n},\mathbf{p})}}\right) \mapsto \left(a_{(\mathbf{i},\mathbf{k})}\right) \otimes_\delta \left(b_{(\mathbf{j},\mathbf{k})}\right) := \left(\sum_{\mathbf{k}\in\underline{\mathbf{p}}} a_{(\mathbf{i},\mathbf{k})} b_{(\mathbf{j},\mathbf{k})}\right)_{(\mathbf{i},\mathbf{j})\in\underline{(\mathbf{m},\mathbf{n})}}.$$

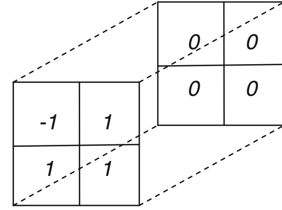


Fig. 9.1 Illustration of a tensor $A \in \Re^{(2,2,2)}$

6. **Multilinear p-*mode* product with a matrix**
   Special case of 5 for $\delta = 1$ and matrix b

$$\times_p : \Re^{\left(m_1,\ldots,m_{p-1},m_p,m_{p+1},\ldots,m_d\right)} \times \Re^{\left(n,m_p\right)} \to \Re^{\left(m_1,\ldots,m_{p-1},m_p,m_{p-1},\ldots,m_d\right)},$$

$$\left((a_{\mathbf{i}})_{\mathbf{i}\in\underline{\mathbf{m}}}, \left(b_{jk}\right)_{(j,k)\in\underline{\left(n,m_p\right)}}\right) \mapsto (a_{\mathbf{i}})\times_p\left(b_{jk}\right) := \left(\sum_{i_p}^{m_p} a_{i_1\ldots i_p\ldots i_d} b_{ji_p}\right)_{(\mathbf{i},j)\in\underline{(\mathbf{m},n)}}.$$

Furthermore, we endow the vector space $\Re^{\mathbf{n}}$ (w.r.t. the previously defined sum and scalar multiplication) with the *Euclidean inner product*

$$\langle A, B\rangle := \sum_{\mathbf{i}\in\underline{\mathbf{n}}} a_{\mathbf{i}} b_{\mathbf{i}}$$

and the thus induced (generalized) *Frobenius norm*

$$\|A\|_F^2 := \langle A, A\rangle.$$

Definition 9.2 Let $A \in \Re^{\mathbf{n}}, k \in \underline{d}$. We define $\mathbf{n}^{(k)} := (n_1, \ldots, n_{k-1}, n_{k+1}, \ldots, n_d)$. Furthermore, let $\upsilon : \underline{\mathbf{n}^{(k)}} \to \left|\underline{\mathbf{n}^{(k)}}\right|$ be 1–1, *i.e.*, an enumeration. Then the matrix $A^{(k)}$ with entries

$$a^{(k)}_{i_k \upsilon\left(i^{(k)}\right)} := a_{\mathbf{i}}, \mathbf{i} \in \underline{\mathbf{n}}$$

is referred to as k-mode matricization of A.

As regards the scope of this chapter, the choice of enumeration is inconsequential if deployed consistently. Hence, in what follows, we shall consider the k-mode matricization of a given tensor as a uniquely defined object.

Example 9.1 Let $A \in \Re^{(2,2,2)}$ with entries

$$\begin{aligned} &a_{(1,1,1)} = -1, \quad a_{(1,2,1)} = 1, \quad a_{(1,1,2)} = 0, \quad a_{(1,2,2)} = 0,\\ &a_{(2,1,1)} = 1, \quad\;\; a_{(2,2,1)} = 1, \quad a_{(2,1,2)} = 0, \quad a_{(2,2,2)} = 0. \end{aligned}$$

A graphical rendition of this tensor is provided by Fig. 9.1.

Then its matricizations are given by

$$A^{(1)} = \begin{bmatrix} -1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix},$$

$$A^{(2)} = \begin{bmatrix} -1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix},$$

and

$$A^{(3)} = \begin{bmatrix} -1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

■

9.1.2 And Why We Should Care

In classical data mining, matrices are deployed to model weighted binary relations, such as the session-product relation in the framework presented in Sect. 8.1. Tensor algebra provides a means to model higher-order relations. The latter, e.g., arise in context-aware models: with regard to user-rating prediction, it is natural to expect that certain extrinsic circumstances of the situation in which a user is prompted for a rating affect the latter. As an illustration, consider the following datasets described by [KABO10]:

> [The first dataset] contains 1464 ratings by 84 users for 192 movies. . . . [The users] were asked to fill out a questionnaire on movies using a rating scale ranging from 1 . . . to 13 . . . . They were also queried on additional information about the context of the movie-watching experience[:] . . . companion, day of the week, if it was on the opening weekend, season, and year seen. . . .
> [The second dataset] contains food rating data from 212 users on 20 food menus. . . . The users were asked to rate the food menu while being in different levels of hunger. Moreover, some ratings were done when really experiencing the situation (i.e., participants were hungry and ordered the menu) and some while imagining the situation. For example, in the virtual situation participants could be full, but should have provided a rating for a food menu imagining that they are hungry.

The above-outlined datasets are established upon relations of genuinely high order. Beyond that, it is also possible to construct higher-order relations by combining lower-order ones. For example, users may endow products in a shop with tags and provide (implicit or explicit) ratings for both products and tags. This gives rise to two weighted relations, which may be combined as follows: if a particular user u endows the product p with a rating score S_p which is tagged with t, which, in turn, has been given a score of s_t by the user u, then the triplet (u,s,t) is assigned with the value $s_p \cdot s_t$. In a similar fashion, one might take background information on users and products into account, which leads to relations of arbitrary order.

The above examples give rise to the question of whether the previously developed framework of PCA-based CF may be extended to the tensor case in a

meaningful way. This entails a considerable amount of mathematical as well as engineering-related issues that will be tackled in the following.

Furthermore, we shall be interested in applying tensorial factorization models as a means of regularized estimation of transition probability functions of Markov decision processes so as to tackle large state space arising from state augmentation.

9.1.3 PCA for Tensorial Data: Tucker Tensor and Higher-Order SVD

Definition 9.3 A d-mode $\mathbf{n}$-dimensional Tucker tensor of (Tucker) rank $\mathbf{t} \leq \mathbf{n}$ is a tensor

$$A = C \times_1 U_1 \times_2 \ldots \times_d U_d, \tag{9.1}$$

where $C \in \Re^{\mathbf{t}}$ denotes the core tensor and $U_k \in \Re^{(n_k, t_k)}, k \in \underline{d}$ the mode factors.

Remark 9.1 A Tucker tensor is completely determined by its core tensor and the mode factors. Therefore, with a slight abuse of language, we shall henceforth identify a Tucker tensor (9.1) with the tuple $(U_1, \ldots, U_d, C)$.

Thus, for each component $a_{i_1,\ldots,i_d}$ of A decomposition, (9.1) reads as

$$a_{i_1,\ldots,i_d} = \sum_{j_1=1}^{t_1} \ldots \sum_{j_d=1}^{t_d} c_{j_1,\ldots,j_d} u^1_{i_1,j_1} \ldots u^d_{i_{1d},j_d},$$

where $c_{j_1,\ldots,j_d}$ are the components of the core tensor C and $u^k_{i,j}$ are the elements of the factor U_k.

In utmost generality, tensor factorization problems in terms of the Tucker format may be stated as instances of the framework

$$\min_{U_1 \in C_1,\ldots,U_d \in C_d, C \in C_0} f(A, C \times_1 U_1 \times_2 \ldots \times_d U_d), \tag{9.2}$$

where the cost function f stipulates a notion of approximation quality and constraints, e.g., nonnegativity or orthonormality, are encoded in the sets $C_0, \ldots, C_d$. Please note that the matrix factorization framework (8.1) coincides with the special case of (9.2) where $d = 2$.

Definition 9.4 Let $A \in \Re^{\mathbf{n}}$ und $\mathbf{t} \leq \mathbf{n}$. Then a Tucker tensor

$$A_t = C \times_1 U_1 \times_2 \ldots \times_d U_d,$$

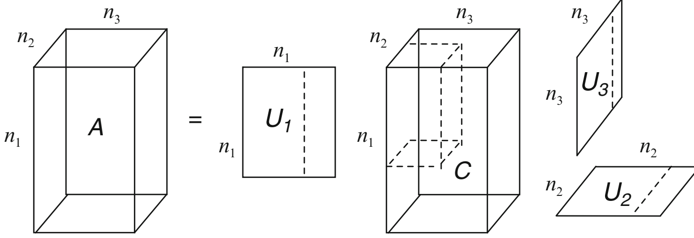


Fig. 9.2 Illustration of the HOSVD for a 3-mode tensor A. The *dotted lines* indicate the boundaries of the submatrices or tensors, respectively, corresponding to a truncated HOSVD

where

$$C := A \times_1 U_1^T \times_2 \ldots \times_d U_d^T$$

and $U_i, i \in \underline{d}$, is a matrix of left singular vectors corresponding to the t_i largest singular values in an SVD of $A^{(i)}$ and is referred to as truncated higher-order singular value decomposition (HOSVD) of rank-$\mathbf{t}$. The i_k th k-mode singular value of A is defined as

$$s_{ik}^{(k)} := \left\| (c_{\mathbf{i}})_{\mathbf{i}^{(k)} \in \mathbf{n}^{(k)}} \right\|_F = \sqrt{\sum_{\mathbf{i}^{(k)} \in \mathbf{n}^{(k)}} c_{\mathbf{i}}^2}.$$

As in the matrix case, we shall refer to the factorization corresponding to the rank-$\mathbf{n}$ truncated HOSVD of an $\mathbf{n}$-dimensional tensor simply as an HOSVD thereof. Similarly to Fig. 8.2, a graphical representation of the (truncated) HOSVD is given in Fig. 9.2.

Definition 9.4 immediately leads to Algorithm 9.1 of a truncated HOSVD.

Algorithm 9.1 Truncated HOSVD

Input: tensor $A \in \mathfrak{R}^{\mathbf{n}}$, truncation rank $\mathbf{t} \leq \mathbf{n}$

Output: factor matrices $U_k \in \mathfrak{R}^{(n_k, t_k)}, k = 1, \ldots, d$, core tensor $C \in \mathfrak{R}^{\mathbf{t}}$

1: **for** $k = 1, \ldots, d$
2: calculate matrix $U_k \in \mathfrak{R}^{(n_k, t_k)}$ of principal left singular vectors of matricization $A^{(k)}$
3: **end for**
4: $C := A \times {}_1U_1^T \times {}_2 \ldots \times {}_dU_d^T$

Example 9.2 Consider the tensor A from the previous Example 9.1. As one easily verifies, it holds that

$$A^{(1)} \left(A^{(1)} \right)^T = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix},$$

$$A^{(2)}\left(A^{(2)}\right)^T = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix},$$

and

$$A^{(3)}\left(A^{(3)}\right)^T = \begin{bmatrix} 4 & 0 \\ 0 & 0 \end{bmatrix}.$$

Hence, $U_1 = U_2 = U_3 = I$. Since there are multiple eigenvalues in the first two modes, there is no unique (1,1,2) truncated HOSVD. Instead, any of the subspaces spanned by e_1 or e_2 are 1-dimensional principal subspaces of both of the matricizations $A^{(1)}$ and $A^{(2)}$. Assigning $\widetilde{U}_1 := \widetilde{U}_2 := e_1$ yields a core tensor C^{11} with entries

$$c_{(1,1,1)} = -1, c_{(1,1,2)} = 0.$$

The remaining three choices give rise to always the same core tensor $\hat{C}$ with entries

$$\hat{c}_{(1,1,1)} = 1, \hat{c}_{(1,1,2)} = 0.$$

The respectively induced rank-(1,1,2) approximations to A are given by

$$\widetilde{A}^{(1)}_{11} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

$$\widetilde{A}^{(1)}_{12} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

$$\widetilde{A}^{(1)}_{21} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix},$$

and

$$\widetilde{A}^{(1)}_{22} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}.$$

In each case, the approximation error is the same. ■

The following properties of the HOSVD have been worked out.

Theorem 9.1 (Theorem 2 in [DLDMV00]) *The core tensor of a* (*rank*-$\mathbf{n}$) *HOSVD of* $A \in \Re^{\mathbf{n}}$ *satisfies:*

1. *Subtensors of order* $d-1$ *are mutually orthogonal,* i.e.,

$$\left\langle (c_{\mathbf{i}})_{\mathbf{i}^{(k)} \in \underline{\mathbf{n}^{(k)}}}, (c_{\mathbf{j}})_{\mathbf{j}^{(k)} \in \underline{\mathbf{n}^{(k)}}} \right\rangle \propto \delta_{i_k j_k} \forall k \in \underline{d}, i_k, j_k \in \underline{n_k}.$$

3. *The k-mode singular values are ordered,* i.e.,

$$s_{i_k+1}^{(k)} \geq s_{i_k}^{(k)} \forall k \in \underline{d}, i_k < n_k.$$

The subsequent bound is an improvement over that in Property 10 in [DLDMV00] by a factor of $\frac{1}{d}$ through a slight modification of the proof therein.

Proposition 9.1 *A rank-*$\mathbf{t}$ *truncated HOSVD* $A_{\mathbf{t}}$ *of* $A \in \Re^{\mathbf{n}}$ *satisfies*

$$\|A - A_{\mathbf{t}}\|_F^2 \leq \frac{1}{d} \sum_{k \in \underline{d}} \sum_{t_k < i_k \leq n_k} \left(s_{i_k}^{(k)}\right)^2.$$

Proof

$$\begin{aligned}
\|A - A_{\mathbf{t}}\|_F^2 &= \sum_{\mathbf{t} < \mathbf{i} \leq \underline{\mathbf{n}}} c_{\mathbf{i}}^2 \\
&= \frac{1}{d} \sum_{k \in \underline{d}} \sum_{\mathbf{t} < \mathbf{i} \leq \underline{\mathbf{n}}} c_{\mathbf{i}}^2 \\
&\leq \frac{1}{d} \sum_{k \in \underline{d}} \sum_{t_k < i_k \leq n_k} \sum_{\mathbf{i}^{(k)} \in \mathbf{n}^{(k)}} c_{\mathbf{i}}^2 \\
&= \frac{1}{d} \sum_{k \in \underline{d}} \sum_{t_k < i_k \leq n_k} \left(s_{i_k}^{(k)}\right)^2.
\end{aligned}$$

□

As opposed to the matrix case, the rank-$\mathbf{t}$ truncated HOSVD does *not* provide an optimal rank-$\mathbf{t}$ approximation. Nevertheless, it appears to be a promising candidate for a tensor generalization of PCA-based CF because:

1. Certain properties of the matrix case are preserved.
2. The quality of approximation may be estimated.
3. The computation of the HOSVD reduces to the computation of matrix SVDs.
4. In particular, this permits incremental computation.
5. The projection approach to the prediction of unknown values may be carried over in a straightforward fashion.

A major drawback, however, is the exponential dependency of computational complexity on the number of modes. Therefore, it is suitable for applications with a moderate number of modes only. To conclude this section, we point out that a mode-scalable generalization of the SVD will be introduced in Sect. 9.3.

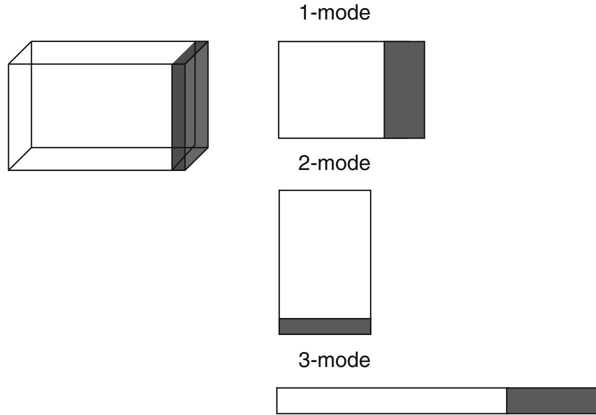


Fig. 9.3 Matricization of a 3-mode tensor after adding a matrix "to the right," i.e., in mode 2 (d = 2)

9.1.4 ... And How to Compute It Adaptively

The tensor generalization of the SVD updating problem discussed in Sect. 8.2 is as follows: how can an HOSVD of a tensor $\widetilde{A} \in \Re^{\mathbf{n}}$ with entries

$$\widetilde{a}_{\mathbf{i}} = \begin{cases} a_{\mathbf{i}}, & \mathbf{i} \leq \mathbf{n} - \mathbf{e}_d \\ b_{\mathbf{i}^{(d)}}, & \mathbf{i}_d = n_d \end{cases}$$

where $A = (a_{\mathbf{i}}) \in \Re^{\mathbf{n}-\mathbf{e}_d}, B \in \Re^{\mathbf{n}^{(d)}}$, and $(\mathbf{e}_k)_z := \delta_{zk}$ be expressed in terms of an HOSVD of A? We refer to the d–1-mode subtensor B of $\widetilde{A}$ as a slice.

Establishing a new tensor by adding a slice to a given one generalizes the previously discussed situation in which a new matrix is established by adding a column.

To extend the adaptive framework presented in Sect. 8.3.2, we need to observe the resulting changes in the mode matricizations, which are graphically illustrated by Fig. 9.3. Let us first consider the case where $k \neq d$ and assume that the multi-index enumeration chosen for the matricization $\widetilde{A}^{(k)}$ satisfies

$$i_d \leq j_d \Rightarrow v(\mathbf{i}) \leq v(\mathbf{j}) \ \forall \mathbf{i}, \mathbf{j} \in \underline{\mathbf{n}}.$$

(This may, e.g., be achieved by using a *lexicographic ordering*.) Then the kth mode matricization is of the form

$$\widetilde{A}^{(k)} = \left[A^{(k)} B^{(k)} \right],$$

where $A^{(k)}$, $B^{(k)}$ are kth mode matricizations of the tensors A, B with respect to a suitably chosen enumeration. Hence, if α denotes the number of columns of $B^{(k)}$, we may obtain an SVD of $\widetilde{A}^{(k)}$, given an SVD of $A^{(k)}$, which is assumed to be available from previous computations, by applying the procedure presented in Sect. 8.3.2 α times.

Let us now turn to the "troublemaker mode," i.e., the frontal mode, $k = d$. In this case, a matricization is of the form

$$\widetilde{A}^{(d)} = \begin{bmatrix} A^{(d)} \\ B^{(d)} \end{bmatrix}.$$

Here, applying the procedure to $\left(\widetilde{A}^{(d)}\right)^T$ and assigning the appropriate matrix of right singular values to U_d may do the trick. Please note, however, that avoiding any explicit representation of a matrix of right singular vectors of a matrix to which columns are appended is critical as regards realtime scalability of the procedure. Therefore, we shall develop a method to compute the projection of the added slide onto the range of the considered matrix of singular vectors in the following. Let

$$\left(\widetilde{A}^{(d)}\right)^T =: \underline{\widetilde{A}} = \widetilde{U} \underbrace{\begin{bmatrix} S & z \\ 0^T & c \end{bmatrix}}_{=:\widetilde{S}} \widetilde{V}^T$$

be a decomposition according to (8.15).

Moreover, let $\overline{U}\,\overline{S}\,\overline{V}^T$ be a truncated SVD of $\widetilde{S}$, $\hat{U} := \widetilde{U}\overline{U}$, and $\hat{V} := \widetilde{V}\overline{V}$. Then $\hat{U}\overline{S}\hat{V}^T$ is a truncated SVD of $\widetilde{A}$. We shall seek after $\underline{\widetilde{A}}\hat{V}\hat{V}^T e_n$, i.e., the projection of the last column of $\underline{\widetilde{A}}$ onto the principal subspace spanned by $\hat{V}$, where e_n denotes the vector of all zeros except for the last entry, which is 1. Since

$$\overline{S}\,\overline{V}^T e_n = \overline{U}^T \widetilde{S} e_n = \overline{U}^T \begin{bmatrix} z \\ c \end{bmatrix}$$

and $\widetilde{V}^T e_n = e_n$ by equation (8.15), we obtain

$$\underline{\widetilde{A}}\hat{V}\hat{V}^T e_n = \widetilde{U}\overline{U}\,\overline{U}^T \begin{bmatrix} z \\ c \end{bmatrix}, \tag{9.3}$$

which avoids the "column scaling."

The entire update procedure is summarized in Algorithm 9.2.

Algorithm 9.2 HOSVD update

Input: matrices $U_k \in \Re^{(n_k, t_k)}$ of principal left singular vectors of $A^{(k)}$, $k = 1, \ldots, d - 1$, matrix U of principal left singular vectors of $(A^{(d)})^T$, new slice $B \in R^{\mathbf{n}^{(d)}}$

(continued)

Algorithm 9.2 HOSVD update (continued)

Output: matrices $U_k \in \Re^{(n_k, t_k)}$ of principal left singular vectors of $[A^{(k)} B^{(k)}]$, $k = 1, \ldots, d-1$, matrix U of principal right singular vectors of $\left(\widetilde{A}^{(d)}\right)^T =: \underline{\widetilde{A}} = \left[\left(A^{(d)}\right)^T \left(B^{(d)}\right)^T\right]$, updated slice B according to updated HOSVD

1: Update U_k, $k = 1, \ldots, d-1$ according to the above-described incremental SVD (Algorithm 8.1)
2: $A := (A^{(d)})^T, \;\; b := (B^{(d)})^T$
3: $\underline{\widetilde{A}} := [A \quad b]$
4: $z := U^T b$, $c := \|b - Uz\|_2$
5: Compute $\widetilde{U}$ and $\overline{U}$
6: $\hat{U} := \widetilde{U}\overline{U}$
7: $b := \hat{U}\overline{U}\begin{bmatrix} z \\ c \end{bmatrix}$
8: $B^{(d)} := b^T$
9: **for** $k = 1, \ldots, d-1$
10: $\quad B^{(k)} := U_k U_k^T B^{(k)}$
11: **end for**

We shall briefly explain Algorithm 9.2. It crucially relies on the decomposition (9.4). First, we update the frontal mode $B \times U_d U_d^T$, i.e., by virtue of $\left(\widetilde{A}^{(d)}\right)^T \hat{V} \hat{V}^T e_n$. The latter is then projected along the other modes by means of (9.5). This relies on the insight that, for the left-projection, we only need the new slice B to compute its approximation. Unfortunately, this does not hold for the right-projection, which is needed in the frontal mode, and we must approximate the entire tensor (even if only adaptively), to obtain the updated slice. This renders the application to computing recommendations more complicated; we shall discuss this in more detail in the next section.

Another drawback is the fact that due to the large number of rows, steps 5 and 6 are computationally expensive and, thus, the procedure scales poorly.

9.1.5 Computing Recommendations

To compute recommendations, we might initially proceed along the lines of the outset of Sect. 8.3.3 by adding the updated slice of the current session B to the previous tensor A in each step of the session and carrying out the incremental learning step according to Algorithm 9.2. The latter provides the approximated slice B_t, which we deploy to forecast the current session. When the session terminates, we compute the HOSVD of $\widetilde{A}$, i.e., we carry out a complete learning step.

Similarly, we may divide into a training and a test set along the sessions, approximate the training set by means of offline learning (recommendable by means of the online HOSVD from Algorithm 9.2), and use the approximation to forecast the remaining session entries in the test set.

Of course, we would prefer the efficient way of left-projection approximations, similarly to Sect. 8.3.3. First, we have, correspondingly to (8.18) according to Definition 9.4,

$$A_{\mathrm{t}} = A \times_1 U_1 U_1^T \times_2 \ldots \times_{d-1} U_{d-1} U_{d-1}^T \times U_d U_d^T, \tag{9.4}$$

i.e., the approximated matrix is the multilinear product of the projections of A on the spaces of the singular vector bases of its n-matricizations. Thus, the question is: can we generalize the left-projection approximation (8.20) to the d-dimensional case for the slice B, i.e.,

$$B_{\mathrm{t}} = B \times_1 U_1 U_1^T \times_2 \ldots \times_{d-1} U_{d-1} U_{d-1}^T ? \tag{9.5}$$

Besides the formal analogy, this conjecture is supported by the fact that, in the HOSVD Algorithm 9.2, too, the projection (9.5) is carried out at the end. Of course, the answer is negative (since, otherwise, the frontal mode in Algorithm 9.2 would be redundant).

Sadly enough, the left-projection approximation is not exact for $n > 2$, i.e., not consistent with (9.4), since it holds in some cases that

$$A_{\mathrm{t}} \neq A \times_1 U_1 U_1^T \times_2 \ldots \times_{d-1} U_{d-1} U_{d-1}^T.$$

One may easily convince oneself of this by a straightforward evaluation of an example. Only for the special case that the SVD corresponding to the frontal mode $A^{(d)}$ is of full rank does (9.5) hold unrestrictedly. This does not mean that (9.5) is outright useless for the high-dimensional case; in practice, it often yields sufficiently good results, as we shall see later. But caution is advised.

Example 9.3 In the two-dimensional case, the "slice" B corresponds to a vector, e.g., the products (dimension 1) within a session (dimension 2), and we obtain according to (9.5)

$$B_{\mathrm{t}} = U_1 U_1^T B,$$

which complies with the SVD case (8.20).

In the following, we shall present experimental results for a real-world data set and compare those to the predictions obtained from a three-dimensional tensor factorization. To this end, we consider the transaction data of a mail-order company. The considered data set encompasses 3,016 products and consists of 25,000 transactions from some 800 sessions. We split the data set into a training set of 20,000 transactions, from which we learn the initial factorization model, and the actual test set of 5,000 transactions.

Table 9.1 Comparison of prediction rates (absolute): adaptive SVD with variable rank

k	p_1
10	26
50	105
100	132
200	142
300	145
400	140
500	**152**
600	148
700	148
800	145

The products correspond to the first, and the sessions to the second dimension. We deploy the adaptive SVD Algorithm 8.1 and compare the (absolute) prediction rates (i.e., numbers of correctly predicted products) with respect to the test set. To do so, we establish the vector a over all products, whereupon we assign the value 1 to the hitherto visited products and 0 to the remaining ones. By virtue of the projection procedure (8.21), we compute the vector a_k and recommend the product with the largest value therein. Upon termination of each session, we carry out an incremental SVD step, i.e., compute the new rank-k SVD.

The result is displayed in Table 9.1.

Rank 500 yields the highest prediction rate, namely, $\frac{152}{5000} \times 100\,\% \approx 3\,\%$. ■

Example 9.4 Let us now turn to the three-dimensional case. The selected product may, for instance, act as the first dimension, the time of transaction as dimension 2, and, again, the session as dimension 3. A new slice thus corresponds to the matrix of the hitherto occurred rewards for the selected products and the times of their being invoked. Then we may write (9.5) as follows (provided that one insists on using it despite its lack of correctness):

$$B_{\mathrm{t}} = U_1 U_1^T B U_2 U_2^T.$$

Thus, or by virtue of Algorithm 9.2, respectively, we obtain a matrix of scores over products and times. For the moment of the next prediction, we insert its time and recommend in the corresponding column again the products with the entries of the highest scores.

In the following, we consider the experimental results for the data set from the previous Example 9.3. For technical reasons, though, we use product variations rather than times as a new dimension.

We shall explain this in more detail. The retailer organizes the product by a master-variation scheme. Thereupon, the master describes the product and the variation its varieties, which, in this case, are given by colors. A possible master might, e.g., be “Nike T-Shirt Air Jordan T-56” and the available colors “white,” “blue,” and “red.” Hence, the master does not exist physically, but only the pairs (master, variation).

Table 9.2 Comparison of prediction rates (absolute): adaptive HOSVD with variable ranks

	Rank w.r.t. the variation mode			
Rank w.r.t. the master mode	10	100	200	300
10	48	56	56	55
100	104	120	121	126
200	122	139	136	138
300	136	**156**	151	149
400	141	146	142	145

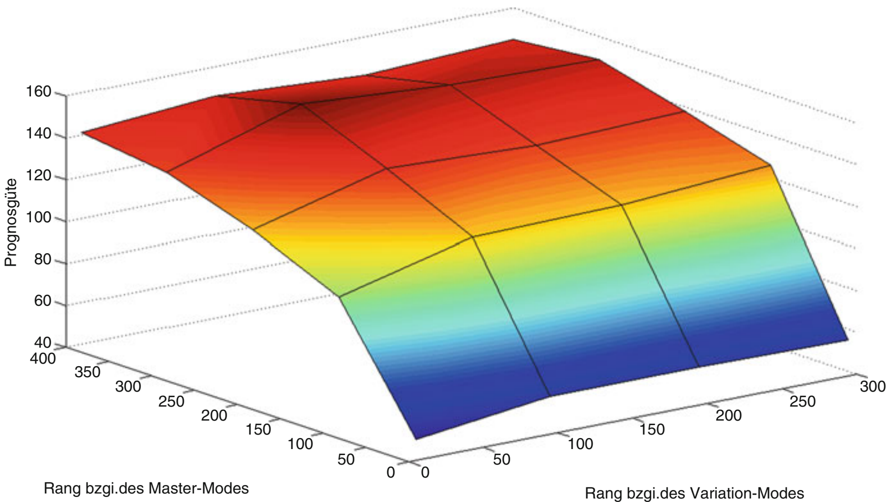


Fig. 9.4 Illustration of the prediction rates (absolute): adaptive HOSVD with variable ranks

The considered data set thus comprises 3,016 masters (i.e., products) and 596 variations (i.e., colors). So, we would like the variation to act as another dimension. All in all, we incorporate the master (dimension 1), the variation (dimension 2), and the session (dimension 3). The factorization according to Algorithm 9.2 is carried out on the training data, and we use the projection procedure (9.5) for the evaluation. Thus, the employed method is consistent with that in the previous example with an initial training set of 20,000 observations and a test set of 5,000 observations for the actual evaluation.

Hence, on each product view, the slice B is formed as a matrix over all masters and variations, whereupon we assign the value 1 to the hitherto considered products, i.e., their (master, variation) pairs, and 0 to the rest. We compute the updated slice B_t by means of the projection procedure (9.5) and recommend the (master, variation) pair with the highest value therein. After each session, we carry out an incremental learning step with respect to all modes except the frontal one. This corresponds to step 1 of Algorithm 9.2.

The result is displayed in Table 9.2 and in Fig. 9.4.

Table 9.3 Comparison of prediction rates (absolute): various SVDs with variable rank

Approach	k (or $\mathbf{t}$)	p_1
Mode 2	10	26
Mode 3/projection	10/10	48
Mode 3/HOSVD	10/10/10	**96**

Obviously, the method works out. Moreover, the result is somewhat better than that of the two-dimensional case in Table 9.1. The improvement is hitherto modest, though.

We shall now consider the complete HOSVD Algorithm 9.2 for evaluation. This is consistent with the just-described approach of a projection procedure, however, including the frontal mode. For the latter, we employ the somewhat awkward online procedure (Algorithm 9.2).

To do so, we must carry out steps 2–7 of Algorithm 9.2 for the frontal mode d after each product view leading to an update of the slice B, which, in particular, includes the incremental update step 5, and delete the updated matrix U afterward. Then we apply the projection procedure (9.5) to the thus updated slice $B^{(d)}$, which is consistent with steps 8–10. We save U not until termination of the session. Hence, the actual "learning" takes place not until the end of the session.

The prediction rate of the complete HOSVD, along with that of the foregoing procedures, is summarized in Table 9.3. For the HOSVD, we need 3 ranks, namely, one for each of the dimensions of the masters, the variations, and the sessions. Sadly enough, due to the high computational complexity of Algorithm 9.2, the comparison is feasible for low ranks only. We use 10 for each dimension.

The result suggests that the complete HOSVD works well in principle and furthermore yields better results than the projection method. This, however, is a statement under reservation: technically, we would have to carry out the entire comparison of prediction rates over varying ranks. ■

Example 9.5 We now consider the transition probabilities as a function of the sessions. The first dimension is thus the considered product (s), the second one is the destination of the transition (s'), and the third one is the session (u) itself. Hence, the first two dimensions span the transition probabilities *for each session*. A new slice therefore represents the matrix $P_u = \left(p_{u,ss'}\right)_{s,s' \in S}$ of the transition probabilities that have hitherto occurred in the session. By applying the factorization, we obtain the matrix $\widetilde{P}_u$ of all estimated transition probabilities for the current session. ■

Example 9.6 Eventually, we may also consider the transition probabilities as a function of the recommendation a. This corresponds to the approach from Example 9.5 with the recommendation a in lieu of the session u. Here, however, all dimensions have the same cardinality and the third dimension does not grow dynamically. Therefore, the adaptive approach makes no sense with respect to content (though, possibly as a technology for offline learning). We thus factorize

the transition probabilities $P = \left(p_{ss'}^{a}\right)_{s,s',a \in S}$ and obtain the approximate transition probabilities, which we may use instead of the original P, hoping that the former turn out to be more stable. ■

Numerous further applications of tensor factorization to recommendation engines are conceivable.

9.2 More Tensor Factorizations

Despite its power, the HOSVD is quite complex. Hence, the question arises of whether there are simpler and less computationally intensive decompositions. Indeed, this is possible in many cases and we shall address some important factorizations in the following.

