Martin P. Robillard · Walid Maalej
Robert J. Walker · Thomas Zimmermann
Editors

Recommendation Systems in Software Engineering

Springer

Recommendation Systems in Software Engineering

Martin P. Robillard • Walid Maalej •
Robert J. Walker • Thomas Zimmermann
Editors

Recommendation Systems in Software Engineering

Springer

Editors

Martin P. Robillard
McGill University
Montréal, QC
Canada

Walid Maalej
University of Hamburg
Hamburg
Germany

Robert J. Walker
University of Calgary
Calgary, AB
Canada

Thomas Zimmermann
Microsoft Research
Redmond, WA
USA

ISBN 978-3-642-45134-8 ISBN 978-3-642-45135-5 (eBook)
DOI 10.1007/978-3-642-45135-5
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014931203

© Springer-Verlag Berlin Heidelberg 2014
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.springer.com)

Preface

Software developers have always used tools to perform their work. In the earliest days of the discipline, the tools provided basic compilation and assembly functionality. Then came tools and environments that increasingly provided sophisticated data about the software under development. Around the turn of the millennium, the systematic and large-scale accumulation of software engineering data opened up new opportunities for the creation of tools that infer information estimated to be helpful to developers in a given context. This new type of software development tools came to be known as recommendation systems, in parallel with similar developments in other domains such as the e-commerce.

Recommendation systems in software engineering (RSSEs) share commonalities with conventional recommendation systems: mainly in their usage model, the usual reliance on data mining, and in the predictive nature of their functionality. Beyond these superficial traits, recommendation systems in software engineering are generally different from those in other domains. Traditional recommendation systems are heavily *user centric*. Users generally create the data items directly, e.g., in the form of ratings. An important challenge for traditional recommendation systems is to infer and model evolving user preferences and needs. In contrast, the major challenge for designing RSSEs is to automatically interpret the highly technical data stored in software repositories.

Realizing that some of the important knowledge that is necessary to build recommendation systems in a technical domain would not be readily found in existing books and other resources on conventional recommendation systems, we set about to capture as much of this knowledge as possible in this book.

About This Book

This book has been a community effort. Prospective authors submitted chapter proposals to an open call for contributions. The proposals and later the selected chapters were reviewed by the editors over four review iterations. In addition, the authors participating in this book were asked to review chapters by other contributors.

A unique aspect of this book was the RSSE Hamburg Meeting in April 2013. The contributing authors were invited to this 2-day event to present their chapter ideas, discuss the RSSE state of the art, and participate in editing and networking sessions. The meeting greatly helped to unify the presentation and content of this book and to further consolidate the RSSE community effort. The meeting has been part of a series of events that started with a workshop on software analysis for recommendation systems at McGill University's Bellairs Research Station in Barbados in 2008 and follow-up workshops at the ACM SIGSOFT International Symposium on the Foundations of Software Engineering in 2008 and at the ACM/IEEE International Conference on Software Engineering in 2010 and 2012. The last workshop in 2012 had over 70 participants, which shows a large interest in the topic.

Structure and Content

This book collects, structures, and formalizes knowledge on recommendation systems in software engineering. It adopts a pragmatic approach with an explicit focus on system design, implementation, and evaluation. The book is intended to complement existing texts on recommender systems, which cover algorithms and traditional application domains.

The book consists of three parts:

Part I: Techniques This part introduces basic techniques for building recommenders in software engineering, including techniques not only to collect and process software engineering data but also to present recommendations to users as part of their workflow.

Part II: Evaluation This part summarizes methods and experimental designs to evaluate recommendations in software engineering.

Part III: Applications This part describes needs, issues, and solution concepts involved in entire recommendation systems for specific software engineering tasks, focusing on the engineering insights required to make effective recommendations.

Target Audience

The book contains knowledge relevant to software professionals and to computer science or software engineering students with an interest in the application of recommendation technologies to highly technical domains, including:

- senior undergraduate and graduate students working on recommendation systems or taking a course in software engineering or applied data mining;
- researchers working on recommendation systems or on software engineering tools;
- software engineering practitioners developing recommendation systems or similar applications with predictive functionality; and
- instructors teaching a course on recommendation systems, applied data mining, or software engineering. The book will be particularly suited to graduate courses involving a project component.

Website and Resources

This book has a webpage at rsse.org/book, which is part of the RSSE community portal rsse.org. This webpage contains free supplemental materials for readers of this book and anyone interested in recommendation systems in software engineering, including:

- lecture slides, datasets, and source code;
- an archive of previous RSSE workshops and meetings;
- a collection of people, papers, groups, and tools related to RSSE. Please contact any of the editors if you would like to be added or to suggest additional resources.

In addition to the RSSE community, there are several other starting points.

- The article "Recommendation Systems for Software Engineering," IEEE Software, 27(4):80–86, July–August 2010, provides a short introduction to the topic.
- The latest research on RSSE systems is regularly published and presented at the International Conference on Software Engineering (ICSE), International Symposium on the Foundations of Software Engineering (FSE), International Conference on Automated Software Engineering (ASE), Working Conference on Mining Software Repositories (MSR), and International Conference on Software Maintenance (ICSM).
- Many researchers working on RSSE systems meet at the International Workshop on Recommendation Systems for Software Engineering, which is typically held every other year.
- The ACM Conference on Recommender Systems (RecSys) covers recommender research in general and in many different application domains, not just software engineering.
- Several books on building conventional recommendation systems have been written. To get started, we recommend "Recommender Systems: An Introduction" (2010) by Jannach, Zanker, Felfernig, and Friedrich.

Acknowledgments

We are indebted to the many people who have made this book possible through their diverse contributions. In particular, we thank:

- The authors of the chapters in this book for the great work that they have done in writing about topics related to RSSEs and their timely and constructive reviews of other chapters.
- The Springer staff, in particular Ralf Gerstner, for their dedication and helpfulness throughout the project.
- The Dean of the MIN faculty and Head of Informatics Department at the University of Hamburg, for their financial and logistical support for the editing workshop, and Rebecca Tiarks and Tobias Roehm, for their help with the local organization of the workshop.
- The attendees at the past RSSE workshops, for their enthusiasm about the topic and their influx of ideas.
- The McGill Bellairs Research Institute, for providing an ideal venue for an initial workshop on the topic.