9.2.1 CANDECOMP/PARAFAC

If we simplify the core tensor C of a Tucker tensor to a diagonal tensor with

$$c_{\mathbf{i}} = \begin{cases} 1, & i_1 = \ldots = i_d, \\ 0, & \text{else} \end{cases},$$

we obtain the canonical decomposition (CANDECOMP) instead of (9.1). It is also known as Parallel Factor Analysis (PARAFAC), so we call it CANDECOMP/PARAFAC (CP).

Definition 9.5 An **n**-dimensional d-mode CP-tensor of (canonical) rank-t is a tensor

$$A = U^1 \times_1 \ldots \times_d U^d, \tag{9.6}$$

where $U^k \in \mathfrak{R}^{(n_k, t)}, k \in \underline{d}$ are the mode factors.

Remark 9.2 A CP-tensor is completely determined by its mode factors. For notational reason that should become clear in the following, we shall henceforth place the mode index in an upper right position, i.e., U^k as opposed to U_k.

By decomposing the mode factors along the rank index into vectors $u_j^k = \left(U_{i,j}^k\right)_{i \in \underline{n}_k}$, we may write the CP-tensor as follows:

$$A = U^1 \times_1 \ldots \times_d U^d = \sum_{j=1}^{t} u_j^1 \circ \ldots \circ u_j^d. \tag{9.7}$$

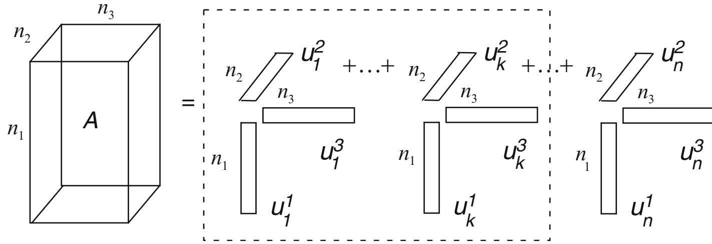


Fig. 9.5 Illustration of the CP-decomposition for a 3-mode tensor A. The *dashed lines* indicate the boundaries of a rank-k factorization

Here, the symbol "∘" represents the outer vector product, i.e., each element of a tensor $A = v^1 \circ \ldots \circ v^d$ with vectors $v^k = \left(v_i^k\right)_{i \in \underline{n_k}}$, which is called a *CP rank-1-tensor*, is the product of its corresponding vector elements

$$a_{i_1,\ldots,i_d} = v_{i_1}^1 v_{i_2}^2 \ldots v_{i_d}^d.$$

Thus, we may write (9.7) component-wise:

$$a_{i_1,\ldots i_d} = \sum_{j=1}^{t} u_{i_1,j}^1 \cdots u_{i_d,j}^d.$$

Therefore, (9.7) tells us that each CP-tensor may be represented by a sum of rank-1 tensors (Fig. 9.5).

For $d = 2$, the HOSVD, i.e., the "classical" SVD, coincides with a CP-decomposition, since the "core tensor" S_k in (8.14) is a diagonal matrix. Though it is not the identity matrix, we may multiply the diagonal values into the matrices of singular vectors, for example, into V_k.

Example 9.7 The rank-2 approximation from Example 8.4 may be written as a sum of two rank-1 tensors:

$$A_k = XY = \begin{pmatrix} 1 & 0.3 \\ 0.2 & -0.7 \\ 0.1 & -0.7 \end{pmatrix} \begin{pmatrix} 0.23 & 2.76 & 9.65 & 5.17 \\ -0.69 & -6.6 & 1.65 & -0.14 \end{pmatrix},$$

$$= \underbrace{\begin{pmatrix} 1 \\ 0.2 \\ 0.1 \end{pmatrix}}_{u_1^1} \underbrace{\begin{pmatrix} 0.23 & 2.76 & 9.65 & 5.17 \end{pmatrix}}_{\left(u_1^2\right)^T} + \underbrace{\begin{pmatrix} 0.3 \\ -0.7 \\ -0.7 \end{pmatrix}}_{u_1^1} \underbrace{\begin{pmatrix} -0.69 & -6.6 & 1.65 & -0.14 \end{pmatrix}}_{\left(u_2^2\right)^T}$$

$$= u_1^1 \circ u_1^2$$
$$+ u_2^1 \circ u_2^2.$$

■

We have not yet addressed computation of the CP-decomposition. Obviously, we can no longer apply the hitherto used adaptive HOSVD, since it is based on the Tucker decomposition (9.1). We shall address the topic in more detail in Sect. 9.2.3.

9.2.2 RE-Specific Factorizations

Besides CANDECOMP/PARAFAC, of course, numerous factorizations are possible. With regard to recommendation engines, methods of nonnegative tensor factorization, corresponding to the NMF from Sect. 8.4.3, are of special interest. In the context of reinforcement learning, we are especially interested in factorizing the transition probabilities.

As for this, an interesting approach may be found in [RFST10]. Therein, sequences of baskets belonging to different users are analyzed with the goal of recommending products that are most likely to be purchased to an identified user. To this end, the transition probabilities are factorized.

We now retrofit the approach in such a way that sequences of products instead of sequences of baskets and sessions instead of users be considered. This is consistent with our Example 9.5. Hence, we seek after a factorization of the transition probability tensor $P_{u,ss'}$, which corresponds to our previously considered matrix $P_{ss'}$ of transition probabilities from s to s' *for the session u*. The proposed factorization is of the form

$$A = \sum_{k=1}^{t_{u,s}} v_k^{u,s} \circ v_k^{s,u} + \sum_{k=1}^{t_{s,s'}} v_k^{s,s'} \circ v_k^{s',s} + \sum_{k=1}^{t_{u,s'}} v_k^{u,s'} \circ v_k^{s',u}. \tag{9.8}$$

The factorization models the pair-wise interaction between the single tensor modes u, s, s'. Therefore, we are dealing with a special case of the CP-decomposition (9.7), where the rank-1 tensors are now established from 2 rather than 3 vectors.

Writing (9.8) element-wise,

$$a_{u,ss'} = \sum_{k=1}^{t_{u,s}} v_{u,k}^{u,s} v_{s,k}^{s,u} + \sum_{k=1}^{t_{s,s'}} v_{s,k}^{s,s'} v_{s',k}^{s',s} + \sum_{k=1}^{t_{u,s'}} v_{u,k}^{u,s'} v_{s',k}^{s',u}$$

and considering the difference between two probabilities with respect to s', i.e., $a_{u,ss'} - a_{u,ss''}$, we notice that the latter is invariant with respect to the first term. Hence, if one is interested only in the ordering of the values $a_{u,ss'}, s' \in S$, it suffices to consider

$$\tilde{a}_{u,ss'} := \sum_{k=1}^{t_{s,s'}} v_{s,k}^{s,s'} v_{s,k}^{s',s} + \sum_{k=1}^{t_{u,s'}} v_{u,k}^{u,s'} v_{s',k}^{s',u}, s' \in S. \tag{9.9}$$

Intuitively, this may be put as

$$\tilde{a}_{u,ss'} = \tilde{a}_{u,ss'}^{MC} + \tilde{a}_{u,ss'}^{CF}. \tag{9.10}$$

Here, the first term *MC* corresponds to a Markov transition, as considered in Chaps. 3, 4, 5 and 6, and the second one, CF, to the approach of PCA-based collaborative filtering, which has been presented in the previous chapter by means of Example 8.1. Thus, this factorization unifies both approaches in a simple manner. Since the parameters of both modes are learned jointly, the approach is by no means trivial.

Even though the approach appears simple, it brings along a fair amount of difficulties. First, the question of approximation error arises, since, as a matter of fact, the approach brings about a great deal of simplification with arguable plausibility. Furthermore, the factorized "probabilities" are, of course, no longer stochastic and thus not probabilities. Granted, the authors of [RFST10] perform an additional transformation by a sigmoid function such that

$$0 \le a_{u,ss'} \le 1, \forall s, s' \in S$$

holds. But even then the row sum condition (3.2) is violated. With regard to the goals of [RFST10], this is not an issue, since the recommendations are derived directly from the probabilities and, therefore, only an ordering of these needs to be ensured. For our purposes of RL, we may not ignore this condition, but must again demand:

$$\sum_{s'} a_{u,ss'} = 1 \, \forall u \in U \wedge \forall s \in S.$$

Hence, we need to incorporate these conditions in the solution procedure. Sadly enough, this doesn't make the computation of the factorization, which is complicated enough in itself, any easier. Apart from that, another difficulty, though not related to the factorization itself, comes into play: allowing the transition probabilities to depend on the session violates Assumption 4.1 of the Markov property. In Chap. 10, we shall develop a in this respect correct approach, which models the transition probabilities as functions of the course of the session and yet satisfies the (generalized) Markov property.

9.2.3 Problems of Tensor Factorizations

There exist many other tensor decomposition methods. However, the main problem is that for most of them, unlike as for the Tucker decomposition, no efficient standard algorithms to calculate the decomposition exist. So these algorithms need to be developed.

To this end, we shall denote the sought-after decomposition of a given tensor A by A_Θ with Θ denoting the sought-after tensor components, e.g., the factors U of the CP-decomposition. Thus, the task consists in determining an optimizer Θ of

$$\min_{\Theta} f(A, A_\Theta),$$

mostly in the form

$$\min_{\Theta} \|A - A_\Theta\|_F.$$

Iterative procedures appear suitable for the solution of this optimization, since they are optimal with respect to memory usage with regard to extremely sparsely populated tensors. Their convergence rates, however, are critical.

In current literature, endless varieties of gradient descent methods (stochastic, partitioned, prioritized, etc.) are often proposed. Mostly, a case for them is their easy implementation. These procedures are then praised as “rapid” and “robust”; estimates of their convergence rates are generously forgone. Of course, the exact opposite of the promised holds true: gradient descent procedures heavily depend on the problem and parameters and often converge rather slowly. Granted, they may now and then be “fine-tuned” to the solution of a particular high-dimensional problem instance, but they fail at solving other, even tiny, problems with the same parameter assignment. All in all, there is no *satisfactory* guaranteed upper bound on the number of iterations (and thus on the computational complexity) for any general class of problem instances. Thus, gradient descent methods are suitable for certain experiments, but not for practical deployment.

To ensure the necessary convergence rate, more sophisticated iteration procedures are required, e.g., projection methods. This may be seen very clearly in the field of matrices, where Krylov-subspace procedures are employed predominantly. For symmetric matrices, this mainly comes to the conjugate gradient (CG) method, mostly in connection with a preconditioner (PCG), which may arguably be considered as the default procedure of numerical analysis. For nonsymmetric matrices, especially for the SVD, one mostly resorts to the Lanczos method, which has been presented in Sect. 8.4.1. Devising robust Lanczos procedures, though, is difficult and cannot be immediately carried over to other formulations, as it requires orthogonality of the bases. Yet there are orthogonalized approaches to the higher-order case but mostly, again, for the Tucker case. An example is the *High-Order Orthogonal Iteration* [KB09].

Most of the current procedures for computing tensor factorization are EM-like (comparable to ALS from Sect. 8.4.3). Therein, in each step a part of the tensor coefficients is kept fix, such that the resulting optimization problem with respect to the “free” variables is convex and thus can be solved easily. After solving the latter, the just-obtained coefficients are fixed in turn, and previously fixed coefficients are computed anew. The method terminates as soon as the error has fallen below a prescribed bound, or other technical termination criteria are satisfied (e.g., a maximum number of iterations has been reached). Methods of this type mostly turn out to be robust in practice, but often their convergence cannot be ensured. For canonical decompositions ALS in general converges slowly (unlike as for the Tucker decomposition where ALS mostly works efficiently).

Additionally, for non-Tucker decompositions, there may be a problem with the stability of ranks. For example, small perturbations can considerably decrease the canonical rank [DSL08]. Thus, in general CP decompositions are not an optimal choice.

All this brings us back to the Tucker decomposition again. The method for computing the HOSVD presented in Sect. 9.1.3 and its adaptive version from Sect. 9.1.4 reduce the HOSVD to a singular value decomposition of matrices. Despite all complexity, the latter has been studied comprehensively and is numerically controllable. Therefore, the presented procedure may be considered robust.

First, let us estimate the complexity of the Tucker decomposition (9.1) in more detail. For simplicity, for now and in the following, we consider all mode ranks of the Tucker rank-$\boldsymbol{t}$ to be equal, i.e., $t_k = t, k \in \underline{d}$, and now refer to t as Tucker rank. Similarly, for complexity estimates we will assume that all mode sizes are equal: $n_k = n, k \in \underline{d}$. Then the number of parameters of (9.1) is $O(dnt + t^d)$. This is acceptable for moderate dimensions, like 3 or 4, granted rank-t is not too high. So if we have an efficient algorithm that can handle large mode sizes (unlike the truncated SVD applied to all k-mode matricizations), it can be applied to large problems. In fact, such algorithms are available, e.g., the cross-approximation [OST08].

Nevertheless, for larger dimensions the classical Tucker decomposition (9.1) is definitely not suitable because of the complexity $O(t^d)$ of the core tensor. This problem is solved by the introduction of hierarchical SVD-based decompositions that will be discussed in the next section. Here, once again, the power of hierarchical approaches, which we already addressed in Chap. 6, shows up.

9.3 Hierarchical Tensor Factorization

9.3.1 Hierarchical Singular Value Decomposition

The ***H***-SVD, where “***H***” stands for “hierarchical,” is being discussed in current research [Gra10, HK09]. The main idea behind hierarchical Tucker decompositions is simple: in order to reduce the complexity of the core tensor C, we use a hierarchical split of the set of dimension indices and observe a dyadic decomposition. Let us

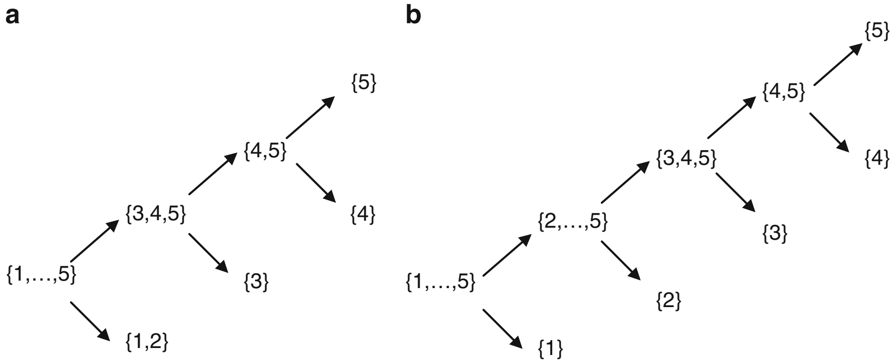


Fig. 9.6 Two examples of hierarchical decompositions of the index set for d = 5

suppose that we subdivide our index set into two subsets where the first k form the first subset and the others the second. Then we arrive at the canonic dyadic decomposition of the core tensor

$$c_{i_1,\ldots,i_k;i_{k+1},\ldots,i_d} = \sum_{s=1}^{t} c_{i_1,\ldots,i_k,s} c_{i_{k+1},\ldots,i_d,s}. \tag{9.11}$$

This reduces the d-dimensional core tensor to the new tensors of dimensions $k + 1$ and $d–k + 1$. The resulting Tucker tensor is

$$a_{i_1,\ldots,i_d} = \sum_{j_1=1}^{t} \cdots \sum_{j_d=1}^{t} \sum_{s=1}^{t} c_{i_1,\ldots,i_k,s} c_{i_{k+1},\ldots,i_d,s} u^1_{i_1,j_1} \cdots u^d_{i_d,j_d}$$

By successively applying decompositions (9.11), we arrive at a hierarchical Tucker decomposition. If we construct the tree such that the index sets of all leafs contain one index only (Fig. 9.6), for the resulting Tucker decomposition, the dependence on d is linear!

Now it is possible to prove that for hierarchical decompositions like the ***H***-SVD, important properties of the Tucker decomposition are retained [Gra10]. This includes the availability of standard computation algorithms, efficient truncation, and stability. We will study this in more detail in the next section which is devoted to an important type of hierarchical Tucker decompositions – the tensor train.

9.3.2 Tensor-Train Decomposition

A hierarchical decomposition of the index set defined by a binary three where each split is done into a node corresponding to a single index and into a node of remaining

nodes, like in Fig. 9.6b, defines a *tensor-train* (TT) decomposition. It has the following form:

$$a_{i_1,\ldots,i_d} = \sum_{\alpha_1=1}^{t_1} \cdots \sum_{\alpha_{d-1}=1}^{t_d} g_{1(1,\alpha_1)}^{i_1} g_{2(\alpha_1,\alpha_2)}^{i_2} \cdots g_{d-1(\alpha_{d-2},\alpha_{d-1})}^{i_{d-1}} g_{d(\alpha_{d-1},1)}^{i_d}. \tag{9.12}$$

Here we again allow the ranks t_k to be different for each mode. For the TT decomposition, we call the ranks t_k *compression ranks*.

In matrix form, (9.12) can be represented as a product of matrices

$$a_{i_1,\ldots,i_d} = G_1^{i_1} G_2^{i_2} \ldots G_{d-1}^{i_{d-1}} G_d^{i_d}, \tag{9.13}$$

where $G_k^{i_k}$ is a matrix of size $t_{k-1} \times t_k$. Note that the first matrix $G_1^{i_1}$ is a row of dimension $1 \times t_1$ and the last matrix $G_d^{i_d}$ is a column of dimension $t_d \times 1$. Thus, the product of the d matrices is a 1 x 1 matrix, i.e., a number.

For equal mode sizes, $n_k = n, k \in \underline{d}$, we have dn matrices $G_k^{i_k}$ and the number of parameters of (9.13) is bounded by $(d-2)nt^2 + 2nt$, where $t = \max_k t_k$, and thus the dependence on d is linear.

The following algorithm TT-SVD calculates a TT decomposition based on truncated SVDs over all modes and can be considered as counterpart to the truncated HOSVD algorithm 9.1.

Algorithm 9.3 TT-SVD

Input: tensor $A \in \Re^{\mathbf{n}}$, truncation rank $\mathbf{t} \le \mathbf{n}$

Output: cores $G_1, \ldots, G_d$ of TT approximation

1: temporary tensor $B := A$, $t_0 := 1$, $N_0 := \prod_{k=1}^{d} n_k$
2: **for** $k = 1,\ldots,d-1$
3: calculate the dimension $N_k = \frac{N_{k-1}}{t_{k-1} n_k}$
3: unfold B into the dimensions $B = B \in \Re^{t_{k-1} n_k \times N_k}$
4: compute rank-t_k truncated SVD $B \approx USV$ of B
5: reshape U such that $G_k := U \in \Re^{t_{k-1} \times n_k \times t_k}$
6: $B := SV^T$
7: **end for**
8: $G_d := B$

Moreover, Algorithm 9.3 can be extended for automatic selection of truncation ranks. At this, we first introduce a bound δ. Now in step 4 of Algorithm 9.3, we determine the minimal rank t_k such that

$$B = USV + E, \|E\|_F \le \delta. \tag{9.14}$$

Then the following theorem holds.

Theorem 9.2 (Theorem 2.2 in [Os11]) *For each tensor* $A \in \Re^{\mathbf{n}}$, *the TT-SVD with* (9.14) *computes a tensor T in the TT format with compression ranks* t_k *such that:*

$$\|A - T\|_F \leq \sqrt{\sum_{k=1}^{d-1} \varepsilon_k^2},$$

where $\varepsilon_k = \|E_k\|_F$ *of all unfoldings* $A_k = R_k + E_k$ *as in step 4 of the TT algorithm.*

From Theorem 9.2 the following corollary can de deduced [Os11].

Corollary 9.1 *Given a tensor* $A \in \Re^{\mathbf{n}}$ *and rank bounds* t_k, *the best approximation to A in the Frobenius norm with TT-ranks bounded by* t_k *(denoted by* T^**) always exists, and the TT-approximation T computed by the TT-SVD algorithm is quasi-optimal:*

$$\|A - T\|_F \leq \sqrt{d-1}\|A - T^*\|_F.$$

Thus from Theorem 9.2, it immediately follows that for a prescribed relative accuracy ε of the TT-SVD algorithm, we just need to select

$$\delta = \frac{\varepsilon}{\sqrt{d-1}}\|A\|_F.$$

This is a very nice (and constructive!) result. Remember that the truncated HOSVD is in general not the optimal Tucker decomposition, although it usually shows good approximation properties. In contrast, the TT-SVD provides us with the (quasi-) optimal TT decomposition.

The tensor-train decomposition also possesses many other advantages; see [Os11, OT09]. Unlike the canonical decomposition, it has stable ranks. Basic tensor operations (Sect. 9.1.1) can be efficiently implemented. Here, efficient recompression procedures play an important role since many basic linear algebra operations with TT-tensors (addition, matrix-by-vector product, etc.) yield increased ranks. *Recompression* (or *rounding*) describes the rank reduction if a tensor is already given in TT format but with suboptimal ranks t_k (i.e., too large). Ivan Oseledets, who is – along with Eugene Tyrtyshnikov – one of the pioneers in the TT area, developed a general TT recompression algorithm with linear complexity in d and n.

Of course, the TT-SVD algorithm is not suited for large mode sizes, for the same reason as the HOSVD: the unfolding matrices are usually too large. But, like in the 3D-Tucker case, other techniques can be used. In fact, recall that due to the complexity bound $(d-2)nt^2 + 2nt$, the TT format is linear in both d and n. So provided the compression ranks are not too high, it can be efficiently used to approximate high-dimensional problems of high mode sizes. The development of efficient algorithms for the TT approximation is currently under way. Beside ALS, again the cross-approximation technique [OT10] is very powerful. In order to determine optimal compression ranks, the DMRG scheme looks promising [SO11].

9.4 Summary

We have generalized the factorization concept, introduced in the last chapter for matrices, to the case of higher dimensions – tensors. We were also able to develop an incremental tensor factorization approach, the incremental HOSVD, based on the incremental SVD. First, experimental results indicate that the tensor factorization is useful for our RE approach.

The main challenge of tensor factorization is complexity. This is a highly demanding task, which certainly will keep researchers occupied for many years to come. For the Tucker model, namely, for the HOSVD, there are some in principle efficient algorithms along with crisp convergence propositions. In particular, this applies to the Lanczos method as well as Brand's incremental SVD. Yet, these methods are complex as regards implementation and computationally intensive in the high-dimensional case. Moreover, the Tucker model itself suffers from the curse of dimensionality since its complexity is exponentially growing with the number of dimensions d.

Thus, simpler factorization models, first of all the canonical decomposition, look quite appealing at the first sight. The complexity of the canonical decomposition grows only linearly with d. However, here we face other problems. Unlike the Tucker decomposition, it is not stable concerning the rank and no general algorithms for its efficient computation exist.

This turns the attention of researchers back to the Tucker decomposition and SVD-type approaches. In fact, here hierarchical decompositions seem to be the solution. Especially the tensor-train decomposition has emerged as efficient instrument to break the curse of dimensionality. It is relatively simple. Like the canonical decomposition, the number of parameters is linear in d. At the same time it shares positive properties with Tucker: TT decompositions have stable ranks. They can be computed based on SVD procedures, rank reduction can be provided efficiently, etc. So the main problem is the development of algorithms to efficiently calculate TT decompositions. This development is being carried out fiercely at present, and a couple of interesting methods have already emerged in the field.

This also applies to many other tensor decomposition algorithms. Many of them, however, are fairly empirical, lacking theoretically valid estimates of convergence rates, alas, even a proof of convergence at all. Hence, it is still difficult to assess the practical eligibility of these recent methods.

In spite of all of the above-outlined difficulties, the meaning of tensor factorization as an approximation tool of future recommendation engines is obvious. This very field is currently undergoing a turbulent development. We shall therefore return to tensor factorization at some point in the next chapter, so as to make an attempt at combining it with reinforcement learning.

Chapter 10
The Big Picture: Toward a Synthesis of RL and Adaptive Tensor Factorization

Abstract We explore the subject of uniting the control-theoretic with the factorization-based approach to recommendation, arguing that tensor factorization may be employed to vanquish combinatorial complexity impediments related to more sophisticated MDP models that take a history of previous states rather than one single state into account. Specifically, we introduce a tensor representation of transition probabilities of Markov-k-processes and devise a Tucker-based approximation architecture that relies crucially on the notion of an aggregation basis described in Chap. 6. As our method requires a partitioning of the set of state transition histories, we are left with the challenge of how to determine a suitable partitioning, for which we propose a genetic algorithm.

In this research-oriented chapter, we shall study a refinement of the concept of a Markov decision process which enables a recommendation engine to incorporate sequences of previously visited products rather than making decision exclusively upon the current state. As foreshadowed in Chap. 8, more sophisticated models of this kind entail some complexity issues. Therefore, so as to vanquish the latter, we shall introduce a tensor factorization-based approximation framework. The reasoning provided in this chapter is thus a step toward a unification of classical (factorization based) data mining on one hand and the novel control-theoretic framework on the other hand. We should stress, however, that the approach presented in the following is currently still in its infancy and a subject of ongoing research. Hence, a major part of the subsequent elaborations are still based upon speculation rather than scientific rigor.

A. Paprotny and M. Thess, *Realtime Data Mining: Self-Learning Techniques for Recommendation Engines*, Applied and Numerical Harmonic Analysis,
DOI 10.1007/978-3-319-01321-3_10, © Springer International Publishing Switzerland 2013

10.1 Markov-k-Processes and Augmented State Spaces

The notion of a k-Markov decision process (k-MDP) is a generalization of that of an MDP. The generalized framework enables to formulate control problems which involve environments where transition probabilities, rewards, and policies depend on the sequence of the k most recently visited states rather than only the current one. We shall refer to this assumption as the *generalized Markov property*.

Assumption 10.1 (Generalized Markov property): In each state, the choice of the optimal action depends exclusively on the k most recently visited states.

Hence, given state and action spaces S and A, the dynamics are characterized by transition probabilities of the form

$$p^a_{s_1,\ldots,s_k,s'}, s_1, \ldots, s_k, s' \in S, a \in A,$$

which translates into plain English as "The probability of a transition to state s' given action a, current state s_k and previous states $s_1, \ldots, s_{k-1}$." Similarly, policies and the reward function are of the form

$$r^a_{s_1,\ldots,s_k,s'}$$

and

$$\pi(s_1, \ldots, s_k, a),$$

respectively. It is worth stressing that by assigning $k := 1$, we recover the classical MDP framework.

It comes as a bit of a surprise that in some theoretical sense, there is no distinction between the classical and the k-MDP case. Given a k-MDP, it is always possible to devise an equivalent MDP by means of a construction which we shall refer to as *state space augmentation*. (The precise meaning of the term *equivalent* will become clear in the course of the subsequent discussion.)

Given the state space S of some k-MDP M, we define the corresponding *augmented state space* of order k by

$$\widetilde{S} := \bigcup_{j=1}^{k} s^j.$$

Upon this state space, we model an MDP $\widetilde{M} := \left(\widetilde{S}, A, \widetilde{p}, \widetilde{r}\right)$ as follows: for $\mathbf{s} := (s_1, \ldots, s_l)$, let

$$\mathbf{s}_* := (s_2, \ldots, s_l).$$

Then the transition probabilities of $\widetilde{M}$ are stipulated as

$$\widetilde{p}^{\,a}_{s(s',s')} := \begin{cases} p^a_{\mathbf{s},s'}, & \mathbf{s}' = \mathbf{s}_*, \\ 0, & \text{otherwise}, \end{cases} \quad a \in A.$$

Similarly, the rewards are taken to be

$$\widetilde{r}^{\,a}_{\mathbf{s}(\mathbf{s}',s')} := \begin{cases} r_{\mathbf{s},s'}, & \mathbf{s}' = \mathbf{s}_*, \\ 0, & \text{otherwise}, \end{cases} \quad a \in A.$$

Now let $\{s_j\}_{j \in \underline{N}} \subset S$ be a trajectory of M and define

$$g : S^N \to \widetilde{S}^N : \{s_j\}_{j \in \underline{N}} \mapsto \{(s_{j-k+1}, \ldots, s_j)\}_{j \in \underline{N}},$$

where we made use of the convention

$$(s_{j-k+1}, \ldots, s_j) := (s_1, \ldots, s_j), j - k + 1 < 1.$$

Conversely, consider

$$h : \widetilde{S}^N \to S^N : \{(\mathbf{s}^j, s_j)\}_{j \in \underline{N}} \mapsto \{s_j\}_{j \in \underline{N}}.$$

Then we have $h \circ g = \mathbf{id}$, i.e., the identical mapping, and all trajectories of $\widetilde{S}$ not contained in $h(S^N)$ have vanishing probability. Furthermore, any trajectory of M has the same probability and reward sequence as its image under h.

By virtue of this result, it is straightforward to verify that the state-value function v of a given policy π satisfies the Bellman equation

$$v^\pi_{s_1,\ldots,s_l} = \sum_{a \in A} \pi(s_1, \ldots, s_l) \sum p^a_{s_1,\ldots,s_l s'} \left[r^a_{s_1,\ldots,s_l s'} + \gamma v_{s_2,\ldots,s_l s'}\left(s'\right) \right] \tag{10.1}$$

Also by means of state space augmentation, we may devise a k-MDP generalization of temporal-difference learning. Given a transition from state s to s' given the history $\mathbf{s} = (s_1, \ldots, s_{l-1})$, the update rule reads as

$$v := v + \alpha z d(v), \tag{10.2}$$

Where

$$d(v) := r_{(\mathbf{s},s),s'} - \left(v_{(\mathbf{s},s)} - \gamma v_{(\mathbf{s}_*,s')} \right), z := \lambda \gamma z + e_{(\mathbf{s},s)}. \tag{10.3}$$

10.2 Breaking the Curse of Dimensionality: A Tensor View on Augmented State Spaces

Unfortunately, the complexity of k-MDP models with $k > 1$ renders a direct computational treatment intractable. Specifically, the dimensionality of the corresponding Bellman equations grows exponentially with k. Hence, we need to settle for an approximate solution based on a suitably reduced model. We shall introduce a tensor-based model reduction framework for k-MDPs in what follows. To this end, define

$$\hat{S} := \bigcup_{j=1}^{k-1} S^j,$$

and

$$\hat{p} \in \mathfrak{R}^{\hat{s} \times s \times s}, \hat{p}^{\,a}_{\mathbf{s}ss'} := \hat{p}^{\,a}_{(s_1,\ldots,s_l)ss'} := \hat{p}^{\,a}_{s_1,\ldots,s_l,s,s'}, l < k, s_1, \ldots, s_l, s, s' \in S.$$

This rendition endows us with a three-mode tensor view of the transition probabilities. In precisely the same fashion, we introduce a tensor view $\hat{r}$ on the rewards. Consider the following Tucker-style factorization model (we shall omit modes corresponding to actions in the following):

$$\hat{p}_{\mathbf{s}ss'} \approx U \otimes_1 C = \sum_{\beta \in \underline{m}} u_{\mathbf{s}\beta} c_{ss'\beta}, \tag{10.4}$$

where $U \in \mathfrak{R}^{\hat{s} \times m}, C \in \mathfrak{R}^{m \times s \times s}$ with $m \ll |\hat{S}|$. See Fig. 10.1 for a graphical representation of this model.

We shall focus on the case where U is taken to be an aggregation prolongator, i.e.,

$$u_{\mathbf{s}\beta} = \Theta_{\mathbf{s} \in G_\beta}, \mathbf{s} \in \hat{S}, \beta \in \underline{m} \tag{10.5}$$

for some partition $\{G_\beta\}_{\beta \in \underline{m}}$ of $\hat{S}$ (see also Sect. 6.2.1). How are we supposed to pick the partition and the core tensor C? Basically, we have to meet three requirements:

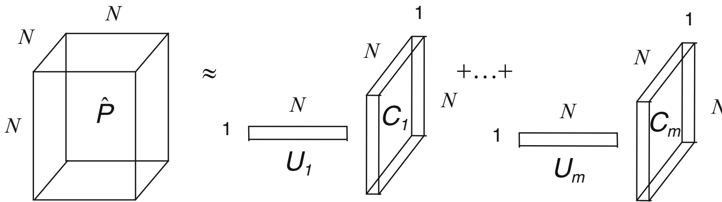


Fig. 10.1 Illustration of the factorization model of the augmented state spaces for k = 2 and l = 1. Here, we denote $U_\beta = (u_{\mathbf{s}\beta})_{\mathbf{s} \in \hat{S}}, c_\beta = \left(c_{ss'\beta}\right)_{s,s' \in S}$.

1. **Complexity**: $m = O(k^q)$ for some $q \in \mathbf{N}$, i.e., m should increase at most polynomially with the order k.
2. **Approximation**: $\left\| \left(\hat{p}_{\mathbf{s}s'} - \sum_{\beta \in \underline{m}} u_{\mathbf{s}\beta} c_{ss'\beta} \right)_{\mathbf{s}s'} \right\|$ should be small with respect to an appropriate norm $\| \cdot \|$.
3. **Consistency**: Each slice obtained by fixing the index corresponding to the first mode should be a row-stochastic matrix.

In a nutshell, fulfilling the above requirements amounts to reducing complexity under preservation of as much information as possible.