Montréal, QC, Canada	Martin P. Robillard
Hamburg, Germany	Walid Maalej
Calgary, AB, Canada	Robert J. Walker
Redmond, WA, USA	Thomas Zimmermann
October 2013	

Contents

List of Contributors

Colin Atkinson Software-Engineering Group, University of Mannheim, Mannheim, Germany

Iman Avazpour Faculty of ICT, Centre for Computing and Engineering Software and Systems (SUCCESS), Swinburne University of Technology, Hawthorn, Australia

Gabriele Bavota University of Sannio, Benevento, Italy

Ayşe Bener Ryerson University, Toronto, ON, Canada

Markus Borg Department of Computer Science, Lund University, Lund, Sweden

Bora Çağlayan Boğaziçi University, Istanbul, Turkey

Gül Çalıklı Ryerson University, Toronto, ON, Canada

Carlos Castro-Herrera GOOGLE, Chicago, IL, USA

Jane Cleland-Huang School of Computing, DePaul University, Chicago, IL, USA

Paolo Cremonesi Politecnico di Milano, Milano, Italy

Andrea De Lucia University of Salerno, Fisciano, Italy

Daniel Diaz Université Paris 1 Panthéon-Sorbonne, Paris, France

Cosmin Dumitrescu Université Paris 1 Panthéon-Sorbonne, Paris, France

Alexander Felfernig Institute for Software Technology, Graz University of Technology, Graz, Austria

Thomas Fritz Department of Informatics, University of Zurich, Zurich, Switzerland

John Grundy Faculty of ICT, Centre for Computing and Engineering Software and Systems (SUCCESS), Swinburne University of Technology, Hawthorn, Australia

Lars Grunske Institute of Software Technology, Universität Stuttgart, Stuttgart, Germany

Negar Hariri School of Computing, DePaul University, Chicago, IL, USA

Kim Herzig Saarland University, Saarbrücken, Germany

Reid Holmes David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada

Oliver Hummel Institute for Program Structures and Data Organization, Karlsruhe Institute of Technology, Karlsruhe, Germany

Laura Inozemtseva David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada

Werner Janjic Software-Engineering Group, University of Mannheim, Mannheim, Germany

Michael Jeran Institute for Software Technology, Graz University of Technology, Graz, Austria

Miryung Kim The University of Texas at Austin, Austin, TX, USA

Angela Lozano ICTEAM, Université catholique de Louvain, Louvain-la-Neuve, Belgium

Walid Maalej Department of Informatics, University of Hamburg, Hamburg, Germany

Andrian Marcus Wayne State University, Detroit, MI, USA

Raúl Mazo Université Paris 1 Panthéon-Sorbonne, Paris, France

Na Meng The University of Texas at Austin, Austin, TX, USA

Kim Mens ICTEAM, Université Catholique de Louvain, Louvain-la-Neuve, Belgium

Tim Menzies Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV, USA

Ayşe Tosun Mısırlı University of Oulu, Oulu, Finland

Bamshad Mobasher School of Computing, DePaul University, Chicago, IL, USA

Gail C. Murphy University of British Columbia, Vancouver, BC, Canada

Emerson Murphy-Hill North Carolina State University, Raleigh, NC, USA

Gerald Ninaus Institute for Software Technology, Graz University of Technology, Graz, Austria

Rocco Oliveto University of Molise, Pesche, Italy

Teerat Pitakrat Institute of Software Technology, Universität Stuttgart, Stuttgart, Germany

Florian Reinfrank Institute for Software Technology, Graz University of Technology, Graz, Austria

Stefan Reiterer Institute for Software Technology, Graz University of Technology, Graz, Austria

Romain Robbes Computer Science Department (DCC), University of Chile, Santiago, Chile

Martin P. Robillard McGill University, Montréal, QC, Canada

Per Runeson Department of Computer Science, Lund University, Lund, Sweden

Alan Said Centrum Wiskunde & Informatica, Amsterdam, The Netherlands

Camille Salinesi Université Paris 1 Panthéon-Sorbonne, Paris, France

Martin Stettinger Institute for Software Technology, Graz University of Technology, Graz, Austria

Domonkos Tikk Gravity R&D, Budapest, Hungary

Óbuda University, Budapest, Hungary

Burak Turhan University of Oulu, Oulu, Finland

Robert J. Walker Department of Computer Science, University of Calgary, Calgary, AB, Canada

Annie T.T. Ying McGill University, Montréal, QC, Canada

Andreas Zeller Saarland University, Saarbrücken, Germany

Chapter 1
An Introduction to Recommendation Systems in Software Engineering

Martin P. Robillard and Robert J. Walker

Abstract Software engineering is a knowledge-intensive activity that presents many information navigation challenges. Information spaces in software engineering include the source code and change history of the software, discussion lists and forums, issue databases, component technologies and their learning resources, and the development environment. The technical nature, size, and dynamicity of these information spaces motivate the development of a special class of applications to support developers: recommendation systems in software engineering (RSSEs), which are software applications that provide information items estimated to be valuable for a software engineering task in a given context. In this introduction, we review the characteristics of information spaces in software engineering, describe the unique aspects of RSSEs, present an overview of the issues and considerations involved in creating, evaluating, and using RSSEs, and present a general outlook on the current state of research and development in the field of recommendation systems for highly technical domains.

1.1 Introduction

Despite steady advancement in the state of the art, software development remains a challenging and knowledge-intensive activity. Mastering a programming language is no longer sufficient to ensure software development proficiency. Developers are continually introduced to new technologies, components, and ideas. The systems on

M.P. Robillard (✉)
McGill University, Montréal, QC, Canada
e-mail: martin@cs.mcgill.ca

R.J. Walker
University of Calgary, Calgary, AB, Canada
e-mail: walker@ucalgary.ca

which they work tend to keep growing and to depend on an ever-increasing array of external libraries and resources.

We have long since reached the point where the scale of the *information space*—facing a typical developer easily exceeds an individual's capacity to assimilate it. Software developers and other technical knowledge workers must now routinely spend a large fraction of their working time searching for information, for example, to understand existing code or to discover how to properly implement a feature. Often, the timely or serendipitous discovery of a critical piece of information can have a dramatic impact on productivity [6].

Although rigorous training and effective interpersonal communication can help knowledge workers orient themselves in a sea of information, these strategies are painfully limited by scale. Data mining and other knowledge inference techniques are among the ways to provide automated assistance to developers in navigating large information spaces. Just as recommendation systems for popular e-commerce Web sites can help expose users to interesting items previously unknown to them [15], recommendation systems can be used in technical domains to help surface previously unknown information that can directly assist knowledge workers in their task.

Recommendation systems in software engineering (RSSEs) are now emerging to assist software developers in various activities—from reusing code to writing effective bug reports.

1.2 Information Spaces in Software Engineering

When developers join a project, they are typically faced with a *landscape* [4] of information with which they must get acquainted. Although this information landscape will vary according to the organization and the development process employed, the landscape will typically involve information from a number of sources.

The project source code. In the case of large software systems, the codebase itself will already represent a formidable information space. According to Ohloh.net, in October 2013 the source code of the Mozilla Firefox Web browser totaled close to 10 million lines written in 33 different programming languages. Understanding source code, even at a much smaller scale, requires answering numerous different types of questions, such as "where is this method called?" [19]. Answering such structural questions can require a lot of navigation through the project source code [11, 17], including reading comments and identifiers, following dependencies, and abstracting details.

The project history. Much knowledge about a software project is captured in the version control system (VCS) for the project. Useful information stored in a VCS includes systematic code change patterns (e.g., files *A* and *B* were often changed together [22]), design decisions associated with specific changes (stored in commit logs), and, more indirectly, information about which developers have knowledge of which part of the code [13]. Unfortunately, the information contained in a VCS is not easily searchable or browsable. Useful knowledge must often be inferred from the VCS and other repositories, typically by using a combination of heuristics and data mining techniques [21].

Communication archives. Forums and mailing lists, often used for informal communication among developers and other stakeholders of a project, contain a wealth of knowledge about a system [3]. Communication is also recorded in issue management systems and code review tools.

The dependent APIs and their learning resources. Most modern software development relies on reusable software assets (frameworks and libraries) exported through application programming interfaces (APIs). Like the project source code itself, APIs introduce a large, heavily structured information space that developers must understand and navigate to complete their tasks. In addition, large and popular APIs typically come with extensive documentation [5], including reference documentation, user manuals, and code examples.

The development environment. The development environment for a software system includes all the development tools, scripts, and commands used to build and test the system. Such an environment can quickly become complex to the point where developers perform suboptimally simply because they are unaware of the tools and commands at their disposal [14].

Interaction traces. It is now common practice for many software applications to collect user interaction data to improve the user experience. User interaction data consists of a log of user actions as they visit a Web site or use the various components of the user interface of a desktop or mobile application [8]. In software engineering, this collection of usage data takes the form of the monitoring of developer actions as they use an integrated development environment such as Eclipse [10].

Execution traces. Data collected during the execution of a software system [16, Table 3] also constitutes a source of information that can be useful to software engineers, and in particular to software quality assurance teams. This kind of dynamically collected information includes data about the state of the system, the functions called, and the results of computation at different times in the execution of the system.

The web. Ultimately, some of the knowledge sought by or useful to developers can be found in the cloud, hosted on servers unrelated to a given software development project. For example, developers will look for code examples on the web [2], or visit the StackOverflow Questions-and-Answers (Q&A) site in the hopes of finding answers to common programming problems [12]. The problem with the cloud is that it is often difficult to assess the quality of the information found in some Web sites, and near impossible to estimate what information exists beyond the results of search queries.

Together, the various sources of data described above create the information space that software developers and other stakeholders of a software project will face. Although, in principle, all of this information is available to support ongoing development and other engineering activities, in reality it can be dispiritingly hard to extract the answer to a specific information need from software engineering data, or in some cases to even know that the answer exists. A number of aspects of software engineering data make discovering and navigating information in this domain particularly difficult.

1. The sheer amount of information available (the *information overload* problem), while not unique to software engineering, is an important factor that only grows worse with time. Automatically collected execution traces and interaction traces, and the cumulative nature of project history data, all contribute to making this challenge more acute.
2. The information associated with a software project is *heterogeneous*. While a vast array of traditional recommender systems can rely on the general concepts of *item* and *rating* [15], there is no equivalent universal baseline in software engineering. The information sources described above involve a great variety of information formats, including highly structured (source code), semi-structured (bug reports), and loosely structured (mailing lists, user manuals).
3. Technical information is highly *context-sensitive*. To a certain extent, most information is context-sensitive; for example, to interpret a restaurant review, it may be useful to know about the expectations and past reviews of the author. However, even in the absence of such additional context, it will still be possible to construct a coarse interpretation of the information, especially if the restaurant in question is either very good or very bad. In contrast, software engineering data can be devoid of meaning without an explicit connection to the underlying process. For example, if a large amount of changes are committed to a system's version control system on Friday afternoons, it could mean either that team members have chosen that time to merge and integrate their changes or that a scheduled process updates the license headers at that time.
4. Software data *evolves very rapidly*. Ratings for movies can have a useful lifetime measured in decades. Restaurant and product reviews are more ephemeral, but could be expected to remain valid for at least many months. In contrast, some software data experiences high *churn*, meaning that it is modified in some cases multiple times a day [9]. For example, the Mozilla Firefox project receives around 4,000 commits per month, or over 100 per day. Although not all software data gets invalidated on a daily basis (APIs can remain stable for years), the highly dynamic nature of software means that inferred facts must, in principle, continually be verified for consistency with the underlying data.
5. Software data is *partially generated*. Many software artifacts are the result of a combination of manual and automated processes and activities, often involving a complex cycle of artifact generation with manual feedback. Examples include the writing of source code with the help of refactoring or style-checking tools, the authoring of bug reports in which the output or log of a program is copied and pasted, and the use of scripts to automatically generate mailing list messages, for example, when a version of the software is released. These complex and

semiautomated processes can be contrasted, for example, with the authoring of reviews by customers who have bought a certain product. In the latter case, the process employed for generating the data is transparent, and interpreting it will be a function of the content of the item and the attributes of the author; the data generation process would not normally have to be taken into account to understand the review.

Finally, in addition to the challenging attributes of software engineering data that we noted above, we also observe that many problems in software engineering are not limited by data, but rather by computation. Consider a problem like *change impact analysis* [1, 20]: the basic need of the developer—to determine the impact of a proposed change—is clear, but in general it is impossible to compute a precise solution. Thus, in software engineering and other technical domains, guidance in the form of recommendations is needed not only to navigate large information spaces but also to deal with *formally undecidable problems*, or problems where no precise solutions can be computed in a practical amount of time.

1.3 Recommendation Systems in Software Engineering

In our initial publication on the topic, we defined a recommendation system for software engineering to be [18, p.81]:

> . . . a software application that provides information items estimated to be valuable for a software engineering task in a given context.

With the perspective of an additional four years, we still find this definition to be the most useful for distinguishing RSSEs from other software engineering tools. RSSEs' focus is on providing *information* as opposed to other services such as build or test automation. The reference to *estimation* distinguishes RSSEs from fact extractors, such as classical search tools based on regular expressions or the typical cross-reference tools and call-graph browsers found in modern integrated development environments. At the same time, estimation is not necessarily *prediction*: recommendation systems in software engineering need not rely on the accurate prediction of developer behavior or system behavior. The notion of *value* captures two distinct aspects simultaneously: (1) novelty and surprise, because RSSEs support discovering new information and (2) familiarity and reinforcement, because RSSEs support the confirmation of existing knowledge. Finally, the reference to a specific *task* and *context* distinguishes RSSEs from generic search tools, e.g., tools to help developers find code examples.