Let us consider the case where (the matricization of) the core tensor is taken to be

$$C^{(1)} := U^{+} \hat{P}^{(1)},$$

where U^{+} denotes the Moore-Penrose pseudo-inverse (MPP) of U with respect to the canonical inner product, i.e.,

$$U^{+} := \left(U^T U\right)^{-1} U^T = \left(\left|G_\beta\right|^{-1}\right)_{\beta \in \underline{m}} U^T. \tag{10.6}$$

The complexity requirement may be controlled by a suitable choice of the partition. As regards approximation, it follows from the definition of the MPP that the factorization is a minimizer of

$$\left\|\hat{P} - U \otimes_1 C\right\|_F = \min_{C \in \Re^{m \times s \times s}} \left\|\hat{P} - U \otimes_1 \overline{C}\right\|_F.$$

Finally, consistency follows from the fact that each 1 slice of $U \otimes_1 C$ is a convex combination of row-stochastic matrices.

Besides its fulfilling our requirements, a crucial case for this factorization approach can be made in virtue of its intuitive interpretation. Specifically, the second identity in equation (10.6) reveals that the factorization is an algebraic representation of replacing each slice by the centroid of the slices of $\hat{P}$ in the corresponding class of the partition. This is the procedure of vector quantization that we already had used in Sect. 8.6.

Thus, in case we already know the transition probabilities $\hat{P}$, we can apply the k-means algorithm to find the best partition $\{G_\beta\}_{\beta \in \underline{m}}$ for a given partition number m. Here, $k = m$ is used in the k-means algorithm, and the slices of $\hat{P}$ are treated as vectors, i.e., vectorization is applied to the slice matrices in a similar way as matricization to tensors.

Example 10.1 In the following, we shall illustrate the proposed factorization by means of a simple example. For the sake of simplicity, we shall ignore the actions

and identify the states with their indices. To this end, we consider the following 3 sessions with altogether 3 products (for simplicity, we forgo the absorbing state):

- $1 \to 3 \to 2 \to 3$
- $2 \to 1 \to 2 \to 3 \to 1$
- $1 \to 1 \to 2 \to 2 \to 3 \to 3 \to 1 \to 1$

This yields the following global matrix of transition probabilities for $k = 1$:

$$p = \begin{bmatrix} \frac{2}{5} & \frac{2}{5} & \frac{1}{5} \\ \frac{1}{5} & \frac{1}{5} & \frac{3}{5} \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{4} \end{bmatrix}$$

Subsequently, we consider $k = 2$, i.e., the transition probability does not only depend on the current product, but also on its predecessor. Then we obtain the following tensor of transition probabilities $\hat{P}$:

$$\hat{p}_{(1)} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad \hat{p}_{(2)} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ \frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix}, \quad \hat{p}_{(3)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

where $\hat{p}_{(i)} = \left(\hat{p}_{(i)ss'}\right)_{s,s' \in S}$ denotes the matrix of transition probabilities if i has been visited previously. Let us, e.g., consider the entry with index (3,1) of $\hat{p}_{(2)}$, i.e., $\hat{p}_{(2)3,1} = \frac{1}{2}$: after 2 had been visited, a transition to 3 occurred altogether twice. Then, from there, there was one transition to 1 (and one transition to 3). Thus, the probability of a transition from 3 to 1 given that 2 has been considered previously is precisely 0.5.

We now choose the following partition of the state space $S = \{1,2,3\}$, $m = 2$:

$$G_1 = \{1, 2\}, G_2 = \{3\}$$

and thus obtain the corresponding aggregation prolongator and its pseudo-inverse

$$U = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad U^+ = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

We now determine the core tensor C, whereat we matricify $\hat{P}$:

$$C^{(1)} = U^{+}\hat{p}^{(1)} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & \frac{1}{2} & 0 & \frac{1}{2} \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

$$C^{(1)} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

In matricified form, our Tucker tensor $\widetilde{P}$ then turns out to be

$$\widetilde{P}^{(1)} = (U \otimes_1 C)^{(1)} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

$$\widetilde{P}^{(1)} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ 0 & 1 & 0 & 0 & 0 & 1 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Hence, the resulting transition probabilities are

$$\widetilde{P}_{(1)} = \widetilde{P}_{(2)} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix}, \quad \widetilde{P}_{(3)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

As compared to $\hat{P}$, we see that the latter is approximated very well. Indeed, $\widetilde{P}_{(i)}$ and $\hat{P}_{(i)}$ for $i = 1,2$ deviate from each other only in the last row, and for $i = 3$ they are even identical. The Frobenius error of our approximation turns out to be

$$\left\|\hat{P} - \widetilde{P}\right\|_F = \sqrt{\frac{3}{4}} \approx 0.87.$$

Similarly, we may also choose a different partition. For example, we obtain for $G_1 = \{1,3\}$, $G_2 = \{2\}$

$$\widetilde{P}_{(1)} = \widetilde{P}_{(3)} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 1 \\ 0 & \frac{1}{2} & \frac{1}{2} \end{bmatrix}, \quad \widetilde{P}_{(2)} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ \frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix}$$

and the approximation error is higher,

$$\left\|\hat{P} - \widetilde{P}\right\|_F \approx 1.58.$$

The k-means algorithm ($k = 2$) applied to this problem automatically finds the first partition $G_1 = \{1,2\}$, $G_2 = \{3\}$ which is in fact the best one.

If we, by the way, use the global transition probabilities P for all i, i.e., $\widetilde{P}_{(i)} = P$, this results, as expected, in a higher approximation error of approximately 2.07. ■

10.3 Estimation of Factorized Transition Probabilities

In what follows, we shall present a procedure to estimate the factorized transition probabilities in an adaptive online fashion, that is, a method doing without any representation of an estimate of the full transition probability tensor.

To this end, recall the update rule for the transition probabilities of a classical MDP presented in Chap. 3. In virtue of state space augmentation, this rule and the corresponding convergence result carry over to the k-MDP case as follows:

$$\overline{P}_{\mathbf{s}s'} := \left(1 - t^{-1}\right)\overline{P}_{\mathbf{s}s'} + t^{-1}.$$

In tensor notation, the update rule reads as

$$\overline{P} := \left(1 - t^{-1}\right)\overline{P} + t^{-1} e_{\mathbf{s}} \otimes e_{\mathbf{s}} \otimes e_{s'},$$

where

$$(e_i)_j = \begin{cases} 1, & i = j, \\ 0, & i \neq j. \end{cases}$$

Matricifying with respect to the first mode and pre-multiplying with U^+ eventually gives rise to the update rule

$$\overline{C} := \left(1 - t^{-1}\right)C^{-1} + t^{-1}\frac{e_{\beta(\mathbf{s})}}{G_{\beta(\mathbf{s})}} \otimes e_s \otimes e_{s'}, \tag{10.7}$$

where $\beta(\mathbf{s})$ denotes the unique index $\beta \in \underline{m}$ satisfying $\mathbf{s} \in G_\beta$, for the core tensor. In index notation, this corresponds to

$$C_{ss'\beta(\mathbf{s})} := \left(1 - t^{-1}\right)C_{ss'\beta(\mathbf{s})} + t^{-1}. \tag{10.8}$$

Convergence of the update rule follows immediately from convergence of that for $\overline{P}$ and partial continuity of the multilinear product.

Besides its computational use, the update rule also reveals another interesting property of the factorization model: for the trivial partition, that is, the partition

consisting of only one set, the update rule coincides with that for a classical MDP. Hence, our factorization model incorporates the case where a k-MDP is approximated by a 1-MDP model. This bears an epistemic value with regard to the assessment of the quality of 1-MDP models in environments that actually satisfy a GMA with $k > 1$. Specifically, with regard to recommendation environments, which, arguably, may be assumed to be more accurately represented by a k-MDP, this insight may enable us to assess the quality of the classical MDP models discussed in foregoing chapters. For example, we may obtain bounds on the modeling error entailed by employing a classical MDP model from bounds on the approximation error of the factorized representation. Admittedly, we are as yet in no position to produce such error bounds here. Hence, we leave the topic for future research.

10.4 Factored Representation and Computation of the State Values

10.4.1 A Model-Based Approach

In the following, we shall be interested in approximations of the form

$$v_{\mathbf{s}s} \approx \sum_{\beta \in \underline{m}} u_{\mathbf{s}\beta} \theta_{s\beta} = \theta_{s\beta} \tag{10.9}$$

to the state-value function. Here, U denotes an aggregation prolongator as introduced in Equation (10.5). In order to solve the Bellman equation (10.1) approximately, we devise the least squares approach

$$\min_{\theta} \sum_{\mathbf{s} \in \hat{S}, s \in S} \left(\theta_{s\beta(\mathbf{s})} - \gamma \sum_{s' \in S} c_{ss'\beta(\mathbf{s})} \theta_{s'\beta(\mathbf{s}_* s)} - b_{\mathbf{s}s} \right)^2, \tag{10.10}$$

which is obtained by inserting the factorized representations (10.9) and (10.4), with U taken to be the aggregation prolongator defined in (10.5), for v and $\hat{P}$ in the least squares version of (10.1),

$$\min_{v} \sum_{\mathbf{s} \in \hat{S}, s \in S} \left(v_{\mathbf{s}s} - \gamma \sum_{s' \in S} p_{\mathbf{s}ss'} v_{(\mathbf{s}_* s)s'} \right)^2.$$

As regards practical computation in a recommendation framework, one may proceed as follows: first, the core tensor C is estimated from observation by means of the updating procedure (10.7). Eventually, Equation (10.10) may be solved by means of numerical linear algebra.

10.4.2 Model-Free Computation in Virtue of TD (λ) with Function Approximation

We shall be interested in a model-free stochastic iteration scheme for computing an approximation of the form (10.9) which does not rely on a factored representation of the transition probabilities. Though of less practical significance, we shall first attend to discounted problems without terminal state for the sake of simplicity.

Recall the function space framework (6.1). It is possible to consider the factorization of the state-value function (10.9) as a linear function approximation in terms of the basis

$$\phi_{\alpha\beta} : \widetilde{S} \to \Re, \phi_{\alpha\beta}(\boldsymbol{s}, s) = u_{\mathbf{s}\beta}\delta_{\alpha s} = \delta_{\beta(\mathbf{s})=\beta, \alpha=s}$$

for $\alpha \in S, \beta \in \underline{m}$. In this view, the coefficient corresponding to $\phi_{\alpha\beta}$ is given by $\theta_{\alpha\beta}$.

Extensions of the temporal-difference learning algorithms presented in Chap. 3 working with an approximate representation in terms of a linear architecture

$$v \approx \Phi\theta = \sum_j \phi_j \theta_j$$

are presented and discussed in [BT96, TVR97]. Further generalizations incorporating the multigrid framework introduced in Chap. 6 are extensively studied in [Pap11, Ziv04, ZS05]. A model-free update rule, which we state in terms of the state-value rather than the action-value function of the given policy for the sake of simplicity, corresponding to (10.2) is given by

$$\theta := \theta + \alpha\Phi^T z d(\Phi\theta), \tag{10.11}$$

where $d(\cdot)$, z are as stipulated in (10.3).

A mathematically inclined reader may be interested in the following convergence result:

Theorem 10.1 [BT96, TVR97] *Let Φ have linearly independent columns and $\lambda \in [0,1]$. Furthermore, let $A_\lambda := (I - \gamma\lambda P)^{-1}(I - \gamma P)$. Then, under the same assumptions as for ordinary TD(λ), the sequence of iterates generated by the update rule (10.11) converges a.s. to*

$$\theta_\lambda := \left(\Phi^T D A_\lambda \Phi\right)^{-1} \Phi^T D A_\lambda A_0^{-1} \Phi b.$$

Moreover, it holds that $\theta_1 := \text{argmin}_\theta \|v - \Phi\theta\|_D$, and

$$\frac{\|\Phi\theta_\lambda - v\|_D}{\|\Phi\theta_1 - v\|_D} \leq \frac{1 - \lambda\gamma}{\sqrt{(1-\gamma)(1+\gamma-2\gamma\lambda)}}$$

where $\|\cdot\|_D$ denotes the norm corresponding to the inner product induced by the multiplication operator of the steady-state probabilities of the Markov chain.

Let us now return to the special case where the transfer tensor U is taken to be an aggregation prolongator. Intuitively, the following proposition, which, once stated, is obvious, tells us that refining the underlying partition results in refining the corresponding function space. Specifically, we say that a partition G′ is a refinement of a partition G if each element of G′ is a subset of some element of G. In other words, G′ is obtained from subdividing elements of G into smaller sets.

Proposition 10.1 Let G,G' be partitions with corresponding aggregation prolongators U,U'. If G' is a refinement of G, then the range of U is contained in that of U'.

Proof All we have to show is that each column of U may be written as some linear combination of columns of U′. To this end, consider $\beta \in G$ and let $\beta'_1, \ldots, \beta'_l \in G'$ satisfy

$$\beta = \beta'_1 \cup \ldots \cup \beta'_l.$$

This together with the fact that $\beta'_1, \ldots, \beta'_l$ are disjoint yields

$$U_\beta = U'_{\beta'_1} + \ldots + U'_{\beta'_l}. \qquad \square$$

In particular, this result ensures that refining a partition cannot cause any deterioration of the approximation. Yet another straightforward calculation yields the following crucial and, at the same time, rather astonishing insight.

Proposition 10.2 *Approximate TD(λ) for k-MDPs with the approximation architecture induced by the Tucker model with transfer tensor taken to be the aggregation prolongator corresponding to the partition with only one element is equivalent to classical TD(λ) for 1-MDPs applied to a k-MDP.*

Proof The update rule of the algorithm for 1-MDPs as applied to a k-MDP in the above notation is given by

$$\underline{z} := \lambda\gamma\underline{z} + e_s, \underline{v} := \underline{v} + \alpha\underline{z}\underline{d}\left(\underline{v}\right)$$

wherein $\underline{v} : S \to \Re$ denotes the current iterate for the approximate 1-MDP state-value function, and the temporal difference

$$\underline{d}\left(\underline{v}\right) := r - \left(\underline{v}_s - \gamma\underline{v}_{s'}\right),$$

with r signifying the most recently incurred reward, whereas, in the basis function view, the considered approximate algorithm for a k-MDP may be stated as

$$z := \gamma\lambda z + e_{(s,s)}, \theta := \theta + \alpha\Phi^T z d(\Phi\theta),$$

in which θ, Φ are as specified above, and the temporal difference

$$d(\Phi\theta) := r - \left((\Phi\theta)(\boldsymbol{s}, s) - \gamma(\Phi\theta)\left(\boldsymbol{s}^*, s^{'}\right)\right) = r - \left(e_{(\boldsymbol{s},s)} - \gamma e_{\left(\boldsymbol{s}^*,s'\right)}\right)\Phi\theta.$$

We shall show that, when applied simultaneously to the same k-MDP, the iterates satisfy

$$v = \theta$$

for all iterations. To this end, we point out that the following simplification holds in the here addressed special case $m = 1$:

$$\Phi\theta = \sum_{\alpha,\beta} \phi_{\alpha\beta}\theta_{\alpha\beta} = \sum_{\alpha} \phi_{\alpha 1}\theta_{\alpha 1} = \sum_{\alpha} \delta_{\alpha=s}\theta_{\alpha 1}$$

which implies, with an inconsequential abuse of notation in the last equality,

$$(\Phi\theta)\left(\boldsymbol{s}, s^{'}\right) = e^T_{(\boldsymbol{s},s)}\Phi\theta = \theta_{s1} =: \theta_s \forall(\boldsymbol{s}, s) \in \widetilde{S}.$$

Hence,

$$\begin{aligned} d(\Phi\theta) &= r - \left(e_{(\boldsymbol{s},s)} - \gamma e_{\left(\boldsymbol{s}^*,s'\right)}\right)\Phi\theta = r - \left(e_{(\boldsymbol{s},s)}\Phi\theta - \gamma e_{\left(\boldsymbol{s}^*,s'\right)}\Phi\theta\right) \\ &= r - \left(\theta_s - \gamma\theta_{s'}\right) \end{aligned}$$

Now it only remains to show that

$$\Phi^T z = \underline{z}$$

for all iterations, which we carry out by induction: both z and $\underline{z}$ are initialized as vectors of all zeros. Therefore, the sought-after statement holds for the first iteration. To conclude the induction, we argue as follows: let $z_-, \underline{z}_-$ denote the previous values of $z, \underline{z}$, i.e.,

$$z = \gamma\lambda z_- + e_{(\mathbf{s},s)}, \underline{z} = \gamma\lambda \underline{z}_- + e_s.$$

Since by induction assumption,

$$\Phi^T z_- = \underline{z}_-,$$

we obtain

$$\Phi^T z = \Phi^T\left(\gamma\lambda z_- + e_{(\boldsymbol{s},s)}\right) = \gamma\lambda\Phi^T z_- + \Phi^T e_{(\mathbf{s},s)} = \gamma\lambda\underline{z}_- + e_s = \underline{z},$$

which yields the desired result. □

In some sense, this result tells us that we have been doing a special case of the tensor factorization approach all along. Together with Proposition 10.1, moreover, it ensures that, at least mathematically, assuming an MDP of order k and approximating the state-value function according to the aggregation-based tensor model with an *arbitrary* partition is no worse than simply assuming order 1. Hence, the above introduced approach is consistent with what we have been doing all along.

10.5 Clustering Sequences of Products

At the end of the day, it all comes down to a judicious choice of the partition underlying the aggregation prolongator deployed in the factorization. At the first glance, choosing an error-minimizing partition with a prescribed number of classes seems most reasonable. As is well known among computer scientists, however, the arising optimization problem is equivalent to the clustering problem mentioned in the introductory part of the previous chapter, and thus NP-hard. Of course, we can use the k-means algorithm as proposed in Sect. 10.2, also in conjunction with an SVD or NMF to find a good initial guess. Nevertheless, we must keep in mind that, with regard to an application to recommendation engines, an explicit representation of the transition probabilities is typically neither available nor favorable. This situation leaves us with basically two options, the first of which consists in devising an adaptive online method and the second in an a priori choice of the partition based on additional knowledge about the situation described by the model rather than purely mathematical reasoning.

10.5.1 An Adaptive Approach

As it is subject of forthcoming research, the question of how to obtain a partition that minimizes the approximation error has as yet been left open. Nevertheless, we shall outline some ideas and outlooks that we consider promising.

A possible adaptive method may be based upon a *genetic algorithm* (GA). GA, introduced by Holland [Hol92], is a biologically inspired search heuristic framework for discrete optimization. Specifically, these schemes mimic a natural evolution process consisting of *selection*, *crossover*, and *mutation* on a *population* of elements of the search space. In the selection step, a subset of the population the elements of which maximize some *fitness function*, which, more often than not, coincides with the objective function, is picked. The crossover step consists in crossing elements of the selected subset so as to obtain a new generation of *individuals* each of which unites properties of different fittest individuals of the previous generation. In the last step of an iteration of a GA, each individual is exposed to mutation, which, mathematically, amounts to perturbing each individual slightly with respect to a suitably chosen metric. More details and references concerning genetic algorithms may be found in [Zim06].

To devise a GA for computing a partition of the set $\hat{S}$ such that the approximation error of the corresponding factorization be minimized, we consider a partition of cardinality m as a function $\hat{S} \to m$. The fitness function is taken to be some norm of the residual of an equivalent transformation of the Bellman equation evaluated at an approximate solution in terms of the corresponding aggregation. Of course, there is no way to evaluate such a residual directly in a model-free framework. It is, however, possible to estimate the residual in virtue of the temporal differences.

As of the crossover step, we propose the common approach of interleaving function values at random elements. More precisely, to cross two partitions f,g, we randomly pick a bit vector $b \in \{0,1\}^{\hat{S}}$ according to a distribution which favors vectors with approximately equal numbers of entries with value 0 and 1. The crossing h of f,g is then determined as

$$h(\mathbf{s}) := \begin{cases} f(\mathbf{s}), & b(\mathbf{s}) = 1, \\ g(\mathbf{s}), & b(\mathbf{s}) = 0. \end{cases}$$

The mutation may be performed with respect to the *Hamming metric*

$$d_H(f,g) := \left|\left\{\mathbf{s} \middle| f(\mathbf{s}) \neq g(\mathbf{s})\right\}\right|$$

which is well known in the information and coding community. Specifically, for each individual f, we pick a random element of the metric ball

$$\left\{g : \hat{S} \to \underline{m} \middle| d_H(f,g) \leq \varepsilon\right\}$$

for some small positive integer ε stipulated beforehand.

10.5.2 Switching Between Aggregation Bases

In the course of an adaptive computation of the partition underlying the aggregation prolongator, the computation of the state-value function needs to be restarted for each newly obtained partition. This gives rise to the question of whether the last iterate with respect to the previous aggregation may somehow be exploited to obtain a reasonable initial guess for the restarted iteration.

Given two partitions $\{G_\beta\}_{\beta \in \underline{m}}$, $\left\{\widetilde{G}_\beta\right\}_{\beta \in \underline{m}}$ of $\widetilde{S}$ with corresponding aggregation prolongators $U, \widetilde{U}$ and a core matrix (tensor) $\widetilde{\Theta} \in \Re^{S \times \underline{m}}$, what can be considered a faithful representation of $\widetilde{\Theta} U$ in terms of U? We resort to a least squares approach

$$\min_{\Theta} \left\| U\widetilde{\Theta}^T - \widetilde{U}\widetilde{\Theta}^T \right\|_F \tag{10.12}$$

The optimizer is given by

$$(\Theta^*)^T = U^+ \widetilde{U} \widetilde{\Theta}^T$$

Due to the special structure of U, it is possible to represent Θ^* in a simple closed form. To this end, recall that

$$U^T U = \left(|G_\beta| \delta_{\alpha\beta} \right)_{\alpha,\beta \in \underline{m}}$$

Moreover, we obtain

$$U^T U = \left(\left| G_\beta \cap \widetilde{G}_{\widetilde{\beta}} \right| \delta_{\alpha\beta} \right)_{\beta,\beta \in \underline{m}}$$

which gives rise to

$$\Theta_{s\beta} = \sum_{\widetilde{\beta} \in \underline{m}} \frac{\left| G_\beta \cap \widetilde{G}_{\widetilde{\beta}} \right|}{\left| G_\beta \right|} \Theta_{s\widetilde{\beta}}.$$

With some minor effort, this observation may be extended to the case of a weighted Frobenius norm in the optimization (10.12).

As regards the procedure outlined in the previous section, the above-derived least squares framework may be deployed to obtain initial iterates for the individuals in each new generation of the GA procedure. Here, it appears reasonable to take the factors $\widetilde{U}, \widetilde{\Theta}$ to those of the most recent iterate of one of the parents of the considered individual.

10.6 How It All Fits Together

We are now going to discuss how the approaches described so far in this book all fit together. Almost all of them deal with the problem of complexity: hierarchical methods to speed up convergence, factorization, and tensors as well as special empirical assumptions to reduce the complexity of the recommendation model.

We consider the most general task, the k-MDP of Sect. 10.1, and include multiple recommendations. As we stated in Sect. 4.2, multiple recommendations can be interpreted as single actions, and thus all considerations of Sect. 10.1 remain valid. However, the problem of multiple recommendations is their increased complexity.

We proceed similar to Sect. 10.2 and include the space of multiple recommendations

$$\overline{A} := \bigcup_{j=1}^{m} S^j,$$

defined in the same way as in Sect. 4.2 (except that here m denotes the number of recommendations instead of k which now is the number of preceding states). Then we arrive at our complete probability space:

$$P \in \Re^{\hat{s} \times s \times s \times \overline{A}}, P^{\overline{a}}_{\mathbf{s}ss'} := P^{(a_1, \ldots a_m)}_{(s_1, \ldots, s_l)ss'}, l < k, s_1, \ldots s_l, s, s', a_1, \ldots a_m \in S \qquad (10.13)$$

The problem is the high dimensionality of P. For $l = k-1$ the dimension is $k + m + 2$. The same applies to the reward space $R \in \Re^{\hat{s} \times s \times s \times \overline{A}}$; even with Assumption 4.2 we get $R \in \Re^{\hat{s} \times s \times s}$. The action-value function also belongs to $\Re^{\hat{s} \times s \times s}$ and the state-value function to $\Re^{\hat{s} \times s}$.

Let us focus on the most complex quantity, the transition probability (10.13). The best way would be an approximation through a tensor of dimension $k + m + 2$. The general tensor approach is described in Chap. 9. Unfortunately, this is an extremely difficult task because of the complexity of the decomposition algorithms and also the prediction quality of the model. Thus, we may look for a more specific approach. Therefore, we can use separate models for the approximation in the state and action dimensions. Thus, we seek an approximation in the state space and then add the approximation in the action space.

For the state space we can use tensor approximations as in Chap. 9 or the specific one presented in Sect. 10.2. If this is still too difficult, we ignore the previous states, i.e., we consider $k = 1$. In this case P is a matrix. So we can either apply the matrix factorization of Chap. 8 to P or calculate it directly.

To bring in the actions, we proceed as in Sect. 5.2 using the empirical Assumption 5.2. In case of multiple recommendations, we additionally need the framework of Sect. 4.2 which is based on Assumption 4.3. The combination of both for calculating transition probabilities has been demonstrated in Sect. 5.2.3. Similar considerations can be undertaken for the other quantities like transition rewards, action-value function, and state-value function.

Finally, the hierarchical methods presented in Chap. 6 allow to increase the convergence speed.

This way we have developed a complete tool set to handle reinforcement learning for recommendations. The different approaches can be combined in numerous ways. Of course, many of them still need to be refined, and also the question of their best combinations remains to be open. The answer, again, depends on the properties of the different approaches.

10.7 Summary

In this chapter, we have proposed a particular way to combine the factorization-based approach to recommendation with the control-theoretic one. We stress that this is only one specific manner in which the two paradigms may interact with each other, and there are certainly numerous fundamentally different possible connections.

In particular, we have pointed out a possible way to circumvent the complexity issues related to the more sophisticated yet more realistic model of a k-MDP. A case for this particular approach has been made by showing it to be consistent with the classical approach of assuming a 1-MDP. Yet, it is too early to assess the actual practical value of the approach, and there are still a lot of difficulties as regards implementation to overcome. Mathematically, we still lack deep results on approximation quality as well as numerical experience. With regard to recommendation engineering, we still need to confirm that assuming a k-MDP rather than a classical MDP actually results in a significant improvement and, if so, how to choose the order k, which trades off between accuracy and simplicity. Furthermore, we need to ensure that the above-presented factorization approach provides a satisfactory approximation and figure out a way to choose the underlying partition suitably, for example, by virtue of the above-outlined genetic algorithm framework, the practical effectiveness of which, too, still needs to be evaluated.

Further, we have described how the approaches presented in the previous chapters are related to our generalized k-MDP formulation.

With regard to future research, apart from the reasoning presented in this chapter, it might be interesting to state the two approaches, i.e., the factorization-based and the control-theoretic, within a unified framework. Such a framework may then enable to discover more possible ways to connect the two approaches.

Chapter 11
What Cannot Be Measured Cannot Be Controlled: Gauging Success with A/B Tests

Abstract The robust measurement of the efficiency of recommendation algorithms is an extremely important factor in the development of recommendation engines. We provide some useful methodical remarks on this topic in this chapter, even though it is not directly connected to the problem of adaptive learning. We further propose a straightforward algorithm to calculate confidence intervals for REs. At the end, we discuss Simpson's paradox which illustrates the importance of constant environment conditions for testing.

The use of A/B tests to assess the efficiency of recommendation algorithms is on the increase. Here a proportion of all episodes (generally web sessions) is randomly assigned to the recommendation algorithm group (referred to as the "recommendation group"), and the remaining episodes serve as a control group. Depending on the specific objectives, the control group may be empty (i.e., displaying no recommendations) or may be assigned to a different recommendation algorithm. In the group assignment of episodes, there is normally a fixed ratio between the number of episodes in each group, e.g., 50:50 or 90:10. We call this ratio the *episode quotient* q.

Along with the reward r, other relevant statistical characteristics can be measured in each group. In the case of web shops, these could be the number of clicks, shopping baskets, orders, purchased products, and in particular sales. Multiplying these figures by the episode quotient then gives the percentage efficiency of the recommendation algorithm as compared with the control group for all indicators.

The use of A/B tests to determine recommendation quality is widely accepted and meets generally recognized statistical and scientific standards. However, their correct implementation and evaluation require compliance with certain criteria, which we will look at more closely below.

A. Paprotny and M. Thess, *Realtime Data Mining: Self-Learning Techniques for Recommendation Engines*, Applied and Numerical Harmonic Analysis,
DOI 10.1007/978-3-319-01321-3_11, © Springer International Publishing Switzerland 2013

11.1 Same Environments in Both Groups

If the test is to be meaningful, it must not be influenced by any factors other than the recommendation algorithm itself. If a different recommendation algorithm is used in the control group, it is usually satisfied (but not always). It is usually more difficult if no recommendations are displayed in the control group, in other words if we are testing against an empty set.

In a web shop, the free space in the control group is sometimes used to display additional information or services, for example. This influences the outcome of the test, because we are no longer then measuring the use of the recommendation algorithm but rather testing the recommendation algorithm *against* the additional information or service, which is not the intention. If the recommendations are displayed below the product view, for example, but before the detailed product description, the recommendations may reduce the usability of the shop. Our test is then assessing the recommendation algorithm *against* detailed product information.

In a test against an empty control group, the recommendation display will of course inevitably change the appearance of the product detail view. But this change should be kept as minimal as possible. At the same time, however, the recommendations must be displayed prominently; otherwise, they could be ignored. For instance, recommendations could simply be displayed underneath the existing product detail view. Then the appearance of the page would scarcely change, but at the same time the recommendations could be overlooked. So there is clearly an element of conflict between the two requirements, but an effort should be made to find a reasonable compromise. One common solution is to display recommendations on the far right of the page, away from the product information. In this way, the page appearance is virtually unchanged, but the recommendations are well positioned.

The struggle to achieve **maximum constancy of environmental conditions** is one of the key factors differentiating science from scholasticism. It is frequently underestimated (and in A/B tests often complicated), for which reason we would like to spend a little more time on it. The following passage comes from the Soviet winner of the Nobel Prize for chemistry, Nikolay Semyonov, who reflected on the difficulty of biological evaluations [Sem81]:

> It is sometimes said that in biology, because of the complexity, state and individuality of an organism, experimental conditions cannot be set with the same degree of precision as in physics or chemistry, and that as a consequence the results obtained may vary.
>
> Such differences do of course arise in experiments on living creatures, and in particular on human beings. For example, a drug can help some people and harm others suffering from the same illness.
>
> However, the statistical result over a large number of people will show the same distribution.
>
> The causes of this distribution help us to identify the precise physiological characteristics of a certain type of person which determine whether a drug is beneficial or harmful.
>
> The claim that consistent experimental results cannot be obtained objectively in biology is wrong. Otherwise medicine or agronomy would be impossible.
>
> With a large enough data set, the differences between different organisms of the same species can be seen in the statistical distribution, the mean of which is the same

(under constant experimental conditions of course). It is true that setting experimental conditions is more difficult in biology than it is in physics or chemistry.

This means that in biological experiments, more attention has to be paid to this problem, not less.

It would be wrong to think that we do not also encounter great difficulties in physics and particularly in chemistry, but we devote the time and effort to overcome them. The work involved in standardizing experimental conditions in itself delivers important scientific results.

That is why the need to standardize experimental conditions is a difficult but necessary task in science.

These comments apply in their entirety to the complex field of retail, in which REs are mostly used.

11.2 No Loss of Performance Through Recommendations

This is a very important special case relating to the previous point of constancy of environment. It concerns applications with a high recommendation rate, mostly web shops but also sometimes call centers or supermarkets.

In many cases, the complete content in the recommendation group is not delivered until the product recommendations have been fully calculated and integrated. This is particularly true in cases where the recommendation engine is integrated on the server side. In such cases, the content delivery in the recommendation group can tend to be slower than in the control group, and this has a negative influence on user behavior. Caution is advised when manually assessing the time delay. In web shops in particular, it is difficult to estimate accurately. It should always be measured over a number of sessions.

For that reason, an asynchronous delivery of recommendations is generally preferable. Client-side integration via IFrame or Ajax is usually available for web shops. The recommendations are then completely separate from the web shop and are loaded when the product detail view is opened. If asynchronous integration is not an option, it is essential to measure the response times for both groups automatically over large numbers of episodes, and the test should only proceed if the delays in the recommendation group are negligible or irrelevant.

11.3 Assessing the Statistical Stability of the Results

The results of the A/B test can vary widely. It is not unusual to see a 10 % increase in sales 1 day, followed by an 8 % fall the next. So the question is: When are the results reliable? Clearly, the higher the statistical mass and, in particular, the longer the test (under comparable environmental conditions), the more credible the results. But when do we reach the point at which we can say: “Now I am convinced that this increased sales figure is correct”?

The answer lies in confidence intervals, which take the form $[x_u, x_0]$ for the desired indicator and which apply for a specified percentage. For example, a 95 % confidence interval [−1.5, 3.5 %] for the sales increase means that for the period in question, the true value can be expected with 95 % probability in the range between −1.5 % and 3.5 %. This is since the measured value lies with a probability of 95 % in the expected range of the sample. Notice that the simpler formulation that the true value lies with 95 % probability in the confidence interval is mathematically not correct. However, colloquially, it describes the meaning of the confidence interval well.