Our definition of RSSEs is, however, still broad and allows for great variety in recommendation support for developers. Specifically, a large number of different information items can be recommended, including the following:

Source code within a project. Recommenders can help developers navigate the source code of their own project, for example, by attempting to guess the areas of the project's source code a developer might need, or want, to look at.

Reusable source code. Other recommenders in software engineering attempt to help users discover the API elements (such as classes, functions, or scripts) that can help to complete a task.

Code examples. In some cases, a developer may know which source code or API elements are required to complete a task, but may ignore how to correctly employ them. As a complement to reading textual documentation, recommendation systems can also provide code examples that illustrate the use of the code elements of interest.

Issue reports. Much knowledge about a software project can reside in its issue database. When working on a piece of code or attempting to solve a problem, recommendation systems can discover related issue reports.

Tools, commands, and operations. Large software development environments are getting increasingly complex, and the number of open-source software development tools and plug-ins is unbounded. Recommendation systems can help developers and other software engineers by recommending tools, commands, and actions that should solve their problem or increase their efficiency.

People. In some situations recommendation systems can also help finding the best person to assign a task to, or the expert to contact to answer a question.

Although dozens of RSSEs have been built to provide some of the recommendation functionality described above, no reference architecture has emerged to-date. The variety in RSSE architectures is likely a consequence of the fact that most RSSEs work with a dominant source of data, and are therefore engineered to closely integrate with that data source. Nevertheless, the major design concerns for recommendation systems in general are also found in the software engineering domain, each with its particular challenges.

Data preprocessing. In software engineering, a lot of preprocessing effort is required to turn raw character data into a sufficiently interpreted format. For example, source code has to be parsed, commits have to be aggregated, and software has to be abstracted into dependency graphs. This effort is usually needed in addition to more traditional preprocessing tasks such as detecting outliers and replacing missing values.

Capturing context. While in traditional domains, such as e-commerce, recommendations are heavily dependent on user profiles, in software engineering, it is usually the *task* that is the central concept related to recommendations. The *task context* is our representation of all information about the task to which the recommendation system has access in order to produce recommendations. In many cases, a task context will consist of a partial view of the solution to the task: for example, some source code that a developer has written, an element in the code that a user has selected, or an issue report that a user is reading. Context can also be specified explicitly, in which case the definition of the context becomes fused with that of a query in a traditional information retrieval system.

In any case, capturing the context of a task to produce recommendations involves somewhat of a paradox: the more precise the information available about the task is, the more accurate the recommendations can be, but the less likely the user can be expected to need recommendations. Put another way, a user in great need of guidance may not be able to provide enough information to the system to obtain usable recommendations. For this reason, recommendation systems must take into account that task contexts will generally be incomplete and noisy.

Producing recommendations. Once preprocessed data and a sufficient amount of task context are available, recommendation algorithms can be executed. Here the variety of recommendation strategies is only bounded by the problem space and the creativity of the system designer. However, we note that the traditional recommendation algorithms commonly known as collaborative filtering are only seldom used to produce recommendations in software engineering.

Presenting the recommendations. In its simplest form, presenting a recommendation boils down to listing items of potential interest—functions, classes, code examples, issue reports, and so on. Related to the issue of presentation, however, lies the related question of *explanation*: why was an item recommended? The answer to this question is often a summary of the recommendation strategy: "average rating," "customers who bought this item also bought," etc. In software engineering, the conceptual distance between a recommendation algorithm and the domain familiar to the user is often much larger than in other domains. For example, if a code example is recommended to a user because it matches part of the user's current working code, how can this matching be summarized? The absence of a universal concept such as ratings means that for each new type of recommendation, the question of explanation must be revisited.

1.4 Overview of the Book

In the last decade, research and development on recommendation systems has seen important advances, and the knowledge relevant to recommendation systems now easily exceeds the scope of a single book. This book focuses on the development of recommendation systems for technical domains and, in particular, for software engineering. The topic of recommendation systems in software engineering is broad to the point of multidisciplinarity: it requires background in software engineering, data mining and artificial intelligence, knowledge modeling, text analysis and information retrieval, human–computer interaction, as well as a firm grounding in empirical research methods. This book was designed to present a self-contained overview that includes sufficient background in all of the relevant areas to allow readers to quickly get up to speed on the most recent developments, and to actively use the knowledge provided here to build or improve systems that can take advantage of large information spaces that include technical content.

Part I of the book covers the foundational aspects of the field. Chapter 2 presents an overview of the general field of recommendation systems, including a presentation of the major classes of recommendation approaches: collaborative filtering, content-based recommendations, and knowledge-based recommendations. Many recommendation systems rely on data mining algorithms; to help readers orient themselves in the space of techniques available to infer facts from large data sets, Chap. 3 presents a tutorial on popular data mining techniques. In contrast, Chap. 4 examines how recommendation systems can be built without data mining, by relying instead on carefully designed heuristics. To-date, the majority of RSSEs have targeted the recommendation of source code artifacts; Chap. 5 is an extensive review of recommendation systems based on source code that includes many examples of RSSEs. Moving beyond source code, we examine two other important sources of data for RSSE: bug reports in Chap. 6, and user interaction data in Chap. 7. We conclude Part I with two chapters on human–computer interaction (HCI) topics: the use of developer profiles to take personal characteristics into account, in Chap. 8, and the design of user interfaces for delivering recommendations, in Chap. 9.

Now that the field of recommendation systems has matured, many of the basic ideas have been tested, and further progress will require careful, well-designed evaluations. Part II of the book is dedicated to the evaluation of RSSEs with four chapters on the topic. Chapter 10 is a review of the most important dimensions and metrics for evaluating recommendation systems. Chapter 11 focuses on the problem of creating quality benchmarks for evaluating recommendation systems. The last two chapters of Part II describe two particularly useful types of studies for evaluating RSSEs: simulation studies that involve the execution of the RSSE (or of some of its components) in a synthetic environment (Chap. 12), and field studies, which involve the development and deployment of an RSSE in a production setting (Chap. 13).

Part III of the book takes a detailed look at a number of specific applications of recommendation technology in software engineering. By discussing RSSEs in an end-to-end fashion, the chapters in Part III provide not only a discussion of the major concerns and design decisions involved in developing recommendation technology in software engineering but also insightful illustrations of how computation can assist humans in solving a wide variety of complex, information-intensive tasks. Chapter 14 discusses the techniques underlying the recommendation of reusable source code elements. Chapters 15 and 16 present two different approaches to recommend transformations to an existing codebase. Chapter 17 discusses how recommendation technology can assist requirements engineering, and Chap. 18 focuses on recommendations that can assist tasks involving issue reports, such as issue triage tasks. Finally, Chap. 19 shows how recommendations can assist with product line configuration tasks.

1.5 Outlook

As the content of this book shows, the field of recommendation systems in software engineering has already benefited from much effort and attention from researchers, tool developers, and organizations interested in leveraging large collections of software artifacts to improve software engineering productivity. We conclude this introduction with a look at the current state of the field and the road ahead.

Most of the work on RSSEs to-date has focused on the development of algorithms for processing software data. Much of this work has proceeded in the context of the rapid progress in techniques to mine software repositories. As a result, developers of recommendation systems in software engineering can now rely on a mature body of knowledge on the automated extraction and interpretation of software data [7]. At the same time, developments in RSSEs had, up to recently, proceeded somewhat in isolation of the work on traditional recommender systems. However, the parallel has now been recognized, which we hope will lead to a rapid convergence in terminology and concepts that should facilitate further exchange of ideas between the two communities.

Although many of the RSSEs mentioned in this book have been fully implemented, much less energy has been devoted to research on the human aspects of RSSEs. For a given RSSE, simulating the operation of a recommendation algorithm can allow us to record very exactly how the algorithm would behave in a large number of contexts, but provides no clue as to how users would react to the recommendations (see Part II). For this purpose, only user studies can really provide an answer. The dearth of user studies involving recommendation systems in software engineering can be explained and justified by their high cost, which would not always be in proportion to the importance of the research questions involved. However, the consequence is that we still know relatively little about how to best integrate recommendations into a developer's workflow, how to integrate recommendations from multiple sources, and more generally how to maximize the usefulness of recommendation systems in software engineering.

An important distinction between RSSEs and traditional recommendation systems is that RSSEs are task-centric, as opposed to user-centric. In many recommendation situations, we know much more about the task than about the developer carrying it out. This situation is reflected in the limited amount of personalization in RSSEs. It remains an open question whether personalization is necessary or even desirable in software engineering. As in many cases, the accumulation of personal information into a user (or developer) profile has important privacy implications. In software engineering, the most obvious one is that this information could be directly used to evaluate developers. A potential development that could lead to more personalization in recommender systems for software engineering is the increasingly pervasive use of social networking in technical domains. Github is already a platform where the personal characteristics of users can be used to navigate information. In this scenario, we would see a further convergence between RSSEs and traditional recommenders.

Traditional recommendation systems provide a variety of *functions* [15, Sect. 1.2]. Besides assisting the user in a number of ways, these functions also include a number of benefits to other stakeholders, including commercial organizations. For example, recommendation systems can help increase the number of items sold, sell more diverse items, and increase customer loyalty. Although, in the case of RSSEs developed by commercial organizations, these functions can be assumed, we are not aware of any research that focuses on assessing the nontechnical virtues of RSSEs. At this point, most of the work on assessing RSSEs has focused on the support they directly provide to developers.

1.6 Conclusion

The information spaces encountered in software engineering contexts differ markedly from those in nontechnical domains. Five aspects—quantity, heterogeneity, context-sensitivity, dynamicity, and partial generation—all contribute to making it especially difficult to analyze, interpret, and assess the quality of software engineering data. The computational intractability of many questions that surface in software engineering only add to the complexity. Those are the challenges facing organizations that wish to leverage their software data.

Recommendation systems in software engineering are one way to cope with these challenges. At heart, RSSEs must be designed to acknowledge the realities of the tasks, of the people, and of the organizations involved. And while developing effective RSSEs gives rise to new challenges, we have already learned a great deal about the techniques to create them, the methodologies to evaluate them, and the details of their application.

References

1. Arnold, R.S., Bohner, S.A.: Impact analysis: Towards a framework for comparison. In: Proceedings of the Conference on Software Maintenance, pp. 292–301 (1993). DOI 10.1109/ICSM.1993.366933
2. Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M., Klemmer, S.R.: Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In: Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems, pp. 1589–1598 (2009). DOI 10.1145/1518701.1518944
3. Čubranić, D., Murphy, G.C., Singer, J., Booth, K.S.: Hipikat: A project memory for software development. IEEE Trans. Software Eng. **31**(6), 446–465 (2005). DOI 10.1109/TSE.2005.71
4. Dagenais, B., Ossher, H., Bellamy, R.K., Robillard, M.P.: Moving into a new software project landscape. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 275–284 (2010)
5. Dagenais, B., Robillard, M.P.: Creating and evolving developer documentation: Understanding the decisions of open source contributors. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 127–136 (2010). DOI 10.1145/1882291.1882312
6. Duala-Ekoko, E., Robillard, M.P.: Asking and answering questions about unfamiliar APIs: An exploratory study. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 266–276 (2012)
7. Hemmati, H., Nadi, S., Baysal, O., Kononenko, O., Wang, W., Holmes, R., Godfrey, M.W.: The MSR cookbook: Mining a decade of research. In: Proceedings of the International Working Conference on Mining Software Repositories, pp. 343–352 (2013). DOI 10.1109/MSR.2013.6624048

8. Hill, W.C., Hollan, J.D., Wroblewski, D.A., McCandless, T.: Edit wear and read wear. In: Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems, pp. 3–9 (1992). DOI 10.1145/142750.142751
9. Holmes, R., Walker, R.J.: Customized awareness: Recommending relevant external change events. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 465–474 (2010). DOI 10.1145/1806799.1806867
10. Kersten, M., Murphy, G.C.: Mylar: A degree-of-interest model for IDEs. In: Proceedings of the International Conference on Aspect-Oriented Software Deveopment, pp. 159–168 (2005). DOI 10.1145/1052898.1052912
11. Ko, A.J., Myers, B.A., Coblenz, M.J., Aung, H.H.: An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Trans. Software Eng. **32**(12), 971–987 (2006). DOI 10.1109/TSE.2006.116
12. Kononenko, O., Dietrich, D., Sharma, R., Holmes, R.: Automatically locating relevant programming help online. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 127–134 (2012). DOI 10.1109/VLHCC.2012.6344497
13. Mockus, A., Herbsleb, J.D.: Expertise Browser: A quantitative approach to identifying expertise. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 503–512 (2002). DOI 10.1145/581339.581401
14. Murphy-Hill, E., Jiresal, R., Murphy, G.C.: Improving software developers' fluency by recommending development environment commands. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 42:1–42:11 (2012). DOI 10.1145/2393596.2393645
15. Ricci, F., Rokach, L., Shapira, B.: Introduction to Recommender Systems Handbook. In: Ricci, F., Rokach, L., Shapira, B. (eds.) Recommender Systems Handbook, pp. 1–35. Springer, New York (2011). DOI 10.1007/978-0-387-85820-3_1
16. Robillard, M.P., Bodden, E., Kawrykow, D., Mezini, M., Ratchford, T.: Automated API property inference techniques. IEEE Trans. Software Eng. **39**(5), 613–637 (2013). DOI 10.1109/TSE.2012.63
17. Robillard, M.P., Coelho, W., Murphy, G.C.: How effective developers investigate source code: An exploratory study. IEEE Trans. Software Eng. **30**(12), 889–903 (2004). DOI 10.1109/TSE.2004.101
18. Robillard, M.P., Walker, R.J., Zimmermann, T.: Recommendation systems for software engineering. IEEE Software **27**(4), 80–86 (2010). DOI 10.1109/MS.2009.161
19. Sillito, J., Murphy, G.C., De Volder, K.: Asking and answering questions during a programming change task. IEEE Trans. Software Eng. **34**(4), 434–451 (2008). DOI 10.1109/TSE.2008.26
20. Weiser, M.: Program slicing. IEEE Trans. Software Eng. **10**(4), 352–357 (1984). DOI 10.1109/TSE.1984.5010248
21. Zimmermann, T., Weißgerber, P.: Preprocessing CVS data for fine-grained analysis. In: Proceedings of the International Workshop on Mining Software Repositories, pp. 2–6 (2004)
22. Zimmermann, T., Weißgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. IEEE Trans. Software Eng. **31**(6), 429–445 (2005). DOI 10.1109/TSE.2005.72