So rather than simply stating a value of 1.0 % increased sales, its 95 % confidence interval [−1.5, 3.5 %] is given too. As the statistical set increases, the confidence interval narrows and closes in on the indicator.

Determining the confidence interval for the increased sales due to a recommendation engine is by no means straightforward. The method was developed by the mathematicians Holm Sieber and Toni Volkmer in [SV10], and we will present it briefly.

W.l.o.g. we suppose the session quotient $q = 1$. Furthermore, let $\overline{X}_A$ be the average revenue per session of group A and $\overline{X}_B$ the revenue of group B. Then the increase in revenue of group B is calculated as

$$d = \frac{\overline{X}_B}{\overline{X}_A} - 1. \tag{11.1}$$

This value typically has a very high variation, so it is insufficient to state just the mean value in order to make reliable conclusions. Thus, we will present a way to calculate the confidence interval for d.

The revenue increase d is a random variable. This follows from the random character of the revenue of one session.

Let X_A be the revenue of a session in group A, and X_B the revenue of a session group B, and the numbers of the corresponding sessions are n_A and n_B. In sessions without order, the revenue is simply 0. We can assume that the revenue satisfies an unknown but stationary distribution. The expected value and variance are unknown but can be estimated from a sample.

By applying the central limit theorem, we can first describe the distributions of $\overline{X}_A$ and $\overline{X}_B$. Both are approximately normally distributed:

$$\overline{X}_A \sim N\left(EX_A, \frac{D^2 X_A}{n_A}\right),$$

$$\overline{X}_B \sim N\left(EX_B, \frac{D^2 X_B}{n_B}\right).$$

The target quantity (11.1) d thus is the quotient of two normal distributions. As for the treatment thereof, a paper of Robert Geary [Gea30] turns out to be helpful. It considers the expression

$$z = \frac{b + y}{a + x}, \tag{11.2}$$

where x and y are normally distributed with expected values 0, standard deviations α, β, and correlation r. Under these assumptions, the expression

$$t = \frac{az - b}{\sqrt{\alpha^2 z^2 - 2r\alpha\beta z + \beta^2}} \tag{11.3}$$

is standard normally distributed $N(0,1)$ if $a + x$ is in general nonnegative. This assumption is easily satisfied in our case because we consider revenues.

We may apply this insight to our task at hand. From (11.1), it readily follows that

$$d + 1 = z = \frac{\overline{X}_B}{\overline{X}_A}. \tag{11.4}$$

With

$$a = EX_A,$$
$$b = EX_B,$$
$$\alpha^2 = \frac{D^2 X_A}{n_A},$$
$$\beta^2 = \frac{D^2 X_B}{n_B},$$
$$r = 0,$$

we obtain (11.2).

By virtue of (11.4), we may derive the desired confidence interval for d. Due to (11.3), t is normally distributed. Let U_P be the p-quantile of the standard normal distribution for probability p. Then

$$P\left(U_P \leq t \leq U_{(1-p)}\right) = 1 - 2p.$$

The problem is symmetric and therefore it holds that

$$t^2 \leq U_p^2.$$

Taking $u^2 = U_p^2$, we got the following approach for the solution of the inequality:

$$
\begin{aligned}
u^2 &= t^2 \\
u^2 &= \frac{a^2z^2 - 2abz + b^2}{\alpha^2z^2 + \beta^2} \\
0 &= \frac{a^2z^2 - 2abz + b^2 - u^2\alpha^2z^2 - u^2\beta^2}{\alpha^2z^2 + \beta^2} \\
0 &= z^2(a^2 - u^2\alpha^2) - z(2ab) + \left(b^2 - u^2\beta^2\right) \\
0 &= z^2 - z\frac{2ab}{a^2 - u^2\alpha^2} + \frac{b^2 - u^2\beta^2}{a^2 - u^2\alpha^2}.
\end{aligned}
$$

The solution formula for quadratic equations

$$z_{u,o} = -\frac{v}{2} \pm \sqrt{\frac{v^2}{4} - w} \tag{11.5}$$

with

$$v = -\frac{2ab}{a^2 - u^2\alpha^2}, w = -\frac{b^2 - u^2\beta^2}{a^2 - u^2\alpha^2}, \tag{11.6}$$

leads to the desired confidence interval:

$$
\begin{aligned}
z_u &\le d + 1 \le z_o \\
z_u - 1 &\le d \le z_o - 1.
\end{aligned} \tag{11.7}
$$

We now turn to the implementation. First, we need to determine the values a, b, α, β. Here EX_A is "value per visit" of the control group and EX_B the similar value for the recommendation group. For the calculation of the confidence interval for the average order revenue, similarly "avg. order value" has to be used. A confidence interval "CRO" can also be determined.

The variance D^2X_A of the control group and D^2X_B of the recommendation group can be calculated via the quadratic sums of the order revenues:

$$\left.\begin{aligned} D^2X_A &= EX_A^2 - (EX_A)^2 \\ D^2X_B &= EX_B^2 - (EX_B)^2 \end{aligned}\right\}.$$

The values n_A and n_B depend on the target quantity "visits" or "orders."

The quantile of the normal distribution U_p, and hence u, depends on the desired confidence level and is a constant:

- 90 % confidence interval: $U_p = U_{0,95} = 1.6449$
- 95 % confidence interval: $U_p = U_{0,975} = 1.9600$
- 99 % confidence interval: $U_p = U_{0,995} = 2.5758$

Therewith, using (11.6), we can compute p, q, and by (11.5) and (11.7), we get the desired interval.

As skeptical as many companies have been about A/B (and multivariate) testing at the very beginning, the more optimistic many of them increasingly became. As a result, skepticism sometimes has been replaced by an exaggerated belief in the prospects of such testing.

Many arguments seem to support the virtually unlimited power of testing. If I can systematically test where to place best which banner, which button, which navigation element, and which picture and in what color and size, didn't the magical testing tool automatically lead me to the optimal shop? Do we need shop managers, editors, and designers anymore?

Of course, it is not so easy. The reason is complexity, because similar ideas have already been developed some centuries ago. So Jonathan Swift ironically wrote in *Gulliver's Travels* about the flying island of Laputa, a kingdom devoted to the arts of music and mathematics:

> He then led me to the frame, about the sides, whereof all his pupils stood in ranks. It was twenty feet square, placed in the middle of the room. The superfices was composed of several bits of wood, about the bigness of a die, but some larger than others. They were all linked together by slender wires. These bits of wood were covered, on every square, with paper pasted on them; and on these papers were written all the words of their language, in their several moods, tenses, and declensions; but without any order. The professor then desired me "to observe; for he was going to set his engine at work." The pupils, at his command, took each of them hold of an iron handle, whereof there were forty fixed round the edges of the frame; and giving them a sudden turn, the whole disposition of the words was entirely changed. He then commanded six-and-thirty of the lads, to read the several lines softly, as they appeared upon the frame; and where they found three or four words together that might make part of a sentence, they dictated to the four remaining boys, who were scribes. This work was repeated three or four times, and at every turn, the engine was so contrived, that the words shifted into new places, as the square bits of wood moved upside down.
>
> Six hours a day the young students were employed in this labour; and the professor showed me several volumes in large folio, already collected, of broken sentences, which he intended to piece together, and out of those rich materials, to give the world a complete body of all arts and sciences...

In fact, calculation of confidence intervals as described in this section quickly reveals that in general large data volumes are required to obtain reliable results. Moreover, during the tests, conditions like the assortment or product prices in the shop or purchasing behavior of the customers can change. So A/B testing is an important instrument to compare and verify algorithms, but it shall be used on a high level and cannot replace systematic development of recommendations and content.

11.4 Observing Simpson's Paradox

Let us conclude by looking at a curious effect known as Simpson's paradox. This phenomenon is well known in statistics and was first investigated in 1951 by Edward Hugh Simpson. In the area of A/B tests for recommendation engines, it is manifested by a variation in qualitative results when switching to cumulative indicators.

Table 11.1 Simpson's paradox based on the example of a 2-day A/B test

	Recommendation group		Control group		
Period	Sessions	Sales volume	Sessions	Sales volume	Sales increase
1 Day	10	500	20	2,000	−50 %
2 Day	20	3,900	10	2,000	−2.5 %
Total	30	4,400	30	4,000	+10 %

Example 11.1 We can illustrate Simpson's paradox using a simple example of a 2-day A/B test in a web shop (Table 11.1):

Although in percentage terms the results for the recommendation group are worse than those for the control group on both days, the first group appears to emerge at the end as the clear winner, with +10 %. The reason for this is that the session quotient q differed on each day: on day 1, it was $q = 10{:}20 = 0.5$, but on the second day it was $q = 20{:}10 = 2.0$. ■

The superficial reason for the paradox is the fact that the individual results are weighted differently in the overall result. In essence, the paradox usually indicates that certain influencing factors have not been taken into consideration. In our case, it is due to the different session quotients, and the solution is to keep them constant. This underlines once again the need to maintain maximum constancy of all environmental conditions, as we mentioned in point 1.

11.5 Summary

The robust measurement of the efficiency of recommendation algorithms is an extremely important factor in the development of REs. We provided some methodical remarks on this topic in this chapter, even though it is not directly connected to the problem of adaptive learning. We have further proposed a straightforward algorithm to calculate confidence intervals for REs.

Chapter 12
Building a Recommendation Engine: The XELOPES Library

Abstract In this chapter we provide some ideas of implementing the adaptive algorithms described in this book based on the prudsys XELOPES library for BI. We start with the abstract CWM standard and then consider its application to data mining. Next we move to realtime data mining where the central idea is the introduction of agents. The agent framework is further specified for reinforcement learning, and based on RL we next propose a framework for adaptive recommendation engines. At the end, we briefly discuss the application of XELOPES for real recommendation engines.

The prudsys XELOPES is a business intelligence (BI) library with focus on realtime analytics. Especially, it contains all main algorithms of Chaps. 3, 4, 5, 6, 7, 8, and 9. XELOPES is a commercial library, but there exists an open source version of restricted functionality that can be downloaded from http://www.prudsys.com/xelopes. Besides basic algorithms, the open source version provides the complete infrastructure for realtime analytics of XELOPES. No matter if the reader wants to use XELOPES or not, this chapter may provide some useful information about realtime analytics implementations.

In this chapter, we give a short introduction to XELOPES. In Sect. 12.1, we first describe the infrastructure of the library including classical data mining, i.e., offline learning. Section 12.2 is devoted to realtime analytical approach of XELOPES, the online learning. At this, we first present the agent framework of the library. Based on this framework, the packages of reinforcement learning and RL-based recommendations are presented. In Sect. 12.3, we finally present the prudsys RDE as a comprehensive example of how to use XELOPES to build a recommendation engine.

A. Paprotny and M. Thess, *Realtime Data Mining: Self-Learning Techniques for Recommendation Engines*, Applied and Numerical Harmonic Analysis,
DOI 10.1007/978-3-319-01321-3_12, © Springer International Publishing Switzerland 2013

12.1 The XELOPES Library

12.1.1 The Main Design Principles

12.1.1.1 Basis Transformations

XELOPES uses a radical concept to solve analysis tasks: basis transformations! The main task of XELOPES is to provide appropriate bases and efficient basis transformations in order to solve data analysis problems efficiently.

The term *basis* is used in a wider sense than of a mathematical basis only. Basis transformations are applied on the following three levels:

1. *Basis transformations of data*: For analysis tasks, data is often required to be presented in different formats, i.e., in different bases. There exist different bases of training and application data, transactional and non-transactional formats, dense and sparse formats, etc.
2. *Basis transformations inside analysis algorithms*: Most analysis algorithms can be described and implemented in terms of basis transformation methods; we discussed this in Sect. 6.1.
3. *Basis transformations into CWM basis:* The CWM standard (see below) which is used as fundament of XELOPES is a basis itself! Thus, the data that usually exists in many proprietary data storage formats can be represented in the unified CWM format.

Basis transformations consist of two parts:

1. Find an appropriate basis for a problem.
2. Find an *efficient* basis transformation into the new basis.

Both problems are usually not easy to solve. However, the most important step is to understand that such a basis transformation is actually necessary. This requires separating the basis from the data (more specifically, the coordinates of the data instances). Most systems in data mining simply forgo such separation. In contrast, XELOPES clearly distinguishes between basis and the data itself. This separation is done on all three levels mentioned. It is absolutely fundamental for understanding XELOPES.

12.1.1.2 Modular Concept: CWM

CWM (Common Warehouse Metamodel) is the fundament of XELOPES. The primary objective of the CWM is to define a metamodel of a generic data warehouse architecture. In combination with MOF-related standards like XMI and JMI, CWM allows to exchange metadata between BI applications of different vendors and types. In particular, metadata between data warehouses can be exchanged.

Fig. 12.1 Cubism as example of decomposition of structures in small building blocks (Pablo Picasso, Les Demoiselles d'Avignon, 1907)

One may ask: Why are we interested in a data warehouse-related standard like CWM? Isn't the realtime analytical approach promoted in this book the complete antipode to data warehousing? This is definitely true. However, CWM is useful because it can describe highly complex metadata of almost all types of data storage. We have emphasized before that basis transformations are extremely important, not just in modern mathematics but – in a wider sense – in many other areas such as computer science. Similar to realtime analytics, which is becoming a key concept for data analysis, we believe that basis transformations will become central for data processing, including storage, comparison, and exchange of data. For this reason, it was selected for XELOPES. To put it simply, we believe that comprehensive metadata handling is of central importance for business intelligence. Therefore, we will introduce CWM here although it is not directly related to realtime analytics.

CWM is one of the most abstract IT standards at all. Like in cubism Braque and Picasso tried to decompose all images into a small set of geometric forms (most notably cubes), CWM breaks down the IT structure of whole enterprises into smallest atoms of a minimum number (UML-like classes) (Fig. 12.1).

We will give a short introduction into the fascinating world of CWM only. For a comprehensive description of data warehouse principles, we refer to [In96, Kim96]. Our introduction to CWM is based on [PCTM02, PCTM03] which explains the standard in all details.

As we already mentioned, CWM is a quite complex standard and requires knowledge from other OMG standards like MOF, XMI, UML, and MDA. Especially, CWM is described in terms of the MOF (Meta-Object Facility) meta-metamodel. MOF, in turn, leverages concepts from UML (Unified Modeling Language) for the description of metamodels. Therefore, CWM uses UML for description and modeling and is platform independent.

Layer	Packages
Management	Warehouse Process, Warehouse Operation
Analysis	Transformation, OLAP, Data Mining, Business Nomenclature, Information Visualization
Resource	Relational, Record, Multidimensional, XML
Foundation	Business Information, Data Types, Software Deployment, Key Indexes, Expres sions, Type Mapping
Object	Core, Behavioral, Relationships, Instance

Fig. 12.2 CWM packages

Overview

CWM provides a framework for representing metadata about data sources, data targets, transformations, analysis, and the process and operations that create and manage warehouse data and provide lineage information about its use. The primary objective of the CWM is to define a metamodel of a generic data warehouse architecture. Thus, the CWM defines formal rules for modeling instances of data warehouses.

The CWM is split up into a set of packages. This should aid comprehension of the metamodel, by splitting it up into smaller units, and also allow users and implementers to ignore packages that are not relevant for their needs.

The CWM has a layered structure (Fig. 12.2):

- **Foundation layer**: The foundation consists of the UML-based object model and the CWM Foundation, which supports additional concepts and structures that are shared by other packages. Additionally, the *Software Deployment* package supports the deployment information for the data sources and targets in the next layer.
- **Resource layer**: The *Relational, Record, Multidimensional,* and *XML* packages support the definition of various types of data sources and data targets. Often, an Object-Oriented package is included into the Resource layer. It refers to the object model which is reused to model object-oriented data resources.
- **Analysis layer**: The *Transformation, OLAP, Data Mining, Information Visualization*, and *Business Nomenclature* packages define the transformations and analytical processing that take place on these data sources.
- **Warehouse Management layer**: Finally, the *Warehouse Process* package supports scheduling information, and the *Warehouse Operation* package is used to record operational details such as the results of transformation runs.

The CWM is designed to maximize the reuse of object model (a subset of UML) and the **sharing of common modeling constructs where possible**. The most prominent example is that CWM reuses object model for representing object-oriented data resources as noticed above. In addition, where applicable, key elements of the

metamodels for other types of data resources all subclass from the same model elements in object model.

CWM uses UML in three different critical roles:

- **UML is used as the MOF-equivalent meta-metamodel**. UML, or the part that corresponds to the MOF model, UML notation, and OCL (Object Constraint Language) are used as the modeling language, graphical notation, and constraint language, respectively, for defining and representing CWM.
- **UML is used as the foundation metamodel**. UML, specifically a subset as represented by the object model packages, is used as the foundation of CWM from which other metamodels inherit classes and associations.
- **UML is used as the object-oriented metamodel**. UML, specifically the object model package, is relied on for representing object-oriented data sources.

In order to illustrate this, we give a brief introduction to the object model which is fundamental for CWM and hence also for XELOPES.

Object Model

The object model layer contains packages that define fundamental metamodel concepts, relationships, and constraints required by all other CWM packages. The Object Model is essentially a subset of UML. Most of its classes and associations directly correspond to UML classes and associations.

The Object Model consists of the following packages:

- *Core* package: Contains classes and associations that form the core of the CWM Object Model, used by all other CWM packages including other Object Model packages.
- *Behavioral* package: Contains classes and associations that describe the behavior of CWM objects and provide a foundation for describing the invocations of defined behaviors.
- *Relationships* package: Contains classes and associations that describe the relationships between CWM objects.
- *Instance* package: Contains classes and associations that represent instances of CWM classifiers.

We focus on the *Core* package which is most important. The class diagram of *Core* is shown in Fig. 12.3.

Core does not depend on other packages. The initial class of *Core* is *Element*. In CWM, every class in every package is a subclass of the *Element* class. *Element* has no attributes and no methods. Its only function is to represent the root of the tree of all CWM classes. The class *ModelElement* extends *Element*, and, with the exception of a few support classes, all CWM classes are also subclasses of *ModelElement*. *ModelElement* provides some basic attributes like *name* for all of its subclasses.

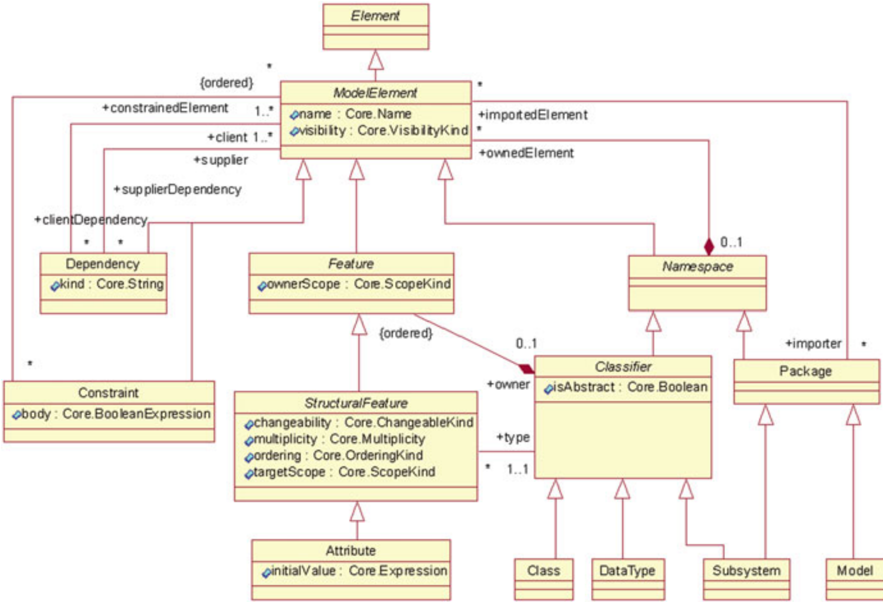


Fig. 12.3 Class diagram of CWM Core package

We cite [PCTM03]:

At its heart, the *Core* package provides for the description of things that have structure. Structured things include familiar computer system objects like relational database tables, records in a file, and members of Online Analytical Processing (OLAP) cube dimensions. In UML terms, the individual items of a thing's structure are called *features* and are represented by the *StructuralFeature* class. For example, the features of a relational table are the columns in the table; for a record, they are an ordered list of the record's fields. CWM allows for nonstructural features as well; they are described by the *Behavioral* package. The *Attribute* class represents structural features that can have an initial value.

Features are owned by *Classifiers* through a composite association. A classifier is a thing that has a structure; for example, both records and relational tables are types of classifiers. The notion of classifier is very similar to the idea of type used in modern programming languages. *Integer* and *character* are simple, frequently encountered programming language types; they are classifiers in CWM, but they have no features. *Address*, in contrast, is a compound type (classifier) whose features might consist of *street, city, state,* and *zip code*. In the same way, a relational table is a classifier whose features are its columns, and a record is a classifier whose features are its fields. Note that *StructuralFeatures* are owned by one classifier and are related on another classifier. The former is the *StructuralFaeture*'s owner, and the latter is its type. A *StructuralFeature* cannot have the same classifier as both its owner and its type.

The class named *Class* represents classifiers that can have multiple instances. So, Tables are really instances of *Class* because they can contain multiple data rows. In contrast, the *DataType* class represents classifiers that have only one instance; *integer* and *character* are instances of *DataType*.

Although *Namespaces* have no attributes, they are critically important because they ensure that individual objects can be uniquely identified by their names. Consequently,

Fig. 12.4 Classifier equivalence (from [PCTM03])

> *nearly every model element in a CWM description will be owned by some namespace*. Normally, the only model elements that are not owned by namespaces are those representing top-level namespaces (that is, namespaces not owned by other namespaces). The composite association between *Namespace* and *ModelElement* allows namespaces to own model elements, and hence, other namespaces. This association is one of the primary structuring mechanisms within the CWM. This association enables model elements to be organized in hierarchical, or tree-like, arrangements in which the parent namespace is said to contain, or own, its child *ModelElement* regardless of their ultimate type. Element ownership is reused extensively throughout the CWM to indicate ownership relationships between classes of every level.
>
> Because of the package structure of CWM, model elements must be able to reference objects in other packages. This is achieved by the *Package* subclass of *Namespace*. Packages, because they are namespaces, allow model elements of arbitrary types to be collected into hierarchies. However, because a model element can be owned by, at most, one namespace, we cannot use this mechanism to pull in model elements owned by different namespaces. Instead, the *Package* class provides the notion of *importing* model elements from other packages.

Resource Packages

Next, we give a short example of how elements of the Object Model (more specific, of the *Core* package) can be efficiently reused in metamodels of the Resource packages. Figure 12.4 shows the equivalence.

The first column of the diagram contains the names of the resource packages. The second column contains the names of the classes of the resource packages corresponding to *Core*'s class *Package*. Similarly, the third column is formed by the class names corresponding to *Core's* class *Classifier*. The fourth column contains the class names corresponding to *Core's* class *Feature*.

For instance, the *Relational* resource package, describing relational databases, uses schemas (package) to structure the tables (classifiers). Every table (classifier), in turn, contains a number of columns (features).

The diagram depicts that *Package, Schema*, and *RecordFile* are equal. In fact, *Schema* and *RecordFile* extend *Core*'s *Package*. The diagram also depicts that *Classifier, Table*, *RecordDef, Dimension,* and *ElementFile* are equal. Again, *Table*, *RecordDef, Dimension,* and *ElementFile* all extend *Core*'s *Classifier*. Finally, the diagram depicts that *Feature, Column, Field, DimensionedObject,* and (XML) *Attribute* are equal. In fact, *Column, Field, DimensionedObject,* and (XML) *Attribute* all extend *Core*'s *Feature*.

We have included this section in order to show how effective different application types can be unified and modeled by elements of CWM. This approach is widely used in XELOPES.

CWM and XELOPES

XELOPES is built on and compatible to the CWM standard 1.0. The *Data Mining* package of CWM is the central class extended by XELOPES. Moreover, meanwhile XELOPES actively uses almost all CWM packages except for the *Warehouse Management* layer.

In the next section, we focus on the functional description of XELOPES, and we will systematically develop the XELOPES foundation. At this, we will briefly explain which packages of CWM are used and which CWM classes are extended. This is important because the structure of XELOPES is highly influenced by the CWM.

We finally mention that the ordinary user of XELOPES does not need to know much about CWM. In contrast, for users who extend XELOPES, it is helpful.

12.1.1.3 Business Intelligence Standards

XELOPES supports different standards from Business Intelligence. Beyond CWM, these are JDM (Java Data Mining) [JDM] and JOLAP (Java OLAP) [JOLAP] and, most importantly, PMML (Predictive Model Markup Language). PMML is a standard for vendor-independent XML exchange of data mining models [PMML]. PMML is supported by the core of XELOPES, and all of its models can be exported into/imported from PMML. This applies not only to data mining models but also to all agents, including that of recommendation engines. For this purpose, the PMML standard was extended by prudsys for agents. A PMML file contains all information required to apply an analysis model/agent like metadata of the input, transformations, and the model itself. This makes this format very compact and easy to use. We will not go into further detail here, since it is quite technical. We just point out the PMML is used for serialization of all models and agents of XELOPES.

Layer	Packages
Management	Warehouse Process; Warehouse Operation
Analysis	Transformation; OLAP; Data Mining; Business Nomenclature; Information Visualization
Resource	Relational; Record; Multidimensional; XML
Foundation	Business Information; Data Types; Software Deployment; Key Indexes; Expres sions; Type Mapping
Object	Core; Behavioral; Relationships; Instance

Fig. 12.5 CWM packages used in MiningDataSpecification

12.1.2 The Building Blocks of the Library

12.1.2.1 The Basis: MiningDataSpecification and MiningAttributes

The class *MiningDataSpecification* represents the basis of a space. Thus, this is the most important class of XELOPES. Often, this class is simply referred to as metadata of the mining what is the equivalent of calling it the basis.

The basis vectors of *MiningDataSpecification* are the *MiningAttributes* representing the attributes. Therefore, *MiningDataSpecification* extends the CWM class *Class*, and *MiningAttribute* extends the CWM class *Attribute* (Fig. 12.3).

There are two basic types of mining attributes extending the abstract class *MiningAttribute*: *NumericAttribute* for numeric attributes like age, income, and time and *CategoricalAttribute* for categorical attributes like names, IDs, and types. The elements of a numeric attribute are real numbers. The elements of a categorical attribute are the categories which are represented by the *Category* class.

Example 12.1

```
// Create category 'knife':
Category catKnife = new Category("knife");
```

■

The categories of a categorical attribute are stored in an array of *Categorical Attribute*. Unlike as for the straightforward *NumericAttribute*, the mathematical nature of the *CategoricalAttribute* is rather ambivalent: it can be interpreted as one or a set of multiple numeric attributes. In the last case (e.g., binning), *CategoricalAttribute* represents a basis itself, with the *Category*-s as basis vectors. Thus, the set of categories is also called the *basis* or the *metadata* of the categorical attribute. Each category of a categorical attribute can be mapped to a unique real number (usually an integer) which is called the key of this category. This establishes a mutually unique mapping between the categories and

a set of real numbers and allows reducing the handling of categories to that of real values.

Example 12.2

```
// Create categorical attribute 'cutlery':
CategoricalAttribute cutlery = new CategoricalAttribute
  ("cutlery");

// Add categories:
Category catFork = new Category("fork"); // new fork
  category
cutlery.addCategory( catKnife );   // from previous example
cutlery.addCategory( catFork );
cutlery.addCategory( new Category("spoon") );

// Show key-value relationship:
double keyKnife    = cutlery.getKey( catKnife );
Category catKnife2 = cutlery.getCategory( keyKnife );

// catKnife == catKnife2
```
■

XELOPES actively supports three storage types of categorical attributes:

- *Static category set* (default): All categories are known a priori. Examples are sex (female, male) or colors (red, green, blue).
- *Dynamic category set*: During the data processing, new categories may appear and are added to the basis (option *unboundedCategories*). Examples are item or category names.
- *Dynamic category set with one category*: During the data processing, only the current category is stored (option *unstoredCategories*, implies *unboundedCategories*). Examples are transaction IDs and customer names.

Of course, the proper use of the storage types for the given examples can also differ. We emphasize that the support of the *unboundedCategories* and *unstoredCategories* types is really complicated – especially in the field of basis transformations – but very valuable since it allows to handle large and live data sources.

Categorical attributes with a defined order of categories are modeled by the class *OrdinalAttribute* which extends *CategoricalAttribute*.

Further, the categories of a categorical attribute can be organized into a hierarchy (also referred to as taxonomy). This is, e.g., required for many basket analysis algorithms or, in an extended form, for multilevel methods as in Chap. 6. Hierarchies of categories are modeled by the class *CategoryHierarchy* and can be assigned to a categorical attribute. *CategoryHierarchy* uses the method *addRelationship* to add a new edge to the hierarchy graph, and many methods allow running calculations on the graph.

Example 12.3

```
// Create category hierarchy:
CategoryHierarchy cah = new CategoryHierarchy();

// Parent category for sharp cutlery:
Category catSharp = new Category("sharp");

// Relations:
cah.addRelationship(catSharp, catKnife); // knife is sharp
cah.addRelationship(catSharp, catFork);  // fork is sharp

// Assign hierarchy to cutlery:
cutlery.setTaxonomy(cah);
```
■

We conclude with an example of *MiningDataSpecification.*

Example 12.4

```
// Create object of metadata 'meal':
MiningDataSpecification meal = new MiningDataSpecification
  ("meal");

// Create numeric attribute 'calories' and add to metadata:
NumericAttribute calories = new NumericAttribute
  ("calories");
meal.addMiningAttribute( calories );

// Create numeric attribute 'numberOfGuests' and add to
  metadata:
NumericAttribute numberOfGuests = new NumericAttribute();
numberOfGuests.setName( "number of guests" );
meal.addMiningAttribute( numberOfGuests );

// Add previous categorical attribute 'cutlery' to metadata:
meal.addMiningAttribute( cutlery );
```
■

12.1.2.2 The Coordinates: MiningVector

After we have modeled the basis of the attribute space by *MiningDataSpecification*, we will now model the coordinates of a vector. This is done through the class *MiningVector*.

MiningVector contains a reference *metaData* (of the class *MiningDataSpecification*) to its basis and an array of real values which stores the coordinates of the vector. *MiningVector* extends the CWM class *Object* of the CWM resource package *Instance* since it represents an instance of the data described by *MiningDataSpecification* (Fig. 12.6).

Management	Warehouse Process	Warehouse Operation			
Analysis	Transformation	OLAP	Data Mining	Business Nomenclature	Information Visualization
Resource	Relational	Record	Multidimensional	XML	
Foundation	Business Information	Data Types	Software Deployment	Key Indexes	Expres sions	Type Mapping
Object	Core	Behavioral	Relationships	Instance	

Fig. 12.6 CWM packages used in MiningVector

Example 12.5 Example of a mining vector for the *meal* basis of the previous section:

```
// Create and fill value vector:
double[] mealValues = new double[3];
mealValues[0] = 33000; // calory number
mealValues[1] = 5;    // 5 guests
mealValues[2] = cutlery.getKey( new Category("spoon") );
    // spoon

// Create mining vector object with values:
MiningVector mealVector = new MiningVector( mealValues );

// Add 'meal' metadata to mining vector:
mealVector.setMetaData( meal );

// Show (double) values of mining vector:
for (int i = 0; i < mealVector.getValues().length; i++)
  System.out.println("value["+i+"] = " + mealVector.
      getValue(i));
```
■

For sparse vectors, i.e., vectors which mainly contain zero coordinate values, the class *MiningSparseVector* could be used which extends *MiningVector*. It stores sparse vectors more efficiently by means of an additional array of indexes of the nonzero coordinate values. For binary sparse vectors, i.e., sparse vectors where the nonzero values are always one, the class *MiningBinarySparseVector* should be utilized which in turn extends *MiningSparseVector*.

12.1.2.3 The Data Matrix: MiningInputStream

So far we have defined the class *MiningVector* that models a data vector. In order to model a whole data matrix, we use the abstract *MiningInputStream* class. *MiningInputStream* is a virtual collection of mining vectors. Like each of its mining vectors, *MiningInputStream* contains a reference *metaData* to the basis of the attribute space.

Notice that *MiningInputStream* implements the interface *MiningVectorSet* which is the most general container of mining vectors. *MiningVectorSet* contains only the method *getMetaData* to access the basis of the attribute space. *MiningInputStream* as container of countable sets of mining vectors is the most important implementation of *MiningVectorSet*. However, some applications like reinforcement learning (Sect. 12.2.2) require access to uncountable sets of mining vectors such as all points of a domain or all unit vectors starting from one point. In this case, the representation of the mining vector set requires a more abstract level such as by functions of the boundary or geometric objects. We will return to this topic in Sect. 12.2.2; at this point, we mention that *MiningInputStream* is sufficient for data mining applications.

MiningInputStream contains a graded spectrum of data access methods depending on its implementation. In the simplest case, the data matrix can be traversed only once using a cursor-based approach using the method *next*. If the *reset* method is supported, the cursor can be set at the initial position. This access type is often supported by files and databases. In a more comfortable case, the cursor can be moved arbitrary using the *move* method (e.g., for databases supporting JDBC 2.0). Even more comfortable is the direct access to the data array of the data matrix, if the matrix fits into memory (e.g., class *MiningStoredData*).