Part I
Techniques

Chapter 2
Basic Approaches in Recommendation Systems

Alexander Felfernig, Michael Jeran, Gerald Ninaus, Florian Reinfrank, Stefan Reiterer, and Martin Stettinger

Abstract Recommendation systems support users in finding items of interest. In this chapter, we introduce the basic approaches of collaborative filtering, content-based filtering, and knowledge-based recommendation. We first discuss principles of the underlying algorithms based on a running example. Thereafter, we provide an overview of hybrid recommendation approaches which combine basic variants. We conclude this chapter with a discussion of newer algorithmic trends, especially critiquing-based and group recommendation.

2.1 Introduction

Recommendation systems [7,33] provide suggestions for items that are of potential interest for a user. These systems are applied for answering questions such as *which book to buy*? [39], *which website to visit next*? [49], and *which financial service to choose*? [19]. In software engineering scenarios, typical questions that can be answered with the support of recommendation systems are, for example, *which software changes probably introduce a bug*? [3], *which requirements to implement in the next software release*? [25], *which stakeholders should participate in the upcoming software project*? [38], *which method calls might be useful in the current development context*? [59], *which software components (or APIs) to reuse?* [45], *which software artifacts are needed next*? [40], and *which effort estimation methods should be applied in the current project phase*? [50]. An overview of the application of different types of recommendation technologies in the software engineering context can be found in Robillard et al. [53].

A. Felfernig (✉) • M. Jeran • G. Ninaus • F. Reinfrank • S. Reiterer • M. Stettinger
Institute for Software Technology, Graz University of Technology,
Inffeldgasse 16b/2, 8010 Graz, Austria
e-mail: alexander.felfernig@ist.tugraz.at; mjeran@ist.tugraz.at; gninaus@ist.tugraz.at; florian.reinfrank@ist.tugraz.at; reiterer@ist.tugraz.at; mstettinger@ist.tugraz.at

The major goal of this book chapter is to shed light on the basic properties of the three major recommendation approaches of (1) collaborative filtering [12,26,36], (2) content-based filtering [49], and (3) knowledge-based recommendation [5, 16]. Starting with the basic algorithmic approaches, we exemplify the functioning of the algorithms and discuss criteria that help to decide which algorithm should be applied in which context.

The remainder of this chapter is organized as follows. In Sect. 2.2 we give an overview of collaborative filtering recommendation approaches. In Sect. 2.3 we introduce the basic concepts of content-based filtering. We close our discussion of basic recommendation approaches with the topic of knowledge-based recommendation (see Sect. 2.4). In Sect. 2.5, we explain example scenarios for integrating the basic recommendation algorithms into hybrid ones. Hints for practitioners interested in the development of recommender applications are given in Sect. 2.6. A short overview of further algorithmic approaches is presented in Sect. 2.7.

2.2 Collaborative Filtering

The item-set in our running examples is *software engineering-related learning material* offered, for example, on an e-learning platform (see Table 2.1). Each learning unit is additionally assigned to a set of categories, for example, the learning unit l_1 is characterized by Java and UML.

Collaborative filtering [12, 36, 56] is based on the idea of word-of-mouth promotion: the opinion of family members and friends plays a major role in personal decision making. In online scenarios (e.g., online purchasing [39]), family members and friends are replaced by the so-called *nearest neighbors* (NN) who are users with a similar preference pattern or purchasing behavior compared to the current user. Collaborative filtering (see Fig. 2.1) relies on two different types of *background data*: (1) a set of users and (2) a set of items. The relationship between users and items is primarily expressed in terms of *ratings* which are provided by users and exploited in future recommendation sessions for predicting the rating a user (in our case user U_a) would provide for a specific item. If we assume that user U_a currently interacts with a collaborative filtering recommendation system, the first step of the recommendation system is to identify the nearest neighbors (users with a similar rating behavior compared to U_a) and to extrapolate from the ratings of the similar users the rating of user U_a.

The basic procedure of collaborative filtering can best be explained based on a running example (see Table 2.2) which is taken from the software engineering domain (collaborative recommendation of learning units). Note that in this chapter we focus on the so-called memory-based approaches to collaborative filtering which—in contrast to model-based approaches—operate on uncompressed versions of the user/item matrix [4]. The two basic approaches to collaborative filtering are *user-based collaborative filtering* [36] and *item-based collaborative filtering* [54]. Both variants are predicting to which extent the active user would be interested in items which have not been rated by her/him up to now.

Table 2.1 Example set of software engineering-related learning units (LU). This set will be exploited for demonstration purposes throughout this chapter. Each of the learning units is additionally characterized by a set of categories (Java, UML, Management, Quality), for example, the learning unit l_1 is assigned to the categories Java and UML

Learning unit	Name	Java	UML	Management	Quality
l_1	Data Structures in Java	yes	yes		
l_2	Object Relational Mapping	yes	yes		
l_3	Software Architectures		yes		
l_4	Project Management		yes	yes	
l_5	Agile Processes			yes	
l_6	Object Oriented Analysis		yes	yes	
l_7	Object Oriented Design	yes	yes		
l_8	UML and the UP		yes	yes	
l_9	Class Diagrams		yes		
l_{10}	OO Complexity Metrics				yes

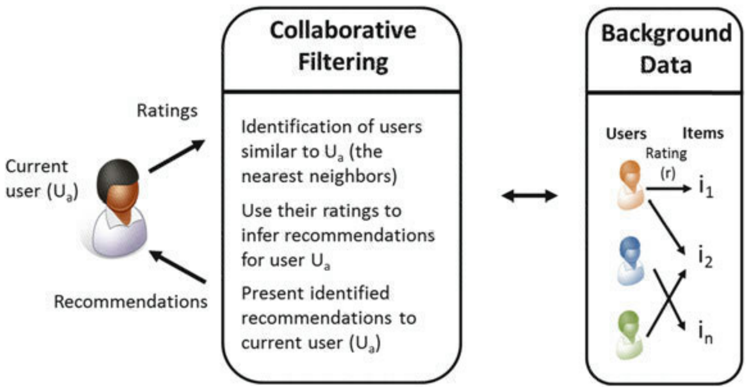
Ratings
Current user (U_a)
Recommendations
Collaborative Filtering
Identification of users similar to U_a (the nearest neighbors)
Use their ratings to infer recommendations for user U_a
Present identified recommendations to current user (U_a)
Background Data
Users
Items
Rating (r)
i_1
i_2
i_n

Fig. 2.1 Collaborative filtering (CF) dataflow. Users are rating items and receive recommendations for items based on the ratings of users with a similar rating behavior—the nearest neighbors (NN)

User-Based Collaborative Filtering. User-based collaborative filtering identifies the k-nearest neighbors of the active user—see Eq. (2.1)[1]—and, based on these nearest neighbors, calculates a prediction of the active user's rating for a specific item (learning unit). In the example of Table 2.2, user U_2 is the nearest neighbor ($k = 1$) of user U_a, based on Eq. (2.1), and his/her rating of learning unit l_3 will be taken as a prediction for the rating of U_a (rating = 3.0). The similarity between a user U_a (the current user) and another user U_x can be determined, for example, based on the Pearson correlation coefficient [33]; see Eq. (2.1), where LU_c is the set of items that have been rated by both users, r_{α,l_i} is the rating of user α for item l_i, and

[1]For simplicity we assume $k = 1$ throughout this chapter.

Table 2.2 Example collaborative filtering data structure (rating matrix): learning units (LU) versus related user ratings (we assume a rating scale of 1–5)

LU	**Name**	U_1	U_2	U_3	U_4	U_a
l_1	Data Structures in Java	5.0			4.0	
l_2	Object Relational Mapping	4.0				
l_3	Software Architectures		3.0	4.0	3.0	
l_4	Project Management		5.0	5.0		4.0
l_5	Agile Processes			3.0		
l_6	Object Oriented Analysis		4.5	4.0		4.0
l_7	Object Oriented Design	4.0				
l_8	UML and the UP		2.0			
l_9	Class Diagrams				3.0	
l_{10}	OO Complexity Metrics				5.0	3.0
average rating ($\overline{r_\alpha}$)		4.33	3.625	4.0	3.75	3.67

Table 2.3 Similarity between user U_a and the users $U_j \neq U_a$ determined based on Eq. (2.1). If the number of commonly rated items is below 2, no similarity between the two users is calculated

	U_1	U_2	U_3	U_4
U_a	–	0.97	0.70	–

$\overline{r_\alpha}$ is the average rating of user α. Similarity values resulting from the application of Eq. (2.1) can take values on a scale of $[-1, \dots, +1]$.

$$\text{similarity}(U_a, U_x) = \frac{\sum_{l_i \in LU_c} (r_{a,l_i} - \overline{r_a}) \times (r_{x,l_i} - \overline{r_x})}{\sqrt{\sum_{l_i \in LU_c} (r_{a,l_i} - \overline{r_a})^2} \times \sqrt{\sum_{l_i \in LU_c} (r_{x,l_i} - \overline{r_x})^2}} \tag{2.1}$$

The similarity values for U_a calculated based on Eq. (2.1) are shown in Table 2.3. For the purposes of our example we assume the existence of at least two items per user pair (U_i, U_j), for $i \neq j$, in order to be able to determine a similarity. This criterion holds for users U_2 and U_3.

A major challenge in the context of estimating the similarity between users is the *sparsity* of the rating matrix since users are typically providing ratings for only a very small subset of the set of offered items. For example, given a large movie dataset that contains thousands of entries, a user will typically be able to rate only a few dozens. A basic approach to tackle this problem is to take into account the number of commonly rated items in terms of a *correlation significance* [30], i.e., the higher the number of commonly rated items, the higher is the significance of

Table 2.4 User-based collaborative filtering-based recommendations (predictions) for items that have not been rated by user U_a up to now

	l_1	l_2	l_3	l_4	l_5	l_6	l_7	l_8	l_9	l_{10}
U_2	–	–	3.0	5.0	–	4.5	–	2.0	–	–
U_a	–	–	–	4.0	–	4.0	–		–	3.0
prediction(U_a, l_i)	–	–	**3.045**	–	–	–	–	**2.045**	–	–

the corresponding correlation. For further information regarding the handling of sparsity, we refer the reader to [30, 33].

The information about the set of users with a similar rating behavior compared to the current user (*NN*, the set of nearest neighbors) is the basis for predicting the rating of user U_a for an *item* that has not been rated up to now by U_a; see Eq. (2.2).

$$\text{prediction}(U_a, item) = \overline{r_a} + \frac{\sum_{U_j \in NN} \text{similarity}(U_a, U_j) \times (r_{j,item} - \overline{r_j})}{\sum_{U_j \in NN} \text{similarity}(U_a, U_j)} \tag{2.2}$$

Based on the rating of the nearest neighbor of U_a, we are able to determine a prediction for user U_a (see Table 2.4). The nearest neighbor of U_a is user U_2 (see Table 2.3). The learning units rated by U_2 but not rated by U_a are l_3 and l_8. Due to the determined predictions—Eq. (2.2)—item l_3 would be ranked higher than item l_8 in a recommendation list.

Item-Based Collaborative Filtering. In contrast to user-based collaborative filtering, item-based collaborative filtering searches for items (nearest neighbors—NN) rated by U_a that received similar ratings as items currently under investigation. In our running example, learning unit l_4 has already received a good evaluation (4.0 on a rating scale of 1–5) by U_a. The item which is most similar to l_4 and has not been rated by U_a is item l_3 (similarity$(l_3, l_4) = 0.35$). In this case, the nearest neighbor of item l_3 is l_4; this calculation is based on Eq. (2.3).

If we want to determine a recommendation based on item-based collaborative filtering, we have to determine the similarity (using the Pearson correlation coefficient) between two items l_a and l_b where U denotes the set of users who both rated l_a and l_b, r_{u,l_i} denotes the rating of user u on item l_i, and $\overline{r_{l_i}}$ is the average rating of the i-th item.

$$\text{similarity}(l_a, l_b) = \frac{\sum_{u \in U} (r_{u,l_a} - \overline{r_{l_a}}) \times (r_{u,l_b} - \overline{r_{l_b}})}{\sqrt{\sum_{u \in U} (r_{u,l_a} - \overline{r_{l_a}})^2} \times \sqrt{\sum_{u \in U} (r_{u,l_b} - \overline{r_{l_b}})^2}} \tag{2.3}$$

The information about the set of items with a similar rating pattern compared to the *item* under consideration is the basis for predicting the rating of user U_a for the *item*; see Eq. (2.4). Note that in this case *NN* represents a set of items already evaluated by U_a. Based on the assumption of $k = 1$, $prediction(U_a, l_3) = 4.0$, i.e., user U_a would rate item l_3 with 4.0.

$$\text{prediction}(U_a, item) = \frac{\sum_{it \in NN} \text{similarity}(item, it) \times r_{a,it}}{\sum_{it \in NN} \text{similarity}(item, it)} \quad (2.4)$$

For a discussion of advanced collaborative recommendation approaches, we refer the reader to Koren et al. [37] and Sarwar et al. [54].