The *read* method returns the mining vector at the current cursor position. Each full implementation of *MiningInputStream* must at least support the *next* and *read* methods. In addition, *MiningInputStream* may implement the interface *MiningOutputStream* to write data to the data source. Each mining input stream is reflective: the method *getSupportedStream* returns all data access (and update) methods supported by the current implementation.

The mining input stream concept is a direct consequence of the fact that almost each data mining algorithm requires a data matrix as input. In the language of CWM, we would say: the logical model of data mining is of the *Classifier* type. Thus, *MiningInputStream* extends the CWM class *Class*.

The physical model describes the physical data source that is used for mining, like a text file or a database. For the data mining process, the physical model must be mapped to the logical one.

The physical model describes the physical data source that is used for mining, like a text file or a database. For the data mining process, the physical model must be mapped to the logical one Fig. 12.7.

In XELOPES, this mapping is done by subclassing: different types of physical data sources can be accessed through different mining input stream classes that extend *MiningInputStream*. Important stream classes of XELOPES are listed in Table 12.1. Often, it is useful to write own resource classes which extend *MiningInputStream* or one of its subclasses.

Notice that the last three streams are composed streams which take an arbitrary mining input stream as input and apply a transformation and multidimensional selection/ordering to the stream, respectively.

Layer	Packages					
Management	Warehouse Process	Warehouse Operation				
Analysis	Transformation	OLAP	Data Mining	Business Nomenclature	Information Visualization	
Resource	Relational	Record	Multidimensional	XML		
Foundation	Business Information	Data Types	Software Deployment	Key Indexes	Expres sions	Type Mapping
Object	Core	Behavioral	Relationships	Instance		

Fig. 12.7 CWM packages used in MiningInputStream

Table 12.1 Important resource streams

Resource stream	Super class	Description
MiningArrayStream	*MiningInputStream*	Access to data stored in array
MiningStoredData	*MiningInputStream*	Access to data stored in vector
MiningIteratorStream	*MiningInputStream*	Access to iterator objects
MiningSqlStream	*MiningInputStream*	Access to data stored in database
MiningFileStream	*MiningInputStream*	Access to data stored in a file
MiningCsvStream	*MiningFileStream*	Access to data in CSV file
MiningExcelStream	*MiningFileStream*	Access to data in Excel file
LogFileStream	*MiningFileStream*	Access to data in web server log file
MiningFilterStream	*MiningInputStream*	Access to transformed stream
MultidimensionalStream	*MiningInputStream*	Access to multidimensional stream
MultidimensionalSqlStream	*MultidimensionalStream*	Access to multidimens. SQL stream

Example 12.6 Example of access to a CSV file using the corresponding mining input stream:

```
// Open CSV file 'iris.dat':
MiningCsvStream inputStream = new MiningCsvStream("csv/
  iris.dat");
inputStream.open();

// Get metadata of Iris:
MiningDataSpecification metaData = inputStream.getMetaData();

// Read all data vectors of Iris:
while( inputStream.next() ) {
  MiningVector mv = inputStream.read();
  // ... //
}

inputStream.close();
```

Table 12.2 Examples of resource streams and their CWM representations

Resource stream	CWM package of physical model	Class of package
MiningArrayStream	*Object-oriented* (i.e., *Core*)	*Package*
MiningFileStream	*Record*	*RecordFile*
MiningSqlStream	*Relational*	*Catalog*
MultidimensionalStream	*Multidimensional*	*Schema*

Layer	Packages
Management	Warehouse Process, Warehouse Operation
Analysis	Transformation, OLAP, Data Mining, Business Nomenclature, Information Visualization
Resource	Relational, Record, Multidimensional, XML
Foundation	Business Information, Data Types, Software Deployment, Key Indexes, Expres sions, Type Mapping
Object	Core, Behavioral, Relationships, Instance

Fig. 12.8 CWM packages used in XELOPES transformations

We mention that each mining input stream contains a method *getLogicalModel* which returns its logical model, i.e., the metadata, and a method *getPhysicalModel* which returns its physical model in terms of a CWM resource packages. The physical models for some typical resource streams are listed in Table 12.2. The last column contains the CWM class that is returned as physical model.

In summary, the mining input stream concept makes XELOPES independent of physical data sources. Each mining algorithm takes a *MiningInputStream* as input, probably requests the supported data access methods, and accesses the stream data through the (supported) standard access methods. The physical stream model, i.e., the subclass of *MiningInputStream* passed to the algorithm, is in general not required to be known by a XELOPES mining algorithm.

12.1.2.4 Transformations

We have introduced the basis and vector classes of the attribute space. Next, the transformations of bases and vectors will be discussed. Transformations are a central part of XELOPES Fig. 12.8.

There are two basic types of transformations supported:

- Transformations of mining vectors
- Transformations of mining input streams

Transformations of Mining Vectors

The first type is transformations of mining vectors which implement the *MiningTransformer* interface. It has a simple structure which clearly reflects the XELOPES approach to basis transformations:

```
public interface MiningTransformer
{
  public MiningDataSpecification transform( MiningData
    SpecificationmetaData ) throws MiningException;

  public MiningVector transform( MiningVector vector )
  throws MiningException;
}
```

The first *transform* method transforms basis A into a basis B. The second *transform* method transforms the coordinates of a vector in basis A into the coordinates of the transformed vector in basis B.

We mention three important special cases of vector transformations: if A and B are bases of the same space and the vector is not transformed (but just its coordinates), this is called a *pure basis transformation*. Basis transformations are discussed below. If the bases are equal, i.e., $A = B$ (first *transform* method is identity), and the vector is transformed, this is called a *pure vector transformation*. Further, basis B could be the basis of another space, and then we have a *space transformation*. Often, these types of transformations are mixed.

Back to *MiningTransformer*, its main advantage is the clear separation of basis and coordinate transformations. This has large practical consequences.

Mining Filter Stream

An example of the advantages of separating the basis from the coordinate transformation is the *MiningFilterStream* which applies transformations dynamically to a mining input stream. *MiningFilterStream* is itself a special type of a mining input stream. Its constructor takes an arbitrary mining input stream object *miningInputStream* and a mining transformer object *miningTransformer* as arguments. Then, in the *getMetaData* and *read* methods of *MiningFilterStream*, the transformations of *miningTransformer* are applied to the metadata and mining vectors of *miningInputStream*. The work of *MiningFilterStream* is illustrated in Fig. 12.9.

The other stream methods of *MiningFilterStream* are simply passed to *miningInputStream*. *MiningFilterStream* is universal and easy to use. It can be applied to streams of almost unbounded size. The disadvantage is the lower access speed since each call of read runs a transformation.

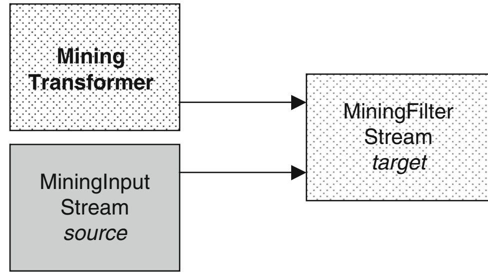


Fig. 12.9 Scheme of dynamic transformations (MiningFilterStream)

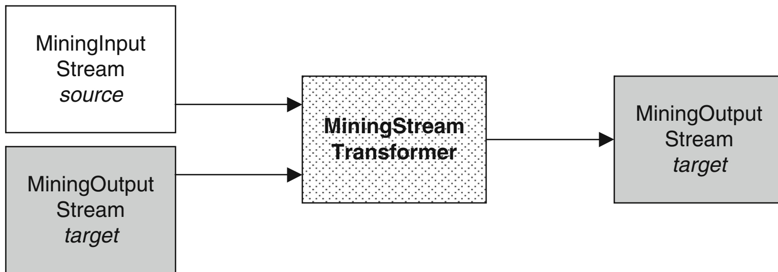


Fig. 12.10 Scheme of static transformations (MiningStreamTransformer)

We have seen that *MiningTransformer* transforms one mining vector into another. Hence, this transformation type transforms attribute values and represents a vector transformation in the attribute space.

Transformations of Mining Input Streams

The transformations of mining input streams are the more general ones and implement the *MiningStreamTransformer* interface. The *MiningStream-Transformer* interface consists of one method *transform* which takes a *source* mining input stream as input and a *target* mining output stream as output of the transformation. Obviously, this type of transformation covers almost any type of transformations of mining input streams. We call this type of transformations *stream transformations* (Fig. 12.10).

Regarding mining input streams, both types of transformations result in the following transformation types:

- *Static transformations*: Here a stream transformation object (that hence implements the *MiningStreamTransformer* interface) converts the source stream into the target stream only *once*, and then the transformed data is available in the target mining input stream.
- *Dynamic transformations*: Here a vector transformation object (that hence implements the *MiningTransformer* interface) is used in *MiningFilterStream*

to run the transformation any time when the *read* method is called (see previous section).

Both types of transformations have advantages and disadvantages: in case of static transformations, the transformation is done only once, and then the target stream contains the complete transformed data. The disadvantage of this approach is that we need two streams to be supported (and usually about twice of memory amount). This means that static transformations are optimal in speed but non-optimal in memory consumption.

For dynamic transformations via mining filter streams, we do not need additional memory, but any time when we access the data from the stream using the *read* method, the transformation of the current vector is carried out again. Hence, dynamic transformations are non-optimal in time but optimal in memory. The static and dynamic transformations represent the classic dilemma that increased speed requires increased memory, and vice versa.

There are many vector transformations implemented into XELOPES based on an extensive CWM framework. For general stream transformations, if they are not based on vector transformations, there is no further framework provided in XELOPES.

Basis Transformations

Basis transformations are very important but also somewhat abstract. Luckily for most applications, the XELOPES user does not have to care about basis transformations because they are automatically executed internally. However, since basis transformations are an important part of this book, we will go more into detail.

For basis transformations, we need to transform metadata (basis) and mining vectors (coordinates) of the application data with respect to the metadata of the mining model, i.e., the metadata of the training data set.

The required basis transformation is addressed by the class *MetaDataOperations* which is a singleton class owned by the metadata class *MiningDataSpecification*. Thus, each *MiningDataSpecification* object owns an object *MetaDataOperations* to transform another *MiningDataSpecification* and appendant mining vectors into its own basis. In addition, *MiningDataSpecification* contains methods like *equals*, *subset*, and *superset* for comparison with another *MiningDataSpecification* object.

In the same way, each *CategoricalAttribute* owns a singleton class *CategoricalAttributeOperations* for basis transformations (because here the categories are referred to as basis of the categorical attribute) from another categorical attribute into the current one and for comparisons. Moreover, even categorical attributes of *unboundedCategories* type are supported by adaptive basis transformation. This is very important because it allows to apply basis transformations, e.g., to categorical attributes of live mining input streams which continuously deliver new categories. An example is the application of a mining model to a large customer database where customers are continuously added during the application process.

The basis transformations of categorical attributes can be embedded into the basis transformation of *MiningDataSpecification* for all categorical attributes included. This results in comfortable nested basis transformation. Caching is used for all metadata and categorical attribute basis transformations. The resulting nested caching ensures the high speed required for basis transformations.

Example 12.7 Again we consider the basis *meal* from Example 12.4 and demonstrate the basis transformation from a second basis *meal2* into *meal*.

```
// -------- Create new metadata 'meal2' ---------
// Create 'cutlery2' attribute with one new category and new
  order:
CategoricalAttribute cutlery2 = new CategoricalAttribute
    ("cutlery");
cutlery2.addCategory( new Category("spoon") );
cutlery2.addCategory( new Category("knife") );
cutlery2.addCategory( new Category("skewer") );
cutlery2.addCategory( new Category("fork") );

// Create new numeric attribute 'price':
NumericAttribute price = new NumericAttribute("price");

// Create new metadata 'meal2':
MiningDataSpecification meal2 = new MiningDataSpecification
    ("meal2");
meal2.addMiningAttribute( cutlery2 );
meal2.addMiningAttribute( numberOfGuests );
meal2.addMiningAttribute( price );
meal2.addMiningAttribute( calories );

// ------- Get basis trafo object of 'meal' -------
MetaDataOperations mealOp = meal.getMetaDataOp();
mealOp.setUsageType(
MetaDataOperations.USE_ATT_NAMES_AND_TYPES_AND_
    CATEGORIES);

// ------- Compare bases 'meal' and 'meal2' -------
System.out.println("'meal' == 'meal2': " +
  mealOp.equals(meal2) );  // false
System.out.println("'meal' subset 'meal2': " +
  mealOp.subset(meal2) );  // true
System.out.println("'meal' superset 'meal2': " +
  mealOp.superset(meal2) ); // false

// ------ Transforms basis of 'meal2' into 'meal' -----
MiningDataSpecification  transMeal2  =  mealOp.transform
    (meal2);
System.out.println("transformed 'meal2': " + transMeal2);
```

```
  // transMeal2 = (calories, numberOfGuests, cutlery) with
  // cutlery = (knife, fork, spoon, skewer) !skewer is added

// ---- Transforms meal vector from 'meal2' into 'meal' ---
// Create meal 'soup' for spoons, 2 persons, 57$, 19 kcal:
double[] soupArr = {0, 2, 57, 19000};
MiningVector soup = new MiningVector(soupArr);
soup.setMetaData(meal2);

// Transform 'soup' from 'meal2' into 'meal' basis:
MiningVector transSoup = mealOp.transform(soup);
System.out.println("transformed 'soup': " + transSoup);
  // transSoup = (19000, 2, 2(=spoon))
```

12.1.3 The Data Mining Framework

12.1.3.1 Models

The abstract class *MiningModel* represents the data mining model which is mainly the mining function. The central method of *MiningModel* is *applyModelFunction* which takes a mining vector as argument and returns the function value. Thus, *applyModelFunction* is used to apply the mining model to data. There exists a second application method *applyModel* which is more general and returns objects (e.g., a mining vector for SV clustering, an item set for an association rule model, or a node for a decision tree model).

Each class representing a type of data mining models extends *MiningModel*. For instance, *AssociationRulesMiningModel* extends *MiningModel* for association rules and contains the implementations (*applyModel* for rules and PMML export/import of rules). For a special association rule model, *AssociationRuleMiningModel* may be further subclassed. For instance, for flat association rules, it may be useful to introduce a new class *FlatRulesMiningModel* which extends *AssociationRulesMiningModel*. XELOPES already contains a wide hierarchy of all basic classes of data mining models including the main implementations like the *apply* methods and PMML serialization. If the user requires a special model, he/she can extend one of the existing models.

Because of the wide variety of data mining models and algorithms, a two-level system of their classification is used.

The *function* level defines the basic types of mining models like *Clustering* and *Regression*. The mining models of XELOPES are organized in packages whose names correspond to the functions. For instance, all classification models are contained in the package *Classification* which contains further subpackages for special classification models.

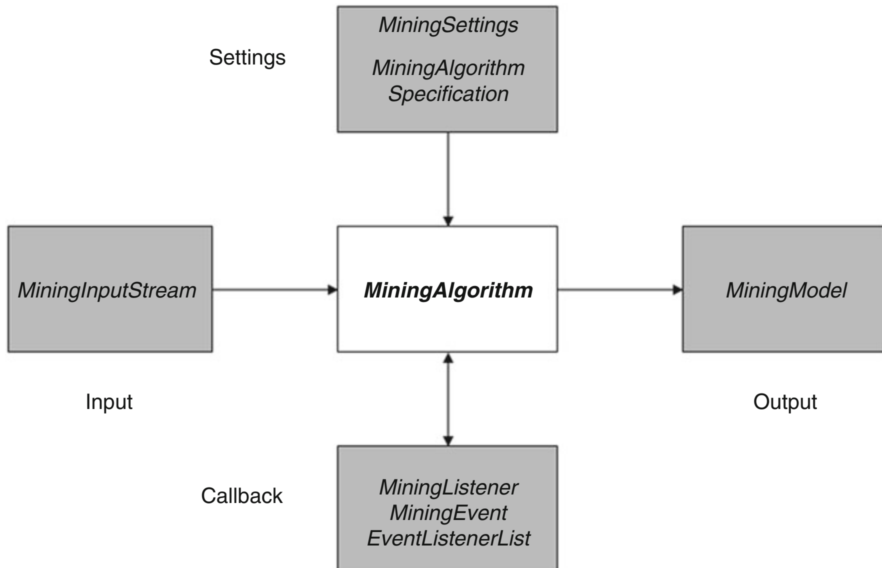


Fig. 12.11 Main interfaces of MiningAlgorithm

The *algorithm* level represents special algorithm types of the functions. Many algorithm types are predefined in CWM, and many have been added for XELOPES. An example is the *decisionTree* algorithm which belongs to the function *Classification* and represents a decision tree.

12.1.3.2 Algorithms

The abstract class *MiningAlgorithm* represents the data mining algorithm that constructs a *MiningModel*.

Thus, *MiningAlgorithm* takes a mining input stream of the training data as input and returns the mining model of the mining function as output. The training parameters are passed through the mining settings on model-type level and mining algorithm specification on algorithm level. Through a callback mechanism, the training process can be monitored and controlled. The complete dataflow is shown in Fig. 12.11.

The central method of *MiningAlgorithm* is *buildModel* which runs the mining algorithm and returns the mining model created by the algorithm. Internally, *buildModel* calls the protected *runAlgorithm* method of the actual training process. The *buildModelWithAutomation* method generates a mining model using techniques for automatic parameter tuning, allowing to build mining models fully automatically.

MiningAlgorithms owns a *verify* method which checks all parameters of the algorithm class for correctness and completeness.

12.1.3.3 Mining Settings

MiningSettings contains the general parameters of a mining model independent of the specific mining algorithm that has created the model. For instance, an association rule settings class must contain the minimum support and confidence parameters because they are required for *each* association rule model. In contrast, parameters like the decomposition size, which is required for specific association rule decomposition algorithms, are contained in the algorithm-specific parameter class *MiningAlgorithmSpecification* that will be described in the next section.

MiningSettings has a reference to its mining model. The most important variable of *MiningSettings* is *dataSpecification* of the class *MiningDataSpecification*. It contains the metadata of the training data used to build the model and is referred to as the *metadata of the mining model*.

MiningSettings contains a *verifySettings* method which checks all parameters of the settings class for correctness and completeness.

Similar to *MiningModel*, each class representing a type of data mining settings extends *MiningSettings*. For the example at the beginning of this section, the settings of association rule models are contained in the class *AssociationRuleSettings* which extends *MiningSettings*. Of course, this is the settings class associated with *AssociationRuleMiningModel* mentioned in the previous section.

Along with all mining models, XELOPES provides their associated settings classes containing all basic parameters of the respective models.

MiningSettings contains the same variables *function* and *algorithm* for storing the function and algorithm type of the mining model. Their values are identical to those of the associated mining model.

12.1.3.4 Mining Algorithm Specification

The algorithm-specific class *MiningAlgorithmSpecification* contains the function and algorithm, the name, the class path, the version, and an array of specific parameters of a mining algorithm. This array contains the specific parameters defined by the *MiningAlgorithmParameter* class. Every parameter is described by its name, type, value, description, and setter method and contains the reference to its associated *MiningAlgorithmSpecification* object.

In most XELOPES distributions, the complete information of *MiningAlgorithmSpecification* for all algorithms and parameters is stored in the configuration file *algorithms.xml*.

Example 12.8 Example of the section of the fast sequential algorithm *Sequential* of *algorithms.xml*:

```
<AlgorithmSpecification name="Sequential"
  function="Sequential"
  algorithm="sequenceAnalysis"
```

```
  classname="com.prudsys.pdm.Models.Sequential.Algo-
     rithms.Seq.SequentialCycle"
  version="1.0">
  <AlgorithmParameter name="minimumItemSize"
    type="int"
    value="1"
    method="setM_minItemSize"
    description="Minimum size for large items" />
  <AlgorithmParameter name="maximumItemSize"
    type="int"
    value="-1"
    method="setM_maxItemSize"
    description="Maximum size for large items" />
</AlgorithmSpecification>
```

Algorithm Types

Similar to *MiningModel* and *MiningSettings*, each class representing a type of data mining algorithms extends *MiningAlgorithm*. For example, the general class of association rule algorithms is *AssociationRulesAlgorithm* which extends *MiningAlgorithm*. Again, this is the algorithm class associated with *AssociationRulesMiningModel* and *AssociationRulesSettings* mentioned in before.

Along with all mining models and their mining settings, XELOPES provides the associated algorithm classes containing the basic implementations.

Example 12.9 We give an example of the whole data mining process for sparse grid classification (Chap. 7). First, we build the sparse grid model. The training data is contained in a CSV file whose path is specified in *TRAIN_FILE*. The target attribute is supposed to be the last one. We apply (0,1) normalization to all numeric attributes before we build the model. The resulting sparse grid model is written to the PMML file *SparseGridsModel.xml*.

The Java code is given below:

```
// Open data source and get metadata:
MiningInputStream inputData = new MiningCsvStream
   ( TRAIN_FILE );
inputData.open();
MiningDataSpecification metaData = inputData.getMetaData
    ();

// Get target attribute (last one):
MiningAttribute targetAttribute =
```

```
metaData.getMiningAttribute( metaData.getAttribu-
    tesNumber ()-1 );

// (0,+1) Normalization of all attributes:
LinearNormalStream lns = new LinearNormalStream
    ( inputData);
lns.setLowerBound(0);
lns.setUpperBound(+1);
lns.setExcludedAttributeName(targetAttribute.getName());

// Create transformation object:
MiningTransformationActivity mta = new MiningTransforma-
 tionActivity();
 mta.addTransformationStep( lns.createMiningTransforma-
 tionStep() );

// Create MiningSettings object:
SparseGridsSettings miningSettings = new SparseGrids-
    Settings();
miningSettings.setDataSpecification( metaData )
miningSettings.setTarget( targetAttribute );

// SG settings:
miningSettings.setSgType(SparseGridsSettings.SG_TENSOR_
    PRODUCT_BASIS_TYPE);
miningSettings.setCoarseGrid(true);
miningSettings.setLevel(4);
miningSettings.setLambda(0.1);
miningSettings.verifySettings();

// Get default mining algorithm specification from 'algo-
    rithms.xml':
MiningAlgorithmSpecification miningAlgorithmSpecification
  =
  MiningAlgorithmSpecification.getMiningAlgorithmSpeci-
    fication( "Sparse Grids" );

// Get class name from algorithms specification:
String className = miningAlgorithmSpecification.
    getClassname();

// Set and display mining parameters:
miningAlgorithmSpecification.setMAPValue("debug", "1" );
GeneralUtils.displayMiningAlgSpecParameters(miningAl-
    gorithmSpecification );

// Create algorithm object with default values:
```

```
MiningAlgorithm algorithm= GeneralUtils.createMiningAl-
    gorithmInstance(className);

// Put it all together:
algorithm.setMiningInputStream( inputData );
algorithm.setOuterMiningTransform( mta );
algorithm.setMiningSettings( miningSettings );
algorithm.setMiningAlgorithmSpecification( miningAlgor-
    ithmSpecification );
algorithm.verify();

// Build the mining model:
MiningModel model = algorithm.buildModel();
System.out.println("calc. time[s]: " + algorithm.
    getTimeSpentToBuildModel());

// Write to PMML:
FileWriter writer = new FileWriter("data/pmml/
    SparseGridsModel.xml");
model.writePmml(writer);
```

We now apply the sparse grid model of the PMML classifier *SparseGridsModel.xml*, created before, to a CSV file *TEST_FILE* and calculate the classification rate. Before applying the classifier, the normalization taken from the model is carried out:

```
// Read SG model from PMML file:
SparseGridsMiningModel model = new SparseGridsMiningModel
    ();
FileReader reader = new FileReader("data/pmml/
    SparseGridsModel.xml");
model.readPmml(reader);
MiningAttribute modelTargetAttribute = ((Supervised
    MiningSettings)
  model.getMiningSettings()).getTarget();
System.out.println("----> PMML model read successfully");

// Open data source and transform into model format:
MiningInputStream inputData0 = new MiningCsvStream(
    TEST_FILE );
MiningInputStream inputData = model.transformIntoMo-
    delFormat(inputData0);

// Get input metadata:
MiningDataSpecification inputMetaData = inputData.
    getMetaData();
CategoricalAttribute inputTargetAttribute =
    (CategoricalAttribute)
```

```
  inputMetaData.getMiningAttribute( modelTargetAttribute.
     getName() );

// Classification:
int nall  = 0;
int nwrong = 0;
while ( inputData.next() ) {

  // Make prediction:
  MiningVector vector = inputData.read();
  double predicted   = model.applyModelFunction(vector);
  Category predCateg  = modelTargetAttribute.getCategory
     (predicted);
  double real     = vector.getValue(inputTargetAttribute);
  Category realCateg  = inputTargetAttribute.getCategory
     (real);

  // Compare;
  if (! predCateg.equals(tarCateg) )
    nwrong = nwrong + 1;
  nall = nall + 1;
};

System.out.println("classification rate = " + (100.0 -
  wrong*100.0/nall) );
```

■

12.1.4 The Mathematics Package

XELOPES is equipped with a useful package *Math* of basic mathematical operations. It is only a utility package for the XELOPES analysis algorithms and does not claim to be a complete mathematical library. Nevertheless, especially for numerical linear algebra, it contains some remarkable implementations of general interest.

The *Math* package contains the following subpackages:

- *Algebra*: vectors, matrices, tensors, and important solver and factorizations
- *Analysis*: functions, derivatives, etc.
- *Approximation:* basic approximation methods
- *Optimization:* basic optimization methods
- *Tools:* useful tools

In the following, we want to delve into the package *Algebra*. It has three subpackages:

- *Core:* data structures for vectors, matrices, tensors
- *Solver:* solvers
- *Factorizations:* factorizations

The background is that the LAPACK and BLAS libraries, which are the de facto standard in numerical linear algebra software (e.g., MATLAB is based on LAPACK), cannot be easily adapted for Java. This problem is rooted in the strong typing of Java: otherwise an advantage, here it inadequately blows up the number of methods of the algebraic libraries. Whereas in FORTRAN or C a scalar value may be passed to a method the same way as an array, Java does not allow this and requires a separate method for all possible combinations of data types. In the result, the number of methods grows exponentially with the number of arguments.

There are two ways to fix this problem. LAPACK/BLAS can be automatically converted from FORTRAN/C into Java using special converters. The advantage of this approach is that compatibility is retained and so existing implementations of algorithms, which use LAPACK/BLAS, automatically can be converted into Java. The price to be paid is the confusing form of the generated Java code which renders a manual usage of the emerged Java BLAS virtually impossible.

For the second approach, we abstain from the BLAS compatibility and create a library instead which offers a functionality similar to LAPACK and BLAS but exploits the advantages of Java. First of all, this is object orientation. For the XELOPES library, we used the second way. Unfortunately, here only few suitable implementations exist, and so most had to be newly developed. Although XELOPES includes only parts of the LAPACK/BLAS functionality, the implementation is very clear and sufficient for most applications.

12.1.4.1 Core

Vector

For vectors, the Core package offers BLAS level-1-like classes *VBAS* for the core vector operations, *VBASJ* for Java-specific extensions, and *VGEN* for general extensions like index calculations and displaying vectors. We will not go into further details here.

Matrix

As starting point for matrices, the extremely lean JAMA (Java Matrix Package) library was used. It mainly consists of the class *Matrix*, representing a dense matrix, as well as a handful classes of basic decompositions and solvers (Cholesky, LU, QR, eigenvalues, SVD located in the *Solvers* and *Factorizations* subpackages). Despite of its minimalistic implementation, JAMA is quite powerful. Roughly

speaking, it represents BLAS levels 2 and 3 (matrix operations) plus main LAPACK functionality (especially decompositions). Thus, we included JAMA into the *Algebra* package and extended the *Matrix* class for further requirements.

Example 12.10 We consider two matrices A and B which are defined as follows:

$$A = \begin{pmatrix} 1 & 4 \\ 3 & -5 \end{pmatrix},\ B = \begin{pmatrix} 1 & 2 & -3 \\ 0 & 0 & 1 \end{pmatrix}.$$

Now we demonstrate the generation and operations on A and B.

```
// Create A and B:
Matrix A = new Matrix( new double[] [] {{1.,4.},{3.,-5.}} );
Matrix B = new Matrix( new double[] [] {{1.,2.,-3},{0.,0.,1.}} );

// Transpose B. BT = B^T:
Matrix BT = B.transpose();

// Scalar multiplication. A2 = 2*A:
Matrix A2 = A.times(2);

// Matrix multiplication. C = A*B:
Matrix C = A.times(B);

// Matrix addition. D = B + C:
Matrix D = B.plus(C);

// Frobenius norm:
double nF = A.normF();

// Concatination of operations. E = (B + 2*A*B + C)^T:
Matrix E = B.plus( A.times(B).times(2) ).plus(C).trans-
    pose();
```

■

Since JAMA only supports dense matrices, it was extended for sparse matrices. Therefore, the abstract class *SparseMatrix* was designed, and a number of implementations of this class had been added. The corresponding class diagram is depicted in Fig. 12.12.

The different implementations of *SparseMatrix* use different storage techniques and are optimized for different applications.

So the classes *SparseMatrixCompRow* and *SparseMatrixCompRowStatic* are based on the format *Compressed Row Storage* (*CRS*). Here all nonzero elements (NZE) are stored in one array, and a second array contains the corresponding column indexes, while a third array contains the pointers to the rows. The CRS format is especially suited for fast matrix–vector multiplications. At the same time, it is relatively static because inserting and deleting of NZEs in general require all arrays to be reordered.

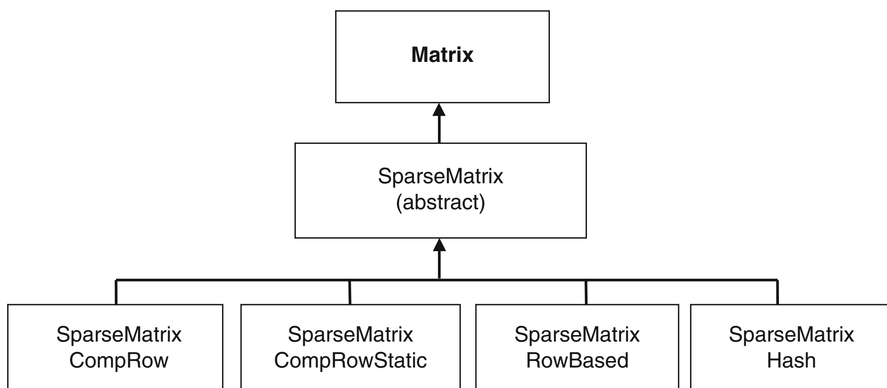


Fig. 12.12 Class hierarchy of matrices

The class *SparseMatrixRowBased* uses the storage format *Direct Row Storage* (*DRS*), where the matrix is stored as array of arrays whose last contains the NZEs with their column indexes for each row. The DRS format is more dynamic than the CRS format concerning changes of its structure but executes slower matrix–vector multiplications. The class *SparseMatrixHash* is entirely based on a hash and thus can be manipulated very easy and fast, but the execution of most operations is slower compared to the CRS and DRS formats.

The *Matrix* class and all of its *SparseMatrix* implementations can be easily converted into each other by special constructors and conversion methods. In this way at the beginning of blocks of algebraic operations, the sparse matrices can be converted into the format that is most efficient for these operations.

Example 12.11 We return to Example 12.10 and introduce a further matrix

$$F = \begin{pmatrix} 0 & 2 & 4 \\ 0 & 0 & 0 \\ 3 & 11 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

that we will use for demonstration of sparse matrices. First, we store F in CRS format and convert it into a matrix G in DRS format that we use for further calculations:

```
// Create F as CRS:
double[] values = {2,4,3,11,2,1};
int[] columnIndexes = {1,2,0,1,2,2};
int[] rowPointer = {0,2,2,5,6};
SparseMatrixCompRowStatic F =
  new SparseMatrixCompRowStatic(4, 3, values,
      columnIndexes, rowPointer);
```

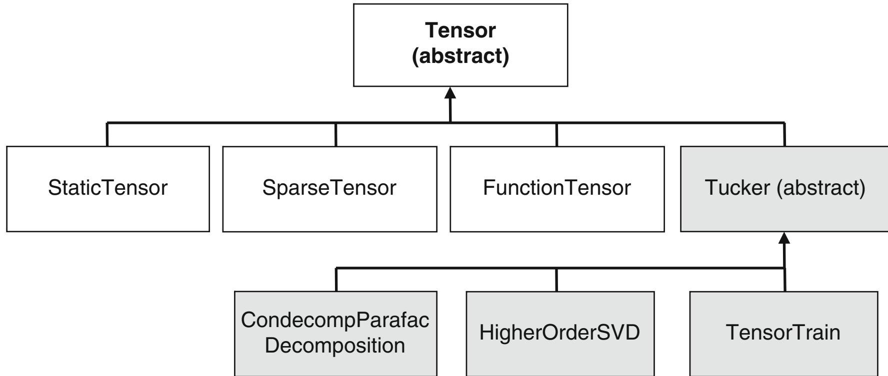


Fig. 12.13 Class hierarchies of tensors. The classes in the gray boxes belong to the Factorizations subpackage

```
// Create G as DRS from F:
SparseMatrixRowBased G = new SparseMatrixRowBased(F);

// Matrix multiplication. H = G*B^T:
Matrix H = G.times( B.transpose() );

// Vector multiplication. w = G*v:
double[] v = {1, 2, 0};
double[] w = G.mult(v);

// Convert G to full matrix I:
Matrix I = G.toMatrix();
```
■

Tensor

The abstract class *Tensor* is the root of all tensor implementations. The tensor class hierarchy is shown in Fig. 12.13. The factorization part will be explained later.

In general, the tensor implementations are built upon a column-based structure. That is, content is stored in columns rather than rows.

For working with a dense tensor, the class *StaticTensor* shall be used. In this case, the tensor is stored in a one-dimensional double array. The indexes of the array are linearized tensor entries. As depicted in Fig. 12.13, there exists also a sparse version of tensors – the class *SparseTensor*. This class uses a hash map to store nonzero. Thus, the linearized indexes form the keys, and the values are the corresponding tensor entries. The class *FunctionTensor* is a wrapper for a high-dimensional function which is provided through the *TensorFunction* interface. The tensor classes implement a set of central operations of *Tensor* (Sect. 9.1.1):

0	0	0
0	0	1
0	-1	0

0	0	-1
0	0	0
1	0	0

0	1	0
-1	0	0
0	0	0

Fig. 12.14 The Levi-Civita symbol

- Addition/subtraction
- Contraction
- Inner product
- Outer (tensor) product
- Contracted (multilinear) product
- *n*-Mode multiplication
- Matricization

We first give an example of the initialization of a sparse tensor Fig. 12.14.

Example 12.12 In order to initialize the Levi-Civita symbol, which can be interpreted as sparse tensor, the following steps are performed:

```
int[] dimension = { 3, 3, 3 };
SparseTensor tensor = new SparseTensor( dimension );
tensor.setEntry( 5, -1 );
tensor.setEntry( 7, 1 );
tensor.setEntry( 11, 1 );
tensor.setEntry( 15, -1 );
tensor.setEntry( 19, -1 );
tensor.setEntry( 21, 1 );
```

or, using a hash map,

```
HashMap<Integer, Double> entries = new HashMap<Integer,
    Double>;
entries.put( 5, -1 );
entries.put( 7, 1 );
entries.put( 11, 1 );
entries.put( 15, -1 );
entries.put( 19, -1 );
entries.put( 21, 1 );

int[] dimension = { 3, 3, 3 };
SparseTensor tensor = new SparseTensor( entries, dimension
    );
```

After the tensors have been defined, we can perform operations on them. This is illustrated in the next example.

Example 12.13 Suppose we have created two static tensor objects *t1* and *t2*. We demonstrate some tensor operations, then create the corresponding sparse tensors, and use them again for tensor operations:

```
// Element-wise multiplication T1 x T1:
Tensor tt1 = t1.mult( t1 );

// Tensor multiplication T1 x T2:
Tensor t12 = t1.tensorMult( t2 );

// N-mode multiplication T1 x T2 (N = 1):
Tensor tn12 = t1.nModeMult( t2.toMatrix(), 1 );

// Inner product T1 x T2:
Tensor ti12 = t1.innerProduct( t2, 0, 0 );

// Matricization of T1 (n-mode = 1):
Matrix m1 = t1.matrice( 1 );

// Create the sparse tensors:
SparseTensor sp1 = new SparseTensor( t1 );
SparseTensor sp2 = new SparseTensor( t2 );

// SparseTensor1 + SparseTensor2:
SparseTensor sp12 = sp1.plus( sp1 );

// N-mode multiplication SparseTensor1 x T2 (N = 1):
SparseTensor spn12 = sp1.nModeMult( t2.toMatrix(), 1 ); ■
```

12.1.4.2 Factorizations

The package *Factorizations* contains two subpackages:

- *Matrix*: matrix factorizations (Sects. 8.3 and 8.4)
- *Tensor*: tensor factorizations (Sects. 9.1, 9.2, and 9.3)

Matrix Factorizations

The matrix factorization package contains basic decompositions from JAMA for dense matrices, namely, Cholesky, LU, QR, eigenvalues, and SVD. Further decompositions are for sparse matrices of large dimensions. These include Lanczos for eigenvalues, SVD and Lanczos vectors, the adaptive SVD of Sect. 8.3, an SVD based on a gradient descent method as of Sect. 8.5, different ALS versions,

nonnegative matrix factorizations (ALS, multiplicative, SVD based), and a cross-approximation algorithm.

Example 12.14 We demonstrate the use of a truncated SVD for a small, sparse 3 × 3 matrix:

```
// Create sparse 3x3 matrix SA:
double[][] valsA = {{1.,2.,-3},{0.,1.,1.}, {1.,0.,1.}};
Matrix A = new Matrix(valsA);
SparseMatrixCompRowStatic AS = new SparseMatrixCom-
   pRowStatic(A);
E.print(2, 2);

// Perform Lansczos SVD, truncate for rank 2:
int rank = 2;
LanczosSVD svd = new LanczosSVD(SA, rank, true);

// Get right singular vector:
Matrix rvec = svd.getV();
rvec.print(2, 2);

// Get singular values:
double[] sv = svd.getSingularValues();
VGEN.DDisplay("svec:", sv);

// Get left singular vector:
Matrix lvec = svd.getU();
lvec.print(2, 2);
```

■

Tensor Factorizations

The *Tucker* class (Definition 9.3) contains general methods of Tucker decomposition, and the classes *CandecompParafacDecomposition* (Definition 9.5), *HigherOrderSVD* (Definition 9.4), and *TensorTrain* are the specific factorization models. *HigherOrderSVD* additionally contains the adaptive HOSVD Algorithm 9.2. Algorithms for offline calculations of decompositions are located in the subpackage *Algorithms*. In order to calculate a tensor decomposition, e.g., the classes *CPDecompositionAlgorithm*, *StandardHosvdAlgorithm/TruncatedHosvdAlgorithm* (Lanczos algorithm), and *TuckerCross3DAlgorithm* can be applied. They contain a method *buildModel(Tensor t)* to calculate the decomposition.

Example 12.15 We demonstrate the use of a truncated HOSVD for a 3 × 4 × 3 tensor.

```
// Create 3x4x3 tensor t1 and sparse tensor sp1:
int[] dims = { 3, 4, 3 };
StaticTensor t1 = new StaticTensor( dims );
```

```
t1.setRandomIntegerEntries( 0, 1 ); // fill tensor with ran-
    dom entries
SparseTensor sp1 = new SparseTensor( t1 );

// Perform HOSVD, truncate for specified ranks:
TruncatedHosvdAlgorithm truncSVD = new TruncatedHosv-
   dAlgorithm();
int[] ranks = { 2, 3, 3 };
HigherOrderSVD hosvd = truncSVD.buildModel( sp1, ranks );

// Get core tensor:
Tensor core = hosvd.getCore();
core.print(2, 2);

// Get mode factors:
for ( int i = 1; i < t1.getOrder() + 1; i++ )
{
  Matrix U = hosvd.getMatrices()[i - 1];
  U.print(2, 2);
}
```

For the same example, we demonstrate the incremental HOSVD. Suppose we add one slice to the existing tensor and update the model, then we need to add the following code to the previous one:

```
// Add slice on mode 3:
SparseMatrixHash slNew = new SparseMatrixHash( sp1.
    getDimensions()[0],
  sp1.getDimensions()[1] );
setRandomIntegerEntries( slNew, 0, 1, 0.2 ); // fill matrix
    with random entries
int nMode = 3;

// Update HOVSM:
hosvd.update( sp1, slNew, nMode );

// Add slice to sparse tensor:
sp1.addSlice( slNew, nMode );

// Compare results (updated HOSVD and sparse tensor in
    Frobenius norm):
Tensor spEnd = hosvd.getCore()
  .nModeMult( hosvd.getMatrices()[0], 1 )
  .nModeMult( hosvd.getMatrices()[1], 2 )
  .nModeMult( hosvd.getMatrices()[2], 3 );

double normF = spEnd.minus( sp1 ).getFNorm();
System.out.printf("F-Norm of the difference: " + normF); ■
```

12.2 The Realtime Analytics Framework of XELOPES

Neither the CWM nor the PMML standard supports realtime analytics functions. So it was newly specified in XELOPES extending the existing framework. We first give an introduction to the agent framework of XELOPES which is more general than reinforcement learning only. Based on this, we then explain the reinforcement package and finally the recommendation package which in turn extends the RL.

12.2.1 The Agent Framework

The agent framework which is implemented for every single agent in XELOPES is inspired by the agent ansatz of artificial intelligence (see [RN02]). The heart of this framework is the *Agent*, an object which interacts with an environment. This agent consists of sensors to receive stimuli from the environment and actuators to perform actions inside of the environment and with it response to the received stimuli.

We mention that nearly everything can be explained in terms of such agent theory. For example, think of a calculator which gets the stimulus "2 + 2" and, as a result, responses with the action "4." Despite to this, we will consider the agent concept for the analysis of systems mainly. In this context, examples for agents are given through pack robots, interactive English teachers, systems for medical diagnostics, and many more. To determine how the agent should respond to a certain stimulus, some rules have to be introduced. Dependent on the environment, especially the number of different stimuli, this can lead to an innumerable number of rules that cannot even be stored on the best performing computers of nowadays.

To help this out, rules are defined which handle more than one stimulus, actions are chosen also randomly, and reward functions are defined. Randomly chosen actions are actually necessary for environments which are not fully observable which implicates that not the whole variety of possible stimuli is known. A reward function measures the success of an action through a reward which, for instance, could be a real number and is communicated to the agent. The aim of the agent is to maximize this reward. Through the corresponding value function, the agent is able to rate possible choices of actions in response to a stimulus. This allows him to make a reasonable decision. The storing and applying of the corresponding rules are described as learning, since these rules are not initially given.

The agent described until now is a so-called stateless agent. In contrast to this, we can also consider stateful agents. These agents include an additional attribute, the state. The state of the agent can change as a result of an action. The current state of the agent is taken into account during the decision process.

The XELOPES *Agent* package consists of several utilities which provide a unique access to the realtime applications of the XELOPES. We need to mention that the environment discussed above, by now, is not modeled in XELOPES but will presumably be added in a future version. This means stimuli of the environment are only

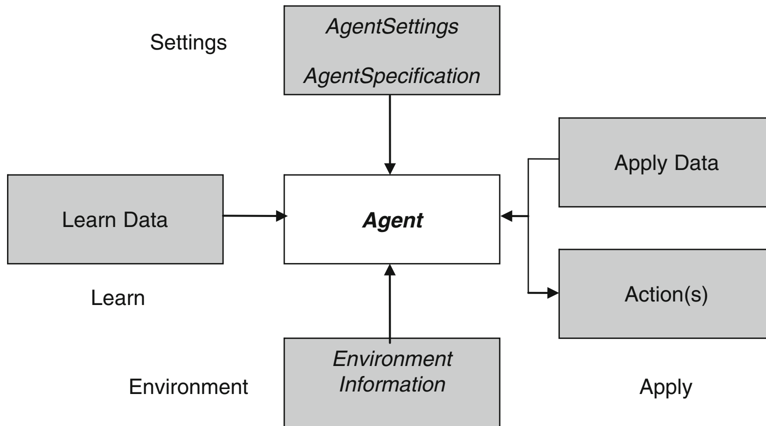


Fig. 12.15 Main interfaces of agent

covered through new data which is handed to the agent in a so-called learn step. Furthermore, the agent needs to be triggered to take an action. This is in contrast to some common agent theories, where an agent decides by himself or herself when to act.

12.2.1.1 Agent

The class *Agent* which represents the root of all agent implementations is abstract. The dataflow is shown in Fig. 12.15. If we compare it with the dataflow of data mining of Fig. 12.11, we see that the main idea is to join the classes of the algorithm and the model into one class – the agent. This is because unlike as in classical data mining an agent does both learning and application, often combined. As we see, the agent parameters are passed through the agent settings on agent-type level and agent specification on algorithm level – very similar to mining settings and mining algorithm specification of the mining process. Additional information like product-specific *master* data required in the specific agent applications can be specified through the environment information which, however, is mandatory.

EnvironmentInformation is basically a hash map of mining input streams where the keys are the names of the streams. Thus, each stream represents a table. For example, there may be a key *product* for a mining input stream of all product informations of a web shop and *regions* for tabular informations about different regions. The inclusion of *EnvironmentInformation* into the *Agent* package is important because unlike data mining, which almost always works on a flat table, agents are often more complex and work on nested schemas. For example, XELOPES contains business-oriented packages of disposition and price optimization as well as a reinforcement learning package described in the next section.

Before we explain the learning and application method of an agent, we mention that we distinguish between stateless and stateful agents. A stateless agent, in contrast to a stateful one, does only store some specific agent parameters representing the rules, not the input data. This means the agent does not know its current state and makes its decisions independent of his state. On the contrary, a stateful agent knows its “history.” In general, stateful agents are easier to implement and to integrate. However, stateless agents support multithreading environments. For example, a recommendation agent working in real time is requested by events of different sessions in a mixed order.

Now we turn to the learning step. If the agent is stateful, the method *addData* with a mining vector or a mining input stream as argument is used to add training data to the agent. Each time *addData* is called, its data is added to the internal input data vector of the agent. The method *clearData* removes all input data from the vector. The method *learn* runs the (adaptive) learning based on the input data. There are two other *learn* methods with a mining vector or a mining input stream as argument, respectively. These methods, which can be applied to both stateful and stateless agents, use only the data passed as argument for learning.

For application, *Agent* provides two *apply* methods, both applicable for stateful and stateless agents. The first takes a mining vector as argument and returns an action, specified by a generic (of the *Agent* class). In case of scoring, this may be a Double or Integer, but an action may also have a more complex object like a recommendation. The second *apply* method receives a mining input stream as argument and returns a list of the generics representing the actions corresponding to each mining vector of the mining input stream.

To combine learning and application in one step, there exist *learnApply* methods. Within these methods, both stateful and stateless agents use the input *miningInputStream*/*miningVector* for learning and application.

By means of this simple set of learning and application methods, a wide variety of realtime learning scenarios can be covered. Note that depending on the implementation of a specific agent, not all of these methods must be supported.

Like *MiningAlgorithm*, also *Agent* owns a *verify* method which checks all parameters of the agent class for correctness and completeness. Similar to *MiningModel*, *Agent* has a variable *function* to specify its basic agent type but no variable similar to *algorithm* because each agent is usually enough specific, and so clustering of agents on algorithm level does not make sense.

12.2.1.2 Agent Settings

AgentSettings contains the general parameters of an agent and is very similar in nature to *MiningSettings*. It also contains a reference to the agent and the metadata of the learning/application data. Moreover, *Agent* owns a *verifySettings* method which checks all parameters of the settings class for correctness and completeness.

Similar to *MiningSettings*, each class representing a type of agent settings extends *AgentSettings*. Along with all agents, XELOPES provides their associated settings classes containing all basic parameters of the respective agents.

AgentSettings contains the same variable *function* or stores the function of the agent. A specific property of agent settings is the methods to request and to define whether the agent works in a stateless or stateful mode. Of course, not all agents support both modes.

12.2.1.3 Agent Specification

The agent-specific class *AgentSpecification*, which is the agent counterpart to *MiningAlgorithmSpecification*, contains the function, the name, the class path, the version, and an array of specific parameters of an agent. This array contains the specific parameters defined by the *AgentParameter* class. Every parameter is described by its name, type, value, description, and setter method and contains the reference to its associated *AgentSpecification* object.

In most XELOPES implementations, the complete information of *AgentSpecification* for all agents and parameters is stored in the configuration file *agents.xml*.

Example 12.16 Example of the section of price optimization algorithm *DiscountAgent* of *agents.xml*:

```
<AgentSpecification name="DiscountAgent"
  function="PriceOptimization"
  classname="com.prudsys.pdm.Agent.Pricing.Discount.
     DiscountAgent"
  description="Discount Price Optimization Agent."
  version="1.0">
  <AgentParameter name="initC"
    type="double"
    value="10.0"
    method="setInitC"
    description="Initial price elasticity after the first
       order."/>
  <AgentParameter name="debug"
    type="java.lang.String"
    value="none"
    method="setDebug"
    description="Possible values:none, all, a list of
       itemIDs."/>
  </AgentSpecification>
```

Agent Types

Similar to *AgentSettings*, each class representing a type of agents extends *Agent*. For example, the general class of price optimization agents is *PricingAgent* which extends *Agent*. This is the agent class associated with *PricingAgentSettings*.

Along with all agent settings, XELOPES provides the associated agent classes containing the basic implementations.

Example 12.17 We give a (simplified) example of price optimization using an agent which calculates the optimal prices of products. The transaction data is contained in a CSV file whose path is specified in *TRANSACT_FILE*. For each transaction product, the agent calculates the optimal price, and based on the resulting orders (or not order), the agent learns. The resulting elasticity agent is written to the PMML file *LinElastAgent.xml*.

```
// Open data source and get metadata:
MiningInputStream inputData = new MiningCsvStream(
    TRANSACT_FILE );
inputData.open();
MiningDataSpecification metaData = inputData.getMetaData
    ();

// Create AgentSettings object:
PricingAgentSettings settings = new PricingAgentSettings
    ();
settings.setInputDataSpecification( metaData );
settings.setUseRoundPrice(true);
settings.setRoundPriceRule(1.2, 0.25, 0.98);
settings.setRoundPriceRule(1.3, 0.25, 0.98);
settings.verifySettings();

// Get agent specification from 'agents.xml':
AgentSpecification agentSpecification =
  AgentSpecification.getAgentSpecification(
     "LinElastAgent" );

// Set agent parameters:
agentSpecification.setAPValue("explorationRate ", 0.1);
agentSpecification.setAPValue("priceEpsilon", 0.02);
agentSpecification.setAPValue("movingAverageTurno-
    verRange", 1.0);

// Create agent object:
PricingAgent agent = (PricingAgent) agentSpecification.
    createAgentInstance();

// Create environment object:
EnvironmentInformation env = getPoEnvironment();  // not
    shown here
```

```
// Put it all together:
agent.setAgentSettings(settings);
agent.setAgentEnvironmentInformation(env);
agent.verify();

// Apply and learn:
int indItem = metaData.getAttributeIndex("itemID");
int indUnits = metaData.getAttributeIndex("units");
while ( inputData.next() ) {
  MiningVector mv = inputData.read();
  Category itemID = mv.getValueCategory(indItem);

  // Get recommended price:
  double price   = agent.apply(mv);

  // Recommend price for item ID, get units ordered (0 if no
     order):
  int unitsOrdered = getResponse(itemID, price); // not
     shown here
  mv.setValue(indUnits, unitsOrdered);

  // Learn from response (stateful):
  agent.addData(mv);
  agent.learn();
  // Alternative - Learn from response (stateless):
  // agent.learn(mv);
}

// Write to PMML:
FileWriter writer = new FileWriter("data/pmml/
   LinElastAgent.xml");
agent.writePmml(writer);
```
■

Finally, we notice that the *Agent* package provides a useful wrapper class *MiningModelAgent*, which extends *Agent*, in order to use data mining models/ algorithms as described in Sect. 12.1.3 inside the agent framework. This is of particular interest for updateable mining models such as naïve Bayes or adaptive decision trees. At this, to *MiningModelAgent*, a *MiningModel* and a suitable *MiningAlgorithm* can be assigned and can then be used as an agent.

12.2.2 The Reinforcement Learning Package

In XELOPES, the reinforcement learning package is implemented as subpackage RL of the *Agent* package (previous section). The RL package in turn contains the following subpackages:

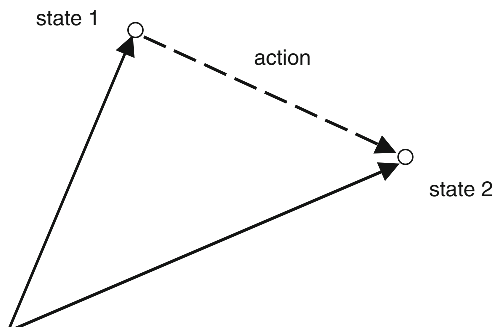


Fig. 12.16 Symbolic vector representation of state transition by taking an action

- *Core:* core and infrastructure
- *DP:* dynamic programming algorithms of RL
- *MC:* Monte Carlo algorithms of RL
- *TD:* temporal-difference learning algorithms of RL
- *Approx:* function approximation
- *MultiLevel:* multilevel methods
- *Recomm:* RL for recommendations

The RL algorithms implement the agent interface described in the previous section. We now describe the central packages except for the *Recomm* package which will be studied in Sect. 12.2.3.

12.2.2.1 Core

State, Action, Reward

The class *State* extends *MiningVector*. To avoid philosophical discussions why a state is a mining vector, we just mention that since it is used as argument of the state-value function *v(s),* which in turn is represented by a *MiningModel* (will be explained below), it must be a mining vector. The class *Action* also extends *MiningVector*. Figure 12.16 illustrates the motivation: an action is something that moves one state into another. Since states are mining vectors (coordinate vectors), actions must be mining vectors (transition vectors), too!

The classes *State* and *Action* do not directly extend *MiningVector* but the class *IndexedMiningVector* which is a mining vector with an index, accessible via *getIndex* and *setIndex* methods. The index is useful for discrete state and action sets *S* and *A(s)*, respectively. The index, as almost all integer-like RL implementations in XELOPES, uses *long* as data type because in RL there may be a huge number of states, actions, steps, etc., that potentially cannot be stored as native integers.

The class *Reward* is mainly a wrapper class for a double value. Due to the fact that *Reward* always contains one value only, instead of an array, it does not extend

MiningVector although this might be useful in some conceptual sense as we will discuss next.

The class *StateActionVector*, which also extends *IndexedMiningVector*, represents a mining vector composed of a state and an action. This class is required to model action-value functions within the XELOPES framework as *MiningModel*, as will be described below. There are also other similar classes that extend *IndexedMiningVector* and represent compositions of states, actions, and rewards. Further examples are *StateRewardVector* (state and reward), *SampleVector* (state, action, reward, next state), and *SarsaVector* (extends *SampleVector* by additional next action and thus represents state, action, reward, next state, next action). All these classes in the end extend *MiningVector* what is extremely helpful because we can use them in the same *learn* and *apply* methods of the agent.

StateSet, ActionSet, StateActionSet

The class *StateSet* models the state set *S*, and *ActionSet* the actions set *A(s)* of all actions available in state *s*. Both are containers of mining vectors of states and actions, respectively. The class *ActionSet* further contains a reference to its state *s*.

Because the number of valid states and actions may be infinite and even uncountable, *MiningVectorSet* is used to store the vectors (Sect. 12.1.2). In case of discrete problems, a *MiningInputStream* as subtype of *MiningVectorSet* is used to store the vectors; we then call the mining vector set *countable* what is indicated by the *isCountable* method. For countable vector sets, there exist specific methods like *nextState* (*nextAction*) and *readState* (*readAction*) which repeat the methods of *MiningInputStream* but are more simple to use because explicit-type conversions (e.g., to *State* object or *long* type) can be avoided. Additionally, the methods *addState* (*addAction*) not only add a new *state* (*action*) to the set but can automatically update the index of the *state* (*action*). Of course, using the *getMiningInputStream* method also the "classic" stream methods can be used to access the states and actions.

The interface *StateActionSet* represents a state set *S* and the action sets *A(s)* for all states *s* of *S*. The method *getStateSet* returns the state set, and the method *getActionSet(State s)* the action set for the specified state *s*. Typically, *StateActionSet* is implemented by environments and contains all admissible states and actions.

Example 12.18 The example assumes countable state and action sets and iterates over all states and corresponding actions of state-action set. We demonstrate different vector iteration and access methods for the state and action sets which are similar to mining input streams.

```
StateActionSet asSet = ... // reference to state-action set
StateSet states = asSet.getStates();
boolean cnSt   = states.isCountable(); // true
for (long i = 0; i < states.getStatesNumber(); i++) {
```

```
  State state  = states.getState(i);
  long is    = state.getIndex();
  ActionSet as = asSet.getActionSet(state);
  boolean cnAs = as.isCountable(); // true
  while ( as.nextAction() ) {
    Action action = as.readAction();
    long ia     = action.getIndex();
    // ... use state (or index is) and action (or index ia) ... //
  }
}
```
■

State-Value Function, Action-Value Function

The classes *StateValueFunction* and *ActionValueFunction* are used for the state-value function *v(s)* and the action-value function *q(s,a)*, respectively.

We start with the state-value function *v(s)*. In this book, we mainly work with discrete state and action sets and use a tabular representation of the state-value function. However, as pointed out in Sect. 6.1.1 and used in Sect. 10.4.2, in many applications, *v(s)* is explicitly represented by a function, in most cases constructed by regression. Thus, we use a *MiningModel* (Sect. 12.1.3) to store the function. This means that in principle all data mining models of XELOPES can be used for *v(s)*, especially the regression models like linear and polynomial regression, regression trees, neural networks, or even sparse grids. For different reasons, *StateValueFunction* does not directly extend *MiningModel* but uses a variable *function* of the class *MiningModel* to store the function.

To handle the important special case of tabular representations, the RL package contains a special mining model – *TableMiningModel* – to store all pairs of argument and function value {*x*, *f(x)*} directly. The class *StateValueTable* extends *TableMiningModel* for state-value functions, i.e., to store all pairs {*s*, *v(s)*}. Depending on whether the number of states is constant or not, it uses an array or a hash table to store the function values. The *StateValueTable* is the default mining model of *StateValueFunction*, i.e., if no other mining model is passed to *StateValueFunction*, this one is used.

We describe the central methods of *StateValueFunction*. The first

```
public double getValue(State state) throws
    MiningException;
```

returns the function value of *state*. Obviously, this method just calls *function.applyModelFunction(state)* to invoke the mining function call.

```
public void setValue(State state, double value) throws
    MiningException;
```

sets the function value at *state* to *value*. It is only supported for *StateValueTable* because in general a mining model cannot be changed directly but is the result of the mining process.

```
public void updateValue(State state, double value) throws
    MiningException;
```

updates the function value. It adds the new value to the current value of the function. This method internally combines *getValue* with *setValue* and thus is also limited to models of *StateValueTable*.

Example 12.19 Consider a simple state-value function with just two states. Using the default *StateValueTable*, it can be written as follows:

```
// Define state-value function (with two states):
StateValueFunction sf = new StateValueFunction(2);
double[] s1 = { 1 };
State st1 = new State(s1, 0); // state index 0
sf.setValue(st1, 0.9);
double[] s2 = { 2 };
State st2 = new State(s2, 1); // state index 1
sf.setValue(st2, 1.9);

// Retrieve function value:
System.out.println(st1 + " -> val1=" + sf.getValue(st1) );
```

■

The class *ActionValueFunction* for action-value functions *q(s, a)* is similar to *StateValueFunction* but uses the state-action pair *(s, a)* instead of a state *s*. The *StateActionVector* class is the internal representative of the state-action pair.

Consequently, *ActionValueFunction* also owns a variable *function* of the class *MiningModel* to store the function values. For discrete problems, in further analogy to *StateValueFunction*, it provides an extended *TableMiningModel*, the *ActionValueTable*, to store all pairs of argument and function value, i.e., {*(s, a)*, *Q(s, a)*}. *ActionValueFunction* contains similar methods to get, set, and update its function values like *StateValueFunction* but with state-action pairs as keys (instead of states only).

Policies

The abstract class *Policy* is the base class of the stochastic policy $\pi(s, a)$ (see Sect. 3.3).

It owns a variable *actionSet* to store the corresponding action set *A(s)*. The values of the actions, called *action values*, can be defined in different ways, most importantly by virtue of an *ActionValueFunction*.

The main methods of *Policy* are:

```
public Action nextAction() throws MiningException;
```

returns next action following the policy,

```
public abstract double probability(Action action) throws
    MiningException;
```

returns the probability *p(s,a)* of an action to be taken.

Different subclasses extend *Policy* for different types of policies. The most important policy class is *GreedyPolicy*, which selects the action(s) of the highest action value (best action). The class *EpsilonGreedyPolicy* extends *GreedyPolicy* for an ε-greedy policy. The class *SoftmaxPolicy* extends *Policy* for a softmax policy.

Example 12.20 We give an example of an ε-greedy policy. The action set contains three possible actions. The action values are defined through an action-value function with the second action having maximum reward. For $\varepsilon = 0.2$, the ε-greedy policy selects the "greedy" action 2 in 80 % of all calls of *nextAction*.

```
// Define action set:
double[] s1 = { 0 };
State st1 = new State(s1, 0); // state index 0
double[] a1 = { 1 };
Action act1 = new Action(a1);
double[] a2 = { 2 };
Action act2 = new Action(a2);
double[] a3 = { 3 };
Action act3 = new Action(a3);
ActionSet as = new ActionSet();
as.setState(st1);
as.addAction(act1);   // action index 0, automatically
  assigned
as.addAction(act2);   // action index 1, automatically
    assigned
as.addAction(act3);   // action index 2, automatically
  assigned

// Define action-value function:
ActionValueFunction qfunction = new ActionValueFunction
  ();
qfunction.setValue(st1, act1, -1);
qfunction.setValue(st1, act2, 8);
qfunction.setValue(st1, act3, 5);

// Define greedy policy:
EpsilonGreedyPolicy egp = new EpsilonGreedyPolicy();
egp.setActionSet(as);
egp.setActionValueFunction(qfunction);
egp.setEpsilon(0.2);
```

```
// Apply policy 10 times:
for (int i = 0; i < 10; i++)
  System.out.println("next action: " + egp.nextAction());
// Result, e.g.:
// next action: action: 2.0 index = 1
// next action: action: 2.0 index = 1
// next action: action: 1.0 index = 0
// next action: action: 2.0 index = 1
// next action: action: 2.0 index = 1
// next action: action: 3.0 index = 2
// next action: action: 2.0 index = 1
...
```

■

Agent, Environment

The central class of the *RL* package is, of course, *RLAgent*. It extends the general *Agent* from Sect. 12.2.1. The generic of *RLAgent* is *Action* because its *apply* and *learnApply* methods return *Action* objects. Unlike the general agent framework of XELOPES, the *RL* package contains a base *Environment* class. *Environment* is an abstract class that extends the *EnvironmentInformation* (Sect. 12.2.1) and implements *StateActionSet*. So the complete interaction of Fig. 3.1 can be modeled by the *RL* package.

RLAgent has an associated settings class *RLAgentSettings* that extends the general *AgentSettings* from Sect. 12.2.1. It stores some basic parameters like the discount rate γ and contains a description of the agent's metadata.

Further, *RLAgent* contains variables *vfunction* for the state-value function, *qfunction* for the action-value function, and *policy* for the policy of the agent (not all must be used). Further, it has a reference to its *Environment*. For the case where the agent knows its environment model (i.e., transition probabilities and -rewards), the variable *envModel* of *RLAgent* can be used. It is of the class *EnvironmentModel* which contains interfaces to access the transition probabilities and -rewards.

The *RL* package also supports simulations in the spirit of Fig. 3.1. The approach was motivated by the RL implementation of Sutton and Santamaria [StSa96]. To this end, the following method is contained in *Environment*:

```
public abstract StateRewardVector step(Action action)
    throws MiningException;
```

This method will be called once by the simulation instance in each step of the simulation. *step* causes the environment to undergo a transition from its current state to a next state dependent on the *action*. The method returns the next state and reward as *StateRewardVector* object. If *action* is null, a new episode starts.

The *learnApply* method of *RLAgent*, inherited from *Agent*, with a *StateRewardVector* object as argument serves as counterpart to the *step* method from the agent side. It takes the next state and reward from the environment and

returns an action which in turn is passed to the environment. If it receives a terminal state from the environment, it returns a null action that causes the environment to start a new episode. In this way, the interaction of Fig. 3.1 is supported by *RLAgent* and its associated *Environment*. The actual simulation is executed by the *Simulation* class which finally presents some statistics.

12.2.2.2 RL Algorithm Packages

DP Package

The dynamic programming algorithms are organized in the *DP* package. It contains an own environment class *DPEnvironment* which extends *Environment* from the RL Core package. The central method of *DPEnvironment* is *getEnvironmentModel* which returns the model object of the environment which is an instance of *EnvironmentModel*.

EnvironmentModel contains two methods *getTransProb* and *getTransRew* to return the transition probabilities $p^a_{ss'}$ and -rewards $r^a_{ss'}$, respectively. Both are modeled by the interface *TransitionFunction* which represents the three-dimensional tensor of transition values from state s to state s' under action a.

The abstract class *DPAgent* extends *RLAgent*, and from its assigned *DPEnvironment*, it takes the model of the environment. Since *DPAgent* learns in offline mode, it has a similar method as *MiningAlgorithm* from the data mining framework to run the learning, *buildModel*, that solves the Bellman equation (3.7). Only after this method has been called, the *policy* of the *DPAgent* can be used. The policy of *DPAgent* is always a greedy policy and hence an instance of *GreedyPolicy* class.

The classes *PolicyIterationAgent* and *ValueIterationAgent* both extend *DPAgent* for the *policy iteration* and *value iteration* algorithms explained in Sect. 3.9.4. They have only few parameters, and in most cases the user has not to care about them.

Example 12.21 We show the example that solves the GridWorld problem of [SB98]. (Notice that the main implementation amount requires the environment class *GridJumpEnvironment* not listed here.)

```
// Create agent settings:
RLAgentSettings agentSettings = new RLAgentSettings();
agentSettings.setInputDataSpecification(metaData);
agentSettings.setGamma(0.9);
agentSettings.verifySettings();

// Get default agent specification from 'agents.xml':
AgentSpecification agentSpecification =
  AgentSpecification.getAgentSpecification( "PolicyItera-
     tionAgent");
```

```
// Set agent parameters:
agentSpecification.setAPValue("maxPolIter", 100);
agentSpecification.setAPValue("maxEvalIter", 200);
agentSpecification.setAPValue("theta", 0.0001);

// Create algorithm object with default values:
DPAgent agent = (DPAgent) agentSpecification.createAgen-
    tInstance();

// Put it all together:
agent.setAgentSettings(agentSettings);
agent.verify();

// Create DP environment:
DPEnvironment env = new GridJumpEnvironment();

// Create and init simulation object:
Simulation sim = new Simulation(agent, env);
sim.init(null); // assigns environment to agent

// Build DP model solving Bellman equation:
System.out.println("TRAINING");
agent.buildModel();
System.out.println( agent.getVfunction() ); // optimal
    state-value function

// Run simulation:
System.out.println("SIMULATION");
int maxStepsPerTrial = 10;
sim.steps(maxStepsPerTrial);
System.out.println("total time [s]: " + sim.getTimeSpent-
    ToRunTrials() );
```
■

MC Package

The Monte Carlo algorithms are organized in the *MC* package. These algorithms are simple, and the package contains basic implementations of MC algorithms like *OnPolicyMCAgent* for the on-policy MC algorithm and *OffPolicyMCAgent* for the off-policy MC algorithm. Consult [SB98] for these algorithms and their parameters, whose names in XELOPES are consistent to the book.

TD Package

The temporal-difference learning algorithms are organized in the *TD* package. Examples are the classes *SarsaAgent* for the Sarsa, on-policy algorithm and *SarsaLambdaAgent* for the Sarsa(λ), on-policy algorithm and *WatkinsQAgent* for

the Watkins Q-learning, off-policy algorithm and *WatkinsQLambdaAgent* for the Watkins $Q(\lambda)$, and off-policy algorithm. Again the names of all parameters are consistent to [SB98].

Example 12.22 We give an example of a modified GridWorld example representing an episodic task (in contrast to our previous GridWorld, which was a continuing task). This new GridWorld has a terminal state after which the episode terminates. Here the reward is -1 for all transitions; thus, we want to reach the terminal state as fast as possible. Like in the previous example, we omit the implementation of the *GridEnvironment* but focus on the solution process.

```
// Create agent settings:
TDAgentSettings agentSettings = new TDAgentSettings();
agentSettings.setInputDataSpecification(metaData);
agentSettings.setGamma(1.0);
agentSettings.setAlpha(0.01);
agentSettings.setLambda(0.9);
agentSettings.verifySettings();

// Get default agent specification from 'agents.xml':
AgentSpecification agentSpecification =
  AgentSpecification.getAgentSpecification("SarsaLambda
      Agent");

// Create algorithm object with default values:
RLAgent agent = (RLAgent) agentSpecification.createAgen-
    tInstance();

// Put it all together:
agent.setAgentSettings(agentSettings);
agent.verify();

// Create environment:
Environment env = new GridEnvironment();

// Create and init simulation object:
Simulation sim = new Simulation(agent, env);
sim.init(null); // assigns environment to agent

// Run simulation:
int numTrials       = 10000;
int maxStepsPerTrial = 100;
sim.setTrialDevisor(1000);
sim.trials(numTrials, maxStepsPerTrial);
System.out.println("total time [s]: " + sim.getTimeSpent
    ToRunTrials());
```

■

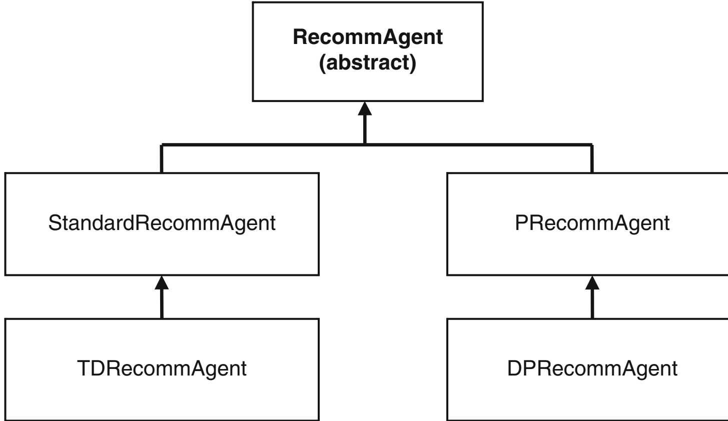


Fig. 12.17 Class hierarchy of recommendation agents

12.2.3 The RL-Based Recommendation Package

The package *Recomm* contains the framework and algorithms for applying RL to recommendation engines following the approach of this book.

The central class *RecommAgent* extends *RLAgent* for RE applications.

Based on the categorization described in Chap. 5, the class *PRecommAgent* extends *RecommAgent* and contains the implementation of the P-Version (Sect. 5.1). It is extended by *DPRecommAgent* for the conditional DP algorithm (Sect. 5.2). In the same way, the class with the historically established name *StandardRecommAgent* extends *RecommAgent* for unconditional TD agents. *TDRecommAgent* extends *StandardRecommAgent* for the conditional TD algorithm. The class hierarchy of the recommendation agents is depicted in Fig. 12.17.

All recommendation agents have a sister settings class as well. The settings classes are organized in a similar hierarchy as their associated agents. In particular, the agent settings class of *RecommAgent* is *RecommAgentSettings* and extends *RLAgentSettings* of the *RLAgent*. For example, *DPRecommAgent* has a settings class *DPRecommAgentSettings*.

All recommendation agents implement the central Agent method

```
public Action learnApply(MiningVector learnApplyVector);
```

It is basically used for learning since we remember that learning for REs is done with a one-step delay. In order to get the recommendations, the *qfunction* object of the *RLAgent* shall be used in combination with a desired policy.

The *learnApply* methods of all recommendation agents require an object of *RecommVector* as argument and work in a stateless mode. This allows mixed learning from multiple sessions. *RecommVector* extends *SarsaVector* and stores the tuple (state, action, reward, next state, next action). However, unlike

SarsaVector, it requires a special action object *ComposedAction* to store the actions. The reason is that we may have multiple recommendations as action.

ComposedAction extends *Action* and has a dual function: it represents a unique, composed action and at the same time stores all the single actions, i.e., the recommended products. At this, each attribute of *ComposedAction* represents a single recommendation. The index of *ComposedAction* stores the selected action, i.e., the "recommendation" that actually has been accepted.

RecommVector owns the method

```
public RecommVector toSelectedRecommVector() throws
    MiningException;
```

which transforms the recommendation vector into a new instance which only contains the selected actions instead of all recommended ones, i.e., the new recommendation vector stores the real transitions of the SARSA tuple as in Fig. 3.7.

Finally, we have the abstract class *RecommEnvironment* that extends *Environment* of the *RL* package and implements *StateBasedActionSet*, an interface reflecting the isomorphism between states and actions (4.1). Further, *RecommEnvironment* contains methods for the mapping between states/actions and their indexes and a special handling for absorbing states. Additionally, *RecommEnvironment* provides methods to access transition probabilities.

Example 12.23 We consider our small test shop of Example 5.4. To make the case more realistic, we further assume that in the course of the sessions, recommendations are displayed. We select the DP Algorithm of Sect. 5.2. In the following, we describe an implementation of a simple recommendation engine by means of the *Recomm* package. We start with the overall execution method:

```
/** The environment object. */
protected RecommEngineEnvironment recoEnv = null;

/** The recommendation agent. */
protected RecommAgent agent = null;

/**
 * Run the recommendation engine example.
 *
 * @throws MiningException error while example is running
 */
public void run() throws MiningException {

  // Create recommendation environment:
  recoEnv = new RecommEngineEnvironment();
  recoEnv.init(null);

  // Create recommendation agent:
  agent = createDPRecommAgent();
```

```
  // Create initial function values:
  createInitialValues();
  showValues();

  // Do the online learning:
  onlineLearning(agent);
  showValues();
}
```

The most complex part is the implementation of *RecommEngineEnvironment* which extends *RecommEnvironment*. Thus, we restrict our attention to the main elements of this class. Especially, we do not show the implementations of all methods of *RecommEnvironment* but just state the internal variables.

```
/**
 * Environment of recommendation engine example. <p>
 *
 * There are six states S = (1,2,3,4,5,6) and in all
 * of these states the same actions may be recommended: A = S.
 */
public class RecommEngineEnvironment extends
   RecommEnvironment
{
 //----------------------------
 // Variables definitions
 //----------------------------
 /** Set of all states. */
 private StateSet states = null;

 /** Array of all action sets. */
 private ActionSet[] actionsets = null;


 //--------- Meta Data States/Actions -------- //
 /** Item ID attribute of states. */
 private CategoricalAttribute itemIDAtt = new Categorica-
    lAttribute("itemIDAtt");

 /** Meta data of states, i.e. item indexes. */
 private MiningDataSpecification metaDataState = null;

 /** Item ID attribute of actions. */
 protected CategoricalAttribute recoIDAtt = new Categori-
    calAttribute("recoIDAtt");

 /** Meta data of actions, i.e. rule indexes. */
 private MiningDataSpecification metaDataAction = null;

 /** Hashtable of step number. */
```

```
private Hashtable<String, Integer> stepHash = new
   Hashtable<String, Integer>();

/** Hashtable of DP. */
private Hashtable<String, double[]> dpHash = new
   Hashtable<String, double[]>();

// -----------------------------
// Constructor
// -----------------------------
/**
 * Empty constructor.
 */
public RecommEngineEnvironment(){
}

// -----------------------------
// Initialization
// -----------------------------
/**
 * Init.
 *
 * @param args array of strings
 * @exception MiningException general exception
 */
public void init(String[] args) throws MiningException {

 // Create state meta data:
 metaDataState = new MiningDataSpecification("itemID");
 int nstates = 6;
 for (int i = 0; i < nstates; i++)
  itemIDAtt.addCategory( new Category( String.valueOf(i
     +1) ) );
 metaDataState.addMiningAttribute(itemIDAtt);

 // Create metadata for actions:
 metaDataAction = new MiningDataSpecification
    ("actions");
 int nactions = nstates;
 for (int i = 0; i < nactions; i++)
  recoIDAtt.addCategory( new Category( String.valueOf(i
     +1) ) );
 metaDataAction.addMiningAttribute(recoIDAtt);

 // Create states and action arrays:
 MiningStoredData msd = new MiningStoredData();
 msd.setMetaData(metaDataState);
```

```
  states = new StateSet(msd);
  states.setStaticStates(true);
  states.setStaticActionSets(false);
  states.setStaticActionSetLength(false);
  actionsets = new ActionSet[nstates];

  // Fill data:
  for (int i = 0; i < nstates; i++) {
   // New state:
   double[] values = {i};
   State state = new State(values);
   state.setMetaData(metaDataState);
   states.addState(state);

   // New action set:
   ActionSet actionSet = new ActionSet();
   actionSet.setState(state);
   actionSet.setStaticActions(false);

   actionsets[i] = actionSet;
  }
}

// ----------------------------
// Methods to access states and their actions
// ----------------------------
/**
 * Returns state set, i.e. all states available in the
   environment.
 *
 * @return state set of the environment
 * @exception MiningException state set access error
 */
public StateSet getStates() throws MiningException {

  return states;
 }

/**
 * Returns action set for a specified state.
 *
 * @param state specified state
 * @return action set of specified state
 * @throws MiningException action set access error
 */
public ActionSet getActionSet(State state) throws
   MiningException {
```

```
 return actionsets[ (int) state.getIndex() ];
}

/**
 * Returns state specified by its name.
 *
 * @param stateStr the name of the state
 * @return the state
 * @throws MiningException
 */
public State getState(String stateStr) throws
    MiningException {

  //... Implementation ...//
 }
/**
 * Returns the action specified by the state and action
   names.
 *
 * @param stateStr the state name
 * @param actionStr the action name
 * @return the action
 * @throws MiningException
 */
public Action getAction(String stateStr, String
    actionStr)
 throws MiningException {

 //... Implementation ...//
}

/**
 * Adds action specified by the state and action name.
 *
 * @param stateStr the state name
 * @param actionStr the action name
 * @return the action added
 * @throws MiningException
 */
public Action addAction(String stateStr, String
    actionStr)
 throws MiningException {

 //... Implementation ...//
}
```

```
// ----------------------------
// Methods to associate actions with states
// ----------------------------
/**
 * Returns state associated to an action.
 *
 * @param state the state of the action set
 * @param action an action applied to the state
 * @return the state associated to the action, null if inva-
   lid action
 * @exception MiningException state set access error
 */
public State getStateFromAction(State state, Action
    action)
 throws MiningException {

 //... Implementation ...//
}
/**
 * Returns state-action pairs associated to an action
   state.<p>
 *
 * Inverse method to getStateFromAction.
 *
 * @param state the action state
 * @return array of all state-action pairs with actions
   associated to the state,
 * null if not found
 * @exception MiningException state set access error
 */
public StateActionVector[] getActionsFromState(State
    state)
 throws MiningException {

 throw new MiningException("not supported");
}

/**
 * Returns all state-action pairs from given initial and
   target
 * set of states.
 *
 * @param initStates array of initial states
 * @param tarStates array of target (i.e. action) states
 * @return array of StateActionVectors
 * @throws MiningException
```

```
 */
public StateActionVector[] getActionsFromStates(State
    [] initStates,
 State[] tarStates) throws MiningException {

 throw new MiningException("not supported");
}

/**
 * Adds a new action for an initial and target state to
 * the action set.
 *
 * @param initState the initial state
 * @param tarState the target state
 * @return new action from initial to target state
 * @throws MiningException
 */
public Action addActionForStates(State initState, State
    tarState)
 throws MiningException {

 //... Implementation ...//
}

// ----------------------------
// Step numbers and transition probabilities
// ----------------------------
/**
 * Returns probability step number of (state, action) pair
   for DP version
 * for conditional and unconditional case.
 *
 * @param state the state
 * @param action the action
 * @param cond conditional probability (else
     unconditional)
 * @return returns step number at specified (state, action)
   pair
 * @throws MiningException
 */
public int getStepNumberP(State state, Action action,
    boolean cond)
 throws MiningException {

 //... Implementation ...//
}
```

```
/**
 * Sets probability step number of (state, action) pair for
   DP version
 * for conditional and unconditional case.
 *
 * @param state the state
 * @param action the action
 * @param the step number at specified (state, action) pair
 * @param cond conditional probability (else
   unconditional)
 * @throws MiningException
 */
public void setStepNumberP(State state, Action action,
   int stepNumber,
 boolean cond) throws MiningException {

 //... Implementation ...//
}

/**
 * Returns transition probability for DP version of (state,
   action) pair
 * for conditional and unconditional case.
 *
 * @param state the state
 * @param action the action
 * @param cond conditional probability (else
   unconditional)
 * @return returns transition probability at specified
   (state, action) pair
 * @throws MiningException
 */
public double getTransP(State state, Action action, bool-
   ean cond)
 throws MiningException {

 //... Implementation ...//
}

/**
 * Sets transition probability for DP version of (state,
   action) pair
 * for conditional and unconditional case.
 *
 * @param state the state
 * @param action the action
```

```
  *   @param   cond   conditional   probability   (else
      unconditional)
  * @param transP the transition probability at specified
    (state, action) pair
  * @throws MiningException
  */
 public void setTransP(State state, Action action, boolean
    cond, double transP)
  throws MiningException {

  //... Implementation ...//
 }

 // ----------------------------
 // Other methods
 // ----------------------------
 ...
}
```

Thus, we use the *StateSet* and *ActionSet* classes of the *RL Core* package to store the products and rules. Of course, in real recommendation engine implementations, this storage is implemented in a much more sophisticated way supporting large rule sets based on hash mappings. The transition probabilities $p_n^{(a)}(S, S')$ including their step numbers n are stored by the hash tables *dpHash* and *stepHash*, respectively. Again, we use a very simple implementation here which is also not very fast.

After we have designed the recommendation environment, we need to create the recommendation agent.

```
 /**
  * Create DP recommendation agent.
  *
  * @return the agent object
  * @throws MiningException
  */
 private   RecommAgent   createDPRecommAgent()   throws
    MiningException {

  // Create settingsobject:
  DPRecommAgentSettings agentSettings = new DPRecommAgen-
     tSettings();
  agentSettings.setInputDataSpecification(new  MiningDa-
     taSpecification("dummy"));
  agentSettings.setDpVersionSubtype(PRecommAgen-
     tSettings.P_VERSION_SUBTYPE_FULL);
  agentSettings.setAlphaType(      RecommAgentSettings.
     RL_STEP_SIZE_FIXED );
  agentSettings.setAlpha( 0.1 );
```

```
 agentSettings.setBetaType(      RecommAgentSettings.
    RL_STEP_SIZE_MEAN );
 agentSettings.setBeta( 1.0 );
 agentSettings.setControlGroupLearning( false );
 agentSettings.setScaleCuDP( 1.1 );
 agentSettings.setMinProbCountDP( 1 );
 agentSettings.verifySettings();

 // Get agent specification:
 String agentName = "DPRecommAgent";
 AgentSpecification agentSpecification = getAgentSpeci-
    fication(agentName);
 if( agentSpecification == null )
  throw new MiningException( "Can't find application " +
     agentName );

 // Create agent object:
 RecommAgent agent = (RecommAgent) agentSpecification.
    createAgentInstance();
 // Create action-value function:
 ActionValueFunction actionValueFct = new ActionValue-
    Function(recoEnv);

 // Put it all together:
 agent.setAgentSettings(agentSettings);
 agent.setEnv(recoEnv);
 agent.setQfunction(actionValueFct);
 agent.setInnerAgent( createInnerAgent() );
 agent.verify();

 return agent;
}
```

After we have created the agent, we can assign initial values to its action-value functions. In reality, this means that we load the previous rule base after the recommendation engine has been restarted and continue the online learning.

Now we turn to the online learning and demonstrate the first three steps and the last step of the first session $1 \to 5^* \to 4 \to \ldots \to 6^*$ and the transition to the next session starting with product 6.

```
/**
 * Do the online learning.
 *
 * @param agent the recommendation agent
 * @throws MiningException
 */
private void onlineLearning(RecommAgent agent) throws
   MiningException {
```

```
  // FIRST SESSION:
  // Step 1 (product 1 clicked):
  State state = recoEnv.getState("1");
  int[] recs  = recos(state, 1);
  showRecs(state, recs);
  learn(state, recs, 1.0);

  // Step 2 (product 5 clicked and added to basket):
  state = recoEnv.getState("5");
  recs  = recos(state, -1);
  showRecs(state, recs);
  learn(state, recs, 1.0 + 15.0);

  // Step 3 (product 4 clicked):
  state = recoEnv.getState("4");
  recs  = recos(state, -1);
  showRecs(state, recs);
  learn(state, recs, 1.0);

  // ... further steps ... //

  // Step 12 (product 6 clicked and added to basket):
  state = recoEnv.getState("6");
  recs  = recos(state, -1);
  showRecs(state, recs);
  learn(state, recs, 1.0 + 4.5);

  // Move to absorbing node in order to terminate first
     session:
  state = recoEnv.getState("_a_");
  learn(state, recs, 0.0);

  // SECOND SESSION:
  // Step 1 (product 6 clicked):
  state = recoEnv.getState("6");
  recs  = recos(state, 1);
  showRecs(state, recs);
  learn(state, recs, 1.0);

  // ... further steps and sessions ... //
}
```

The method *recos* calculates the recommendations using an ε-greedy policy:

```
/**
 * Return recommendations for specified state.
 *
 * @param state the state
```

```
  * @param maxNumberOfRecommendations the number of recom-
    mendations, -1 if all
  * @return the recommendation indexes
  * @throws MiningException
  */
private int[] recos(State state, int
    maxNumberOfRecommendations)
  throws MiningException {

  // Create policy:
  Policy policy = new EpsilonGreedyPolicy( 0.5 );
  ActionSet actionSet = recoEnv.getActionSet(state);
  policy.setActionSet(actionSet);
  policy.setActionValueFunction( agent.getQfunction() );
  policy.setDifferentActions(true);

  // Call policy:
  Vector<Integer> recItems = new Vector<Integer>();
  int nact = (int) actionSet.getActionsNumber();
  for (int i = 0; i < nact; i++) {
    if (recItems.size() == maxNumberOfRecommendations)
      break;
    Action action = policy.nextAction();
    recItems.addElement((int)action.getIndex());
  }
  int[] recs = new int[ recItems.size() ];
  for (int i = 0; i < recItems.size(); i++)
    recs[i] = recItems.elementAt(i);

  return recs;
}
```

The method *showRecs* displays the current recommendations. Finally, the *learn* method takes the current state and recommendations and calls the *learnApply* method of the agent for the *previous* step.

```
// Private session data:
private State state = null;
private int[] recs = null;
private int selRec = -1;
private State nextState = null;
private int[] nextRecs = null;
private int nextSelRec = -1;
/**
  * Learns from previous step.
  *
  * @param cstate the current state
```

```
 * @param crecs the current recommendation indexes
 * @param crewardValue the current reward value
 * @throws MiningException
 */
private void learn(State cstate, int[] crecs, double
    crewardValue)
 throws MiningException {

 String cStateStr = (String) cstate.getValueCategory(0).
     getValue();

 // Create sample vector:
 if (nextState != null) {
  String nextStateStr = (String) nextState.getValue-
      Category(0).getValue();
  Action action = recoEnv.getAction(nextStateStr,
      cStateStr);
  if (action == null) {
   // ... create new rule ... ///
  }
 nextSelRec = (int) action.getIndex();
 }
if (state != null) {
 Action action = new ComposedAction(recs, selRec,
     recoEnv.recoIDAtt);
 Reward reward = new Reward(crewardValue);
 Action nextAction =
  new ComposedAction(nextRecs, nextSelRec, recoEnv.
      recoIDAtt);
 RecommVector recommVec =
  new RecommVector(agent, state, action, reward,
      nextState, nextAction);
 agent.learnApply(recommVec);
}

// Update values:
boolean absorbing = cStateStr.equals("_a_");
 if (absorbing) {
  state  = null;
  recs   = null;
  nextSelRec = -1;
 }
 else {
  state  = nextState;
  recs   = nextRecs;
 }
```

```
  selRec = nextSelRec;
  nextState = cstate;
  nextRecs  = crecs;
 }
```

12.3 Application Example of XELOPES: The prudsys RDE

We conclude this chapter by a practical example for the application of the XELOPES library in recommendation engines. The prudsys Realtime Decisioning Engine (RDE) is a realtime analytics system developed by prudsys AG, which follows the principles of realtime analytics described in this book. Its range of functions extends well beyond that of a standard recommendation engine.

It comprises the following six modules:

1. *RDE / Recommendations*: realtime recommendation engine
2. *RDE / Newsletter*: realtime newsletter personalization
3. *RDE / Pricing*: realtime price optimization
4. *RDE / Assortment Planning*: realtime planning
5. *RDE / Scoring*: realtime scoring
6. *RDE / Search*: realtime search

This book concentrates primarily on the way in which the *RDE / Recommendations* module works, specifically on the adaptive learning, and – first of all – on the reinforcement learning aspect. However, the module also includes many other algorithms, such as basket and sequence analysis, collaborative filtering, item-to-item collaborative filtering, singular value decomposition, and tensor factorization.

It is interesting to note that the concept of realtime learning has now been applied to many new functions across the other modules (so the "realtime" prefix is more than just a marketing ploy). For example, the *RDE / Scoring* module allows realtime scoring as described in Chap. 7. The *RDE / Assortment Planning* module updates its planning model continuously throughout the day, allowing delivery to be brought forward by a day, for example. Probably the most impressive module, *RDE / Pricing*, which carries out dynamic price optimization, is one of the key inventions of the prudsys AG. It varies retail product prices in real time (or at periodic intervals) according to user behavior and demand, with the aim of maximizing profits. Reinforcement learning is used here, too. For more details, see the separate pricing module white paper [Lip11].

The switch to real time not only offers an entirely new standard of quality for analytics (or actions) but also opens up entirely new business scenarios. Welcome to the RDE realtime world!

The layer model on which the prudsys RDE is based is shown in Fig. 12.18.

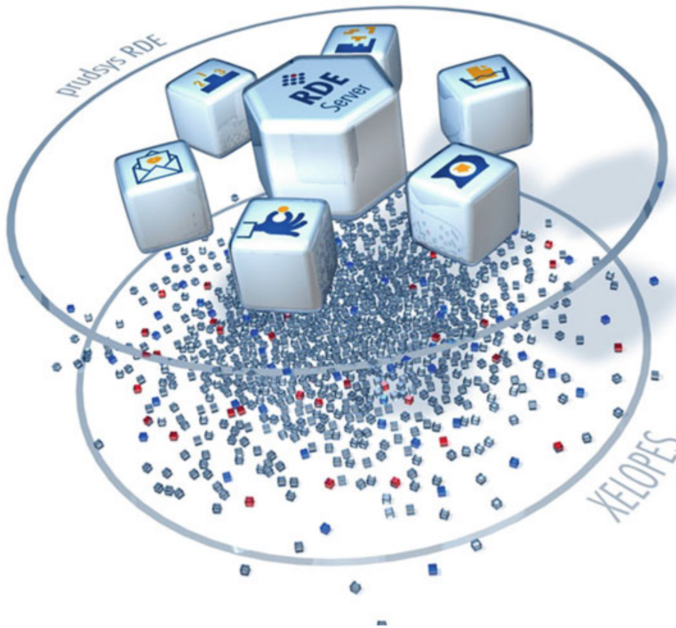


Fig. 12.18 RDE server with modules and XELOPES

Note the split between analysis algorithms and business logic. All prudsys analysis algorithms are provided through the XELOPES library.

All data is specified by means of the flexible CWM, allowing the number of agent methods to be kept to a minimum. The realtime analytics algorithms are then divided into packages such as recommendations, pricing, and planning, all of which ultimately derive from the agent class.

The RDE server then runs on the XELOPES library as a container which implements the business logic. It is possible for multiple instances of recommendation engines to run on a single application server; distributed RE applications are supported in particular. Configuration and administration are carried out via an easy-to-use GUI. The RDE offers numerous options for defining business constraints and delivers realtime statistics, particularly for A/B testing.

12.4 Summary

The aim of this chapter was to provide some ideas of implementing the adaptive algorithms described in this book based on the XELOPES library for BI. We started with the very abstract CWM standard and then considered its application to data

mining. Next, we moved to realtime data mining where the central idea was the introduction of agents. The agent framework was further specified for reinforcement learning, and based on RL we next proposed a framework for adaptive recommendation engines. At the end, we briefly discussed the application of XELOPES for real recommendation engines.

Chapter 13
Last Words: Conclusion

Abstract We first discuss the requirements of a modern data mining system and show that the approach presented in this book fulfills most of them. However, the full realization of this approach is often thwarted by principal problems in the development of the required mathematical instruments. Especially, most of the computational methods developed by mathematicians over the last centuries are designed for engineering problems. We stress the differences to the requirements for data analysis problems and encourage the development of appropriate frameworks. Especially, control theory should play an important role here.

We will conclude by briefly summarizing the approaches to developing modern recommendation engines described in this book and breaking them down into seven general requirements of a modern data mining system:

1. *Autonomous operation*: System learns automatically, no manual operation required.
2. *Realtime operation*: System learns and decides in real time.
3. *Integration into applications*: System is embedded directly in applications.
4. *Control problem approach*: System learns through interaction, "cybernetic" thinking.
5. *Operator description*: Mathematical formulation via operator equations.
6. *Hierarchical approach*: System uses hierarchical methods and architecture.
7. *Distributed operating principle*: System operates on a decentralized, distributed basis.

The requirements are interdependent to some extent of course: realtime operation, for example, requires the ability to work autonomously. And they are not necessarily all indisputable. But they illustrate key requirements and fundamental trends, the use of which will ultimately lead to a new quality of data mining.

Most current data mining systems meet almost none of these requirements. Rather than operating autonomously, they have to be operated manually, by

A. Paprotny and M. Thess, *Realtime Data Mining: Self-Learning Techniques for Recommendation Engines*, Applied and Numerical Harmonic Analysis,
DOI 10.1007/978-3-319-01321-3_13, © Springer International Publishing Switzerland 2013

statisticians or at least by experts. Rather than learning in real time, they learn from historical data, much of which is stored in dinosaur applications like data warehouses. Rather than being integrated directly into applications, they run as separate programs with unwieldy GUIs. Rather than understanding the problem as an interaction of analysis and decision, most of them disregard the decision aspect completely and concentrate entirely on analysis, as a result of which the question of their interaction never even arises. Rather than formulating the problem in the mathematically conventional operator syntax (e.g., as a differential equation), many of them still use the terminology of neural networks, genetic algorithms, etc. Rather than breaking down the solution hierarchically, both in terms of content and mathematically, the often immense problems are approached as a single, gigantic, data block, in the hope that the method will somehow cut its way through the mass of data. Rather than carrying out local analyses on a distributed basis and only joining the results ("taking software to the data"), all data has to be centralized, after which it is stored in an inflexible and incomplete form in massive data warehouses ("taking data to the software").

By contrast, in this book, we made an attempt at devising approaches that satisfy the above requirements, though not entirely (in particular, we hardly addressed the last requirement concerning distributedness in its most visionary form) but in essence at least. This is what we have tried to illustrate in this book. As such, it is the trailblazer for a completely new way of thinking in data mining.

Many of the ideas presented in this book, especially that of reinforcement learning, have originated in artificial intelligence research. Being mathematical computer scientists, we have been aspiring to draw a crisp distinction between mathematical modeling of a real-world problem on one hand and devising computational methods for solving the emerging equations on the other hand. This course of action is still somewhat uncommon in the data mining and artificial intelligence community, where algorithms are often conceived as models of real-world agents solving real-world problems rather than methods to solve mathematical problems that, in turn, represent real-world problems. We believe that our mathematical approach provides insights as to which technical assumptions these AI methods are actually based on, under which circumstance their success can be guaranteed, and what their limitations are. Furthermore, it enables to figure novel and more efficient implementations that facilitate dealing with large and high-dimensional data sets and enable realtime operation.

Nevertheless, our approach has shortcomings of its own. Many of the computational methods devised in this book, especially multigrid methods and tensor approximation for reinforcement learning, are based on, or, at least, inspired by frameworks for problems arising in discretization and numerical treatment of differential equations. The latter setting may be characterized as follows:

1. *Continuity*: A differential equation is a continuous model of a physical phenomenon.
2. *Physical interpretability*: Mathematical structure arises for physical reasons.
3. *A priori model*: The parameters of the model are available beforehand.

4. *Sparsity*: Discretization schemes are designed in such a way that most of the coefficients of the discrete equations vanish.
5. *Patterns*: The nonzero coefficients are arranged in a way governed by a predictable pattern.
6. *Structure inheritance*: The discrete system inherits mathematical structure from the continuous one. (For example, an elliptic differential equation may be discretized in such a way that the discrete system is also elliptic, i.e., positive definite.)

Sadly enough, except for sparsity, none of the above holds with respect to the recommendation setting. First of all, there is no underlying continuous structure at all, let alone a physical interpretation. The real-world phenomenon and the equation of its model are genuinely discrete. Apart from this, most parameters of the model, as, e.g., the transition probabilities, are not known in advance, but have to be figured out empirically as one goes along. Furthermore, although the coefficient matrices of the Bellman equations arising in recommendations are typically sparse, the nonzero coefficients are distributed basically at random. Finally, there is no structure to be inherited from a continuous model. For example, there is no reason to assume positive definiteness, a crucial prerequisite of convergence results on many multigrid-related methods, let alone that the considered Markov chains be reversible.

All in all, this supports the impression we have gained through years of research and practical experience in the field of mathematical data analysis: on one hand, it seems that many approaches from the twentieth-century mathematics are suitable to be carried over to problems arising in data analysis. On the other hand, the underlying mathematical theory is designed for settings, the structure of which differs in many essential respects from that encountered in data analysis-related problems. Our colleague Mijail Guillemard recently put this as follows:

> In hindsight, I recognize that in my years as a PhD-student, I wasted a lot of time immersing myself in mathematical theories that are not quite suited for the problems the solution of which I sought after.

Again, in hindsight, it does not come as a surprise that a major part of state-of-the-art mathematical theory is hardly applicable to data analysis. After all, until lately, the development of mathematics was predominantly driven by problems encountered in science and engineering, to which data analysis was added only recently. As a consequence, data analysis requires outright novel extensions and generalizations of classical mathematical theories, the development of which will certainly keep researches occupied for decades to come.

Let us conclude this outlook by a brief philosophical remark: historically, computers and computational mathematics were primarily designed for solving numerical problems related to differential equations. The development which took place in the course of the following decades, however, is an instance of what the American biologist Stephen Jay Gould refers to as an *exaptation:* computers were gradually transformed into general-purpose information processing, storage, and communication systems and began to figure increasingly in the organization of

industrialized societies, to such an extent that nowadays, we are said to live in the information age. Specifically, this refers to both the significance of information (as opposed to material commodities) as well as the quantity thereof produced, stored, and communicated every day, which has increased exponentially in the course of the (say) last five decades. This development has given rise to the outright novel problems of information science and data analysis such as recommendation. Ironically, to solve these problems, we need to resort to the very devises which gave rise to them in the first place.

"Study cybernetics!" 35 years on, Viktor Pekelis' vision could still become a reality. In the light of the new popularity of realtime analytics, cybernetic principles and approaches should see a renaissance – not in the all-encompassing sense of the 1950s, of course, but in a stricter, mathematical context.

References

[AR02] Andre, D., Russel, S.J.: State abstraction for programmable reinforcement learning agents. In: Proceedings of the National Conference on Artificial Intelligence, pp. 119–125. Edmonton, Alberta, Canada (2002)

[ADT95] Arge, A., Daehlen, M., Tveito, A.: Approximation of scattered data using smooth grid functions. J. Comput. Appl. Math. **59**, 191–205 (1995)

[Bakh66] Bakhvalov, N.S.: On the convergence of a relaxation method with natural constraints on the elliptic operator. USSR Comp. Math. Math. Phys. **6**, 101–113 (1966)

[BC89] Bertsekas, D.P., Castanon, D.A.: Adaptive aggregation methods for infinite horizon dynamic programming. IEEE Trans. Automat. Contr. **34**(6), 589–598 (1998)

[Beb08] Bebendorf, M.: Hierarchical Matrices: A Means to Efficiently Solve Elliptic Boundary Value Problems. Lecture Notes in Computational Science and Engineering (LNCSE), vol. 63. Springer-Verlag, Berlin Heidelberg (2008)

[Beer59] Beer, S.: Cybernetics and Management. English Universities Press, London (1959)

[BMR82] Brandt, A., McCormick S.F., Ruge J.W.: Algebraic multigrid (AMG) for automatic multigrid solution with application in geodetic computations. Technical Report CO POB 1852, Institute Computational Studies State University (1982)

[BNS78] Bunch, J.R., Nielsen, C.P., Sorensen, D.C.: Rank-one modification of the symmetric eigenproblem. Numer. Math. **31**, 31–48 (1978)

[BGGK12] Bokanowski, O., Garcke, J., Griebel, M., Klompmaker, I.: An adaptive sparse grid semi-lagrangian scheme for first order Hamilton-Jacobi Bellman equations, submitted to J. Sci. Comput., also available as INS Preprint No. 1207 (2012)

[BPX90] Bramble, J., Pasciak, J., Xu, J.: Parallel multilevel preconditioners. Math. Comput. **55**, 1–22 (1990)

[Bra77] Brandt, A.: Multi-level adaptive solutions to boundary-value problems. Math. Comput. **31**, 333–390 (1977)

[Bra02] Brand, M.E.: Incremental singular value decomposition of uncertain data with missing values. In: Heyden, A., Sparr, G., Nielsen, M., Johansen, P. (eds.) Computer Vision ECCV. Lecture Notes in Computer Science, vol. 2350, pp. 707–720. Springer, Berlin/Heidelberg (2002)

[Bra03] Brand, M.E.: Fast online svd revisions for lightweight recommender systems. In: SIAM International Conference on Data Mining (SDM). San Francisco, California, USA (2003)

[Bra06] Brand, M.E.: Fast low-rank modifications of the thin singular value decomposition. Linear Algebra Appl. **415**, 20–30 (2006)

[BS10] Bhasker, B., Srikumar, K.: Recommender Systems in E-Commerce. Tata McGraw-Hill Education. Noida, UF, India (2010)

A. Paprotny and M. Thess, *Realtime Data Mining: Self-Learning Techniques for Recommendation Engines*, Applied and Numerical Harmonic Analysis,
DOI 10.1007/978-3-319-01321-3, © Springer International Publishing Switzerland 2013

[BT96] Bertsekas, D.P., Tsitsiklis, J.N.: Neuro-Dynamic Programming. Athena Scientific, Belmont (1996)

[Bun92] Bungartz, H.-J: Dünne Gitter und deren Anwendung bei der adaptiven Lösung der dreidimensionalen Poisson-Gleichung (in German). Dissertation, TU München (1992)

[BGRZ94] Bungartz, H.-J., Griebel, M., Röschke, D., Zenger, C.: Pointwise convergence of the combination technique for Laplace's equation. East–west, J. Numer. Math. **2**, 21–45 (1994)

[CR08] Cande's, E.J., Recht, B.: Exact matrix completion via convex optimization. Found. Comput. Math. **9**, 717–772 (2008)

[CRT06] Candès, E.J., Romberg, J., Tao, T.: Stable signal recovery from incomplete and inaccurate measurements. Comm. Pure Appl. Math. **59**, 1207–1223 (2006)

[CS09] Chen, J., Saad, Y.: Lanczos vectors versus singular vectors for effective dimension reduction. IEEE Trans. Knowl. Data Eng. **21**, 1091–1103 (2009)

[CT10] Candès, E.J., Tao, T.: The power of convex relaxation: near-optimal matrix completion. IEEE Trans. Info. Theor. **56**(5), 2053–2080 (2010)

[Dau92] Daubechies, I.: Ten Lectures on Wavelets. Society for Industrial and Applied Mathematics, Philadelphia (1992)

[DHS05] Ding, C., He, X., Simon, H.D.: On the equivalence of nonnegative matrix factorization and spectral clustering. Proc. SIAM Data Mining Conf. **4**, 606–610 (2005)

[Diet98] Dietterich, T.G.: Approximate statistical tests for comparing supervised classification learning algorithms. Neural Comput. **10**(7), 1895–1924 (1998)

[Diet00] Dietterich, T.G.: Hierarchical reinforcement learning with the MAXQ value function decomposition. J. Artif. Intell. Res. **13**, 227–303 (2000)

[DLDMV00] De Lathauwer, L., De Moor, B., Vandewalle, J.: A multilinear singular value decomposition. SIAM J. Matrix Anal. Appl. **21**(4), 1253–1278 (2000)

[DLJ10] Ding, C., Li, T., Jordan, M.I.: Convex and semi-nonnegative matrix factorizations. IEEE Trans. Pattern Anal. Mach. Intell. **32**(1), 45–55 (2010)

[DMC11] http://www.data-mining-cup.de/en/review/dmc-2011/

[Don06] Donoho, D.L.: Compressed sensing. IEEE Trans. Info.Theor. **52**(4), 1289–1306 (2006)

[DSL08] De Silva, V., Lim, L.-H.: Tensor rank and the ill-posedness of the best low-rank approximation problem. SIAM J. Matrix Anal. Appl. **30**(3), 1084–1127 (2008)

[EPP00] Evgeniou, T., Pontil, M., Poggio, T.: Regularization networks and support vector machines. Adv. Comput. Math. **13**, 1–50 (2000)

[Fab9] Faber, G.: Über stetige Funktionen (in German). Math. Annal. **66**, 81–94 (1909)

[Fed64] Fedorenko, R.P.: The speed of convergence of one iterative process. USSR Comput. Math. Math. Phys. **4**, 227–235 (1964)

[FM 01] Fung, G., Mangasarian, O.L.: Proximity support vector machine classifiers. In: Provost, F., Srikant, R. (eds.) Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 77–86 (2001)

[Fun06] Funk, S.: Netflix update: try this at home. http://sifter.org/~simon/journal/20061211.html

[Gar06] Garcke J.: Regression with the optimized combination technique. In: ICML '06: Proceedings of the 23th International Conference on Machine learning, pp. 321–328. ACM Press, New York (2006)

[Gar11] Garcke, J.: A dimension adaptive sparse grid combination technique for machine learning. In: Read, W., Larson, J.W., Roberts, A.J. (eds.) Proceedings of the 13th Biennial Computational Techniques and Application Conference, CTAC-2006, vol. 48 of ANZIAM J., pp. C725–C740 (2007)

[Gar12b] Garcke, J.: A dimension adaptive combination technique using localized adaptation criteria. In: Bock, H.G., Hoang, X.P., Rannacher, R., Schlöder, J.P. (eds.) Modeling, Simulation and Optimization of Complex Processes, pp. 115–125. Springer, Dordrecht (2012)

[GG01a] Garcke, J., Griebel, M.: Data Mining with sparse grids using simplicial basis functions. In: Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. San Francisco, California, USA (2001)

[GGT01] Garcke, J., Griebel, M., Thess, M.: Data mining with sparse grids. Computing **67**(3), 225–253 (2001)

[Gea30] Geary, R.C.: The frequency distribution of the quotient of two normal variates. J. R. Stat. Soc. **93**(3), 442–446 (1930)

[GG98] Gerstner, T., Griebel, M.: Numerical integration using sparse grids. Numer. Algorithms **18**, 209–232 (1998)

[Gir98] Girosi, F.: An equivalence between sparse approximation and support vector machines. Neural Comput. **10**, 1455–1480 (1998)

[GJP93] Girosi, F., Jones, M., Poggio, T.: Priors, stabilizers and basis functions: from regularization to radial, tensor and additive splines. AI Memo No. 1430. Artificial Intelligence Laboratory, MIT (1993)

[GJP95] Girosi, F., Jones, M., Poggio, T.: Regularization theory and neural networks architectures. Neural Comput. **7**, 219–265 (1995)

[GNBT92] Goldberg, D., Nichols, D., Oki, B.M., Terry, D.: Using collaborative filtering to weave an information tapestry. Commun. ACM **35**(12), 61–70 (1992). Special issue on information filtering

[GR04] Golovin, N., Rahm, E.: Reinforcement Learning Architecture for Web Recommendations. Proc. ITCC2004, IEEE (2004)

[GV81] Glushkov, V.M., Vallakh, V.I.: What is OGAS? (in Russian). Kvant, Nauka (1981)

[GVL96] Goloub, G.H., Van Loan, C.F.: Matrix Computations, 3rd edn. The Johns Hopkins University Press, Baltimore/London (1996)

[Gra10] Grasedyck, L.: Hierarchical singular value decomposition of tensors. SIAM J. Matrix Anal. Appl. **31**, 2029–2054 (2010)

[GH03] Grasedyck, L., Hackbusch, W.: Construction and arithmetics of H-matrices. Computing **70**, 295–334 (2003)

[Gri92] Griebel, M.: The combination technique for the sparse grid solution of PDEs on multiprocessor machines. Parallel Process. Lett. **2**(1), 61–70 (1992)

[GSZ92] Griebel, M., Schneider, M., Zenger, C.: A combination technique for the solution of sparse grid problems. In: de Groen, P., Beauwens, R. (eds.) Iterative Methods in Linear Algebra, pp. 263–281. Elsevier, Amsterdam (1992)

[GE94] Gu, M., Eisenstat, S.C.: A stable and efficient algorithm for the rank-one modification of the symmetric eigenproblem. SIAM J. Matrix Anal. Appl. **15**, 1266–1276 (1994)

[Ha85] Hackbusch, W.: Multigrid Methods and Applications. Springer, Berlin (1985)

[Ha99] Hackbusch, W.: A sparse matrix arithmetic based on H-matrices. Part I: introduction to H-matrices. Computing **62**, 89–108 (1999)

[HK00] Hackbusch, W., Khoromskij, B.N.: A sparse H-matrix arithmetic. Part II: application to multi-dimensional problems. Computing **64**, 21–47 (2000)

[HK09] Hackbusch, W., Kühn, S.: A new scheme for the tensor representation. J. Fourier Anal. Appl. **15**, 706–722 (2009)

[Heg03] Hegland, M.: Adaptive sparse grids. In: Burrage, K., Sidje, R.B. (eds.) Proceedings of 10th Computational Techniques and Application Conference CTAC-2001, vol. 44, pp. C335–C353. Brisbane, Australia (2003)

[Hol92] Holland, J.H.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. The MIT Press, Cambridge (1992)

[HJ85] Horn, R.A., Johnson, C.R.: Matrix Analysis. Cambridge University Press, New York (1985)

[HL92] Hoschek, J., Lasser, D.: Grundlagen der geometrischen Datenverarbeitung. Teubner, Stuttgart (1992). Chapter 9 (in German)

[In96] Inmon, W.H.: Building the Data Warehouse, 2nd edn. Wiley, New York (1996)
[JDM] http://jcp.org/en/jsr/detail?id=73
[JOLAP] http://jcp.org/en/jsr/detail?id=69
[JZFF10] Jannach, D., Zanker, M., Felfernig, A., Friedrich, G.: Recommender Systems: An Introduction. Cambridge University Press, Cambridge (2010)
[KABO10] Karatzoglou, A., Amatriain, X., Baltrunas, L., Oliver, N.: Multiverse recommendation: n-dimensional tensor factorization for context-aware collaborative filtering. In: Proceedings of the Fourth ACM Conference on Recommender Systems, RecSys '10, pp. 79–86. ACM, New York (2010)
[KB09] Kolda, T.G., Bader, B.W.: Tensor decompositions and applications. SIAM Rev. **51**(3), 455–500 (2009)
[Kim96] Kimball, R.: The Data Warehouse Toolkit. Wiley, New York (1996)
[Kuh60] Kuhn, H.W.: Some combinatorial lemmas in topology. IBM J. Res. Develop. **4**, 518–524 (1960)
[Lip11] Lippert, J.: Dynamic Price Optimization in Retail. Whitepaper, prudsys AG (2011)
[LSY03] Linden, G., Smith, B., York, J.: Amazon.com recommendations: item-to-item collaborative filtering. IEEE Internet Comput. (2003)
[Mah10] Mahmood, T.: Learning User-Adapted Strategies in Conversational Recommender Systems: Application of Reinforcement Learning to E-commerce Portals for Learning a System Behavior that is Adapted to the Users in an Interaction Context. VDM Verlag Dr. Müller, Saarbrücken (2010)
[Ma99] Mallat, S.: A Wavelet Tour of Signal Processing, 2nd edn. Academic Press, San Diego (1999)
[MM01] Mangasarian, O.L., Musicant, D.R.: Lagrangian Support Vector Machines, Journal of Machine Learning Research **1**, 161–177 (2001)
[MRLG05] Marthi, B., Russell, S.J., Latham, D., Guestrin, C.: Concurrent hierarchical reinforcement learning. In: Proceedings of the 19th International Conference on Artificial Intelligence (IJCAI), pp. 779–785. Edinburgh, Scotland, UK (2005)
[MHT10] Mazumder, R., Hastie, T., Tibshirani, R.: Spectral regularization algorithms for learning large incomplete matrices. J. Mach. Learn. Res. **11**, 2287–2322 (2010)
[Mun00] Munos, R.: A study of reinforcement learning in the continuous case by the means of viscosity solutions. Mach. Learn. **40**, 265–299 (2000)
[Net06] http://www.netflixprize.com/
[Os11] Oseledets, I.: Tensor-train decomposition. SIAM J. Sci. Comput. **33**(5), 2295–2317 (2011)
[OST08] Oseledets, I., Savostyanov, D., Tyrtyshnikov, E.: Tucker dimensionality reduction of three-dimensional arrays in linear time. SIAM J. Matrix Anal. Appl. **30**(3), 939–956 (2008)
[Os94] Oswald, P.: Multilevel Finite Element Approximation. B.G. Teubner, Stuttgart (1994)
[OT09] Oseledets, I., Tyrtyshnikov, E.: Recursive and tensor-train decompositions in higher dimensions. In: Proceedings of The 9th Hellenic European Research on Computer Mathematics & its Applications Conference. Athens, Greece (2009)
[OT10] Oseledets, I., Tyrtyshnikov, E.: TT-cross approximation for multidimensional arrays. Linear Algebra Appl. **432**, 70–88 (2010)
[Pap09] Paprotny A.: Praktikumsbericht zum Fachpraktikum bei der Firma prudsys AG. (in German) Report. TU Hamburg-Harburg (2009)
[Pap10] Paprotny A.: Hierarchical methods for the solution of dynamic programming equations arising from optimal control problems related to recommendation. Diploma Thesis, TU Hamburg-Harburg (2010)
[Pap11] Paprotny, A.: Multilevel Methods for Dynamic Programming: Deterministic and Stochastic Iterative Methods with Application to Recommendation Engines. AVM – Akademische Verlagsgemeinschaft, München (2011)

[PCTM02] Poole, J., Chang, D.T., Tolbert, D., Mellor, D.: Common Warehouse Metamodel. An Introduction to the Standard for Data Warehouse Integration. Wiley, New York (2002)

[PCTM03] Poole, J., Chang, D.T., Tolbert, D., Mellor, D.: Common Warehouse Metamodel. Developer's Guide. Wiley, New York (2003)

[Pfl10] Pflüger D.: Spatial adaptive sparse grids for high-dimensional problems. Dissertation, TU München (2010)

[PR98] Parr, R., Russel, S.J.: Reinforcement learning with hierarchies of machines. Adv. Neural Inf. Process. Syst., **11**, 1088–1095 (1998)

[PMML] http://www.dmg.org

[Pek77] Pekelis V.: Kleine Enzyklopädie von der großen Kybernetik (in German). Kinderbuchverlag, Berlin (Ost) (1977)

[Pia04] Pias C.: Zeit der Kybernetik – Eine Einstimmung (in German). In: Cyberntics | Kybernetik. The Macy-Conferences 1946–1953. Band 2. diaphanes, Zürich/Berlin (2004)

[RFP10] Recht, B., Fazel, M., Parrilo, P.A.: Guaranteed minimum rank solutions of matrix equations via nuclear norm minimization. SIAM Rev. **52**(3), 471–501 (2010)

[RFST10] Rendle, S., Freudenthaler, C., Schmidt-Thieme L.: Factorizing personalized Markov chains for next-basket recommendation. In: Proceedings of the 19th International World Wide Web Conference (WWW 2010), ACM. Raleigh, NC, USA (2010)

[Rip94] Ripley, B.D.: Neural networks and related methods for classification. J. R. Stat. Soc. B **56**(3), 409–456 (1994)

[RN02] Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall, Englewood Cliffs (2002)

[RRSK11] Ricci, F., Rokach, L., Shapira, B., Kantor, P.B. (eds.): Recommender Systems Handbook. Springer, Berlin (2011)

[RS05] Rennie J.D.M., Srebro N.: Fast maximum margin matrix factorization for collaborative prediction. In: Proceedings of the 22nd International Conference on Machine Learning, pp. 713–719. ACM. New York City (2005)

[RSP05] Rojanavasu, P., Srinil, P., Pinngern, O.: New recommendation system using reinforcement learning. Proceedings of the Fourth International Conference on eBusiness, Bangkok, 19–20 Nov 2005

[Sal97] Salzberg, S.L.: On comparing classifiers: pitfalls to avoid and a recommended approach. Data Min. Knowl Discov. **1**, 317–327 (1997)

[SB98] Sutton, R.S., Barto, A.G.: Reinforcement Learning. An Introduction. MIT Press, Cambridge/London (1998)

[Sem81] Semjonow, N. Wissenschaft und Gesellschaft (in Russian). Nauka (1981)

[SHB05] Shani, G., Heckerman, D., Brafman, R.I.: An MDP-based recommender system. J. Mach. Learn. Res. **6**, 1265–1295 (2005)

[Sin98] Singh, S.: 2d spiral pattern recognition with possible measures. Pattern Recognit. Lett. **19**(2), 141–147 (1998)

[SKKR00] Sarwar, B., Karypis, G., Konstan, J., Riedl J.: Analysis of recommendation algorithms for e-commerce. EC'00, Minneapolis, 17–20 Oct 2000

[Smo63] Smolyak, S.A.: Quadrature and interpolation formulas for tensor products of certain classes of functions. Dokl. Akad. Nauk SSSR. **148**, 1042–1043 (1963). Russian, Engl. Transl.: Soviet Math. Dokl. 4, 240–243 (1963)

[SO11] Savostyanov, D., Oseledets, I.: Fast adaptive interpolation of multi-dimensional arrays in tensor train format. In: Proceedings of 7th International Workshop on Multidimensional Systems (nDS), IEEE. Poitiers, France (2011)

[SPS99] Sutton, R., Precup, D., Singh, S.: Between MDPs and semi-MDPs: a framework for temporal abstraction in reinforcement learning. Artif. Intell. **112**(1), 181–211 (1999)

[StSa96] Sutton R.S., Santamaria J.C.: A standard interface for reinforcement learning software in C++. http://envy.cs.umass.edu/People/sutton/RLinterface/RLinterface.html

[SV10] Sieber, H., Volkmer, T.: Ein Konfidenzintervall für den Mehrumsatz bei einem A-B-Test. (in German) Documentation, prudsys AG, 2010

[The12] Thess, M.: Multilevel preconditioners for temporal-difference learning methods related to recommendation engines. In: Apel, T., Steinbach, O. (eds.) Advanced Finite Element Methods and Applications. Springer, Berlin (2012)

[TA99] Tikhonov, A.N., Arsenin, V.A.: Solutions of Ill-Posed Problems. W.H. Winston, Washington, DC (1977)

[TGK07] Taghipour, N., Ghidary, S.S., Kardan A.: Using q-learning for web recommendations from web usage data. In: 12th International CSI Computer Conference, Teheran (2007)

[TOS01] Stüben, K.: An introduction to algebraic multigrid. In: Trottenberg, U., Oosterlee, C., Schüller, A. (eds.) Multigrid, pp. 413–532. Academic Press, San Diego (2001)

[TVR97] Tsitsiklis, J.N., Roy, B.V.: An analysis of temporal-difference learning with function approximation. IEE Trans. Autom. Control **42**(5), 674–690 (1997)

[VB96] Vandenberghe, L., Boyd, S.: Semidefinite programming. SIAM Rev. **38**, 49–95 (1996)

[Wah90] Wahba, G.: Spline models for observational data, vol. 59. SIAM, Philadelphia (1990)

[Wie88] Wieland, A.: Spiral data set. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/areas/neural/bench/cmu/0.html.

[Wien48] Wiener, N.: Cybernetics or Control and Communication in the Animal and the Machine. Hermann & Cie Editeurs/The Technology Press/Wiley, Paris/Cambridge, MA/New York (1948)

[Wien64] Wiener, N.: God & Golem, Inc. MIT Press, Cambridge (1964)

[Ys86] Yserentant, H.: On the multi-level splitting of finite element spaces. Numer. Math. **49**, 379–412 (1986)

[Zen91] Zenger, C.: Sparse grids. In: Hackbusch, W. (ed.) Parallel Algorithms for Partial Differential Equations, Proceedings of the Sixth GAMM Seminar, Kiel, 1990, Vol. 31 of Notes on Num. Fluid Mech., pp. 241–251. Vieweg-Verlag (1991)

[Zim06] Zimmermann K.-H.: Diskrete Mathematik (in German). Books on Demand, Norderstedt (2006)

[Ziv04] Ziv, O.: Algebraic multigrid for reinforcement learning. Master's Thesis, Technion (2004)

[ZS05] Ziv, O., Shimkin, N.: Multigrid methods for policy evaluation and reinforcement learning. In: 2005 International Symposium on Intelligent Control. Limassol, Cyprus (2005)

[Zu00] Zumbusch, G.: A sparse grid PDE solver. In: Langtangen, H.P., Bruaset A.M., Quak E. (eds.) Advances in Software Tools for Scientific Computing, Proceedings SciTools '98. Lecture Notes in Computational Science and Engineering, vol. 10, chapter 4, pp. 133–178. Springer, Berlin (2000)

[ZWSP08] Zhou Y., Wilkinson, D., Schreiber, R. Pan, R.: Large-scale Parallel Collaborative Filtering for the Netflix Prize. In: AAIM '08: Proceedings of the 4th international conference on Algorithmic Aspects in Information and Management, pp. 337–348, Springer-Verlag, Berlin, Heidelberg (2008)

Applied and Numerical Harmonic Analysis (65 volumes)

A. Saichev and W.A. Woyczyński: *Distributions in the Physical and Engineering Sciences* (ISBN 978-0-8176-3924-2)

C.E. D'Attellis and E.M. Fernandez-Berdaguer: *Wavelet Theory and Harmonic Analysis in Applied Sciences* (ISBN 978-0-8176-3953-2)

H.G. Feichtinger and T. Strohmer: *Gabor Analysis and Algorithms* (ISBN 978-0-8176-3959-4)

R. Tolimieri and M. An: *Time-Frequency Representations* (ISBN 978-0-8176-3918-1)

T.M. Peters and J.C. Williams: *The Fourier Transform in Biomedical Engineering* (ISBN 978-0-8176-3941-9)

G.T. Herman: *Geometry of Digital Spaces* (ISBN 978-0-8176-3897-9)

A. Teolis: *Computational Signal Processing with Wavelets* (ISBN 978-0-8176-3909-9)

J. Ramanathan: *Methods of Applied Fourier Analysis* (ISBN 978-0-8176-3963-1)

J.M. Cooper: *Introduction to Partial Differential Equations with MATLAB* (ISBN 978-0-8176-3967-9)

A. Procházka, N.G. Kingsbury, P.J. Payner, and J. Uhlir: *Signal Analysis and Prediction* (ISBN 978-0-8176-4042-2)

W. Bray and C. Stanojevic: *Analysis of Divergence* (ISBN 978-1-4612-7467-4)

G.T. Herman and A. Kuba: *Discrete Tomography* (ISBN 978-0-8176-4101-6)

K. Gröchenig: *Foundations of Time-Frequency Analysis* (ISBN 978-0-8176-4022-4)

L. Debnath: *Wavelet Transforms and Time-Frequency Signal Analysis* (ISBN 978-0-8176-4104-7)

J.J. Benedetto and P.J.S.G. Ferreira: *Modern Sampling Theory* (ISBN 978-0-8176-4023-1)

D.F. Walnut: *An Introduction to Wavelet Analysis* (ISBN 978-0-8176-3962-4)

A. Abbate, C. DeCusatis, and P.K. Das: *Wavelets and Subbands* (ISBN 978-0-8176-4136-8)

O. Bratteli, P. Jorgensen, and B. Treadway: *Wavelets Through a Looking Glass* (ISBN 978-0-8176-4280-80

A. Paprotny and M. Thess, *Realtime Data Mining: Self-Learning Techniques for Recommendation Engines*, Applied and Numerical Harmonic Analysis,
DOI 10.1007/978-3-319-01321-3, © Springer International Publishing Switzerland 2013

H.G. Feichtinger and T. Strohmer: *Advances in Gabor Analysis* (ISBN 978-0-8176-4239-6)

O. Christensen: *An Introduction to Frames and Riesz Bases* (ISBN 978-0-8176-4295-2)

L. Debnath: *Wavelets and Signal Processing* (ISBN 978-0-8176-4235-8)

G. Bi and Y. Zeng: *Transforms and Fast Algorithms for Signal Analysis and Representations* (ISBN 978-0-8176-4279-2)

J.H. Davis: *Methods of Applied Mathematics with a MATLAB Overview* (ISBN 978-0-8176-4331-7)

J.J. Benedetto and A.I. Zayed: *Modern Sampling Theory* (ISBN 978-0-8176-4023-1)

E. Prestini: *The Evolution of Applied Harmonic Analysis* (ISBN 978-0-8176-4125-2)

L. Brandolini, L. Colzani, A. Iosevich, and G. Travaglini: *Fourier Analysis and Convexity* (ISBN 978-0-8176-3263-2)

W. Freeden and V. Michel: *Multiscale Potential Theory* (ISBN 978-0-8176-4105-4)

O. Christensen and K.L. Christensen: *Approximation Theory* (ISBN 978-0-8176-3600-5)

O. Calin and D.-C. Chang: *Geometric Mechanics on Riemannian Manifolds* (ISBN 978-0-8176-4354-6)

J.A. Hogan: *Time-Frequency and Time-Scale Methods* (ISBN 978-0-8176-4276-1)

C. Heil: *Harmonic Analysis and Applications* (ISBN 978-0-8176-3778-1)

K. Borre, D.M. Akos, N. Bertelsen, P. Rinder, and S.H. Jensen: *A Software-Defined GPS and Galileo Receiver* (ISBN 978-0-8176-4390-4)

T. Qian, M.I. Vai, and Y. Xu: *Wavelet Analysis and Applications* (ISBN 978-3-7643-7777-9)

G.T. Herman and A. Kuba: *Advances in Discrete Tomography and Its Applications* (ISBN 978-0-8176-3614-2)

M.C. Fu, R.A. Jarrow, J.-Y. Yen, and R.J. Elliott: *Advances in Mathematical Finance* (ISBN 978-0-8176-4544-1)

O. Christensen: *Frames and Bases* (ISBN 978-0-8176-4677-6)

P.E.T. Jorgensen, J.D. Merrill, and J.A. Packer: *Representations, Wavelets, and Frames* (ISBN 978-0-8176-4682-0)

M. An, A.K. Brodzik, and R. Tolimieri: *Ideal Sequence Design in Time-Frequency Space* (ISBN 978-0-8176-4737-7)

S.G. Krantz: *Explorations in Harmonic Analysis* (ISBN 978-0-8176-4668-4)

B. Luong: *Fourier Analysis on Finite Abelian Groups* (ISBN 978-0-8176-4915-9)

G.S. Chirikjian: *Stochastic Models, Information Theory, and Lie Groups, Volume 1* (ISBN 978-0-8176-4802-2)

C. Cabrelli and J.L. Torrea: *Recent Developments in Real and Harmonic Analysis* (ISBN 978-0-8176-4531-1)

M.V. Wickerhauser: *Mathematics for Multimedia* (ISBN 978-0-8176-4879-4)

B. Forster, P. Massopust, O. Christensen, K. Gröchenig, D. Labate, P. Vandergheynst, G. Weiss, and Y. Wiaux: *Four Short Courses on Harmonic Analysis* (ISBN 978-0-8176-4890-9)

O. Christensen: *Functions, Spaces, and Expansions* (ISBN 978-0-8176-4979-1)

J. Barral and S. Seuret: *Recent Developments in Fractals and Related Fields* (ISBN 978-0-8176-4887-9)

O. Calin, D.-C. Chang, and K. Furutani, and C. Iwasaki: *Heat Kernels for Elliptic and Sub-elliptic Operators* (ISBN 978-0-8176-4994-4)

C. Heil: *A Basis Theory Primer* (ISBN 978-0-8176-4686-8)

J.R. Klauder: *A Modern Approach to Functional Integration* (ISBN 978-0-8176-4790-2)

J. Cohen and A.I. Zayed: *Wavelets and Multiscale Analysis* (ISBN 978-0-8176-8094-7)

D. Joyner and J.-L. Kim: *Selected Unsolved Problems in Coding Theory* (ISBN 978-0-8176-8255-2)

G.S. Chirikjian: *Stochastic Models, Information Theory, and Lie Groups, Volume 2* (ISBN 978-0-8176-4943-2)

J.A. Hogan and J.D. Lakey: *Duration and Bandwidth Limiting* (ISBN 978-0-8176-8306-1)

G. Kutyniok and D. Labate: *Shearlets* (ISBN 978-0-8176-8315-3)

P.G. Casazza and P. Kutyniok: *Finite Frames* (ISBN 978-0-8176-8372-6)

V. Michel: *Lectures on Constructive Approximation* (ISBN 978-0-8176-8402-0)

D. Mitrea, I. Mitrea, M. Mitrea, and S. Monniaux: *Groupoid Metrization Theory* (ISBN 978-0-8176-8396-2)

T.D. Andrews, R. Balan, J.J. Benedetto, W. Czaja, and K.A. Okoudjou: *Excursions in Harmonic Analysis, Volume 1* (ISBN 978-0-8176-8375-7)

T.D. Andrews, R. Balan, J.J. Benedetto, W. Czaja, and K.A. Okoudjou: *Excursions in Harmonic Analysis, Volume 2* (ISBN 978-0-8176-8378-8)

D.V. Cruz-Uribe and A. Fiorenza: *Variable Lebesgue Spaces* (ISBN 978-3-0348-0547-6)

W. Freeden and M. Gutting: *Special Functions of Mathematical (Geo-)Physics* (ISBN 978-3-0348-0562-9)

A. Saichev and W.A. Woyczyński: *Distributions in the Physical and Engineering Sciences, Volume 2: Linear and Nonlinear Dynamics of Continuous Media* (ISBN 978-0-8176-3942-6)

S. Foucart and H. Rauhut: *A Mathematical Introduction to Compressive Sensing* (ISBN 978-0-8176-4947-0)

G. Herman and J. Frank: *Computational Methods for Three-Dimensional Microscopy Reconstruction* (ISBN 978-1-4614-9520-8)

A. Paprotny and M. Thess: *Realtime Data Mining: Self-Learning Techniques for Recommendation Engines* (ISBN 978-3-319-01320-6)

For an up-to-date list of ANHA titles, please visit http://www.springer.com/series/4968