2.3 Content-Based Filtering

Content-based filtering [49] is based on the assumption of monotonic personal interests. For example, users interested in the topic *Operating Systems* are typically not changing their interest profile from one day to another but will also be interested in the topic in the (near) future. In online scenarios, content-based recommendation approaches are applied, for example, when it comes to the recommendation of websites [49] (news items with a similar content compared to the set of already consumed news).

Content-based filtering (see Fig. 2.2) relies on two different types of background data: (1) a set of users and (2) a set of categories (or keywords) that have been assigned to (or extracted from) the available items (item descriptions). Content-based filtering recommendation systems calculate a set of items that are most similar to items already known to the current user U_a.

The basic approach of content-based filtering is to compare the content of already consumed items (e.g., a list of news articles) with new items that can potentially be recommended to the user, i.e., to find items that are similar to those already consumed (positively rated) by the user. The basis for determining such a similarity are *keywords* extracted from the item content descriptions (e.g., keywords extracted from news articles) or *categories* in the case that items have been annotated with the relevant meta-information. Readers interested in the principles of keyword extraction are referred to Jannach et al. [33]. Within the scope of this chapter we focus on content-based recommendation which exploits item categories (see Table 2.1).

Content-based filtering will now be explained based on a running example which relies on the information depicted in Tables 2.1, 2.5, and 2.6. Table 2.1 provides an overview of the relevant items and the assignments of items to categories. Table 2.5 provides information on which categories are of relevance for the different users. For example, user U_1 is primarily interested in items related to the categories *Java* and *UML*. In our running example, this information has been derived from the rating matrix depicted in Table 2.2. Since user U_a already rated the items l_4, l_6, and l_{10} (see Table 2.2), we can infer that U_a is interested in the categories UML, Management, and Quality (see Table 2.5) where items related to the category UML and Management have been evaluated two times and items related to Quality have been evaluated once.

If we are interested in an item recommendation for the user U_a we have to search for those items which are most similar to the items that have already been consumed

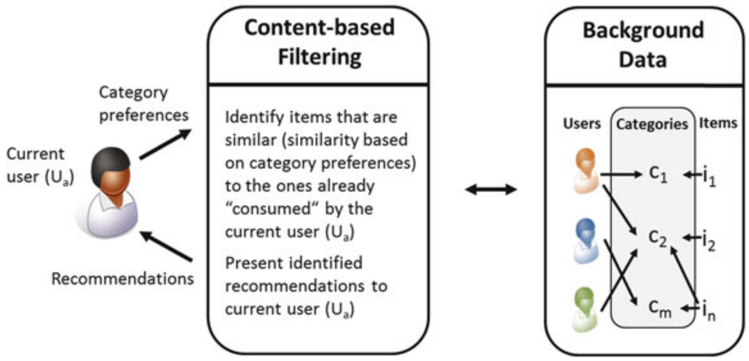
Category preferences
Current user (U$_a$)
Recommendations
Content-based Filtering
Identify items that are similar (similarity based on category preferences) to the ones already "consumed" by the current user (U$_a$)
Present identified recommendations to current user (U$_a$)
Background Data
Users
Categories
Items
c$_1$
c$_2$
c$_m$
i$_1$
i$_2$
i$_n$

Fig. 2.2 Content-based filtering (CBF) dataflow. Users rate items and receive recommendations of items similar to those that have received a good evaluation from the current user U_a

Table 2.5 Degree of interest in different categories. For example, user U_1 accessed a learning unit related to the category *Java* three times. If a user accessed an item at least once, it is inferred that the user is interested in this item

Category	U_1	U_2	U_3	U_4	U_a
Java	3 (yes)			1 (yes)	
UML	3 (yes)	4 (yes)	3 (yes)	3 (yes)	2 (yes)
Management		3 (yes)	3 (yes)		2 (yes)
Quality				1 (yes)	1 (yes)

(evaluated) by the U_a. This relies on the simple similarity metric shown in Eq. (2.5) (the Dice coefficient, which is a variation of the Jaccard coefficient that "intensively" takes into account category commonalities—see also Jannach et al. [33]). The major difference from the similarity metrics introduced in the context of collaborative filtering is that in this case similarity is measured using keywords (in contrast to ratings).

$$\text{similarity}(U_a, item) = \frac{2 \times \text{categories}(U_a) \ \cap \ \text{categories}(item)}{\text{categories}(U_a) + \text{categories}(item)} \tag{2.5}$$

2.4 Knowledge-Based Recommendation

Compared to the approaches of collaborative filtering and content-based filtering, *knowledge-based recommendation* [5, 14, 16, 23, 42] does not primarily rely on item ratings and textual item descriptions but on deep knowledge about the offered items. Such deep knowledge (semantic knowledge [16]) describes an item in more detail and thus allows for a different recommendation approach (see Table 2.7).

Table 2.6 Example of content-based filtering. User U_a has already consumed the items l_4, l_6, and l_{10}; see Table 2.2. The item most similar—see Eq. (2.5)—to the preferences of U_a is l_8 and is now the best recommendation candidate for the current user

LU	Rating of U_a	Name	Java	UML	Management	Quality	similarity(U_a, l_i)
l_1		Data Structures in Java	yes	yes			$2/5$
l_2		Object Relational Mapping	yes	yes			$2/5$
l_3		Software Architectures		yes			$2/4$
l_4	4.0	Project Management		yes	yes		–
l_5		Agile Processes			yes		$2/4$
l_6	4.0	Object Oriented Analysis		yes	yes		–
l_7		Object Oriented Design	yes	yes			$2/5$
l_8		UML and the UP		yes	yes		$\mathbf{4/5}$
l_9		Class Diagrams		yes			$2/4$
l_{10}	3.0	OO Complexity Metrics				yes	–
U_a				yes	yes	yes	

Table 2.7 Software engineering learning units (LU) described based on *deep knowledge*: obligatory vs. nonobligatory (Oblig.), duration of consumption (Dur.), recommended semester (Sem.), complexity of the learning unit (Compl.), associated topics (Topics), and average user rating (Eval.)

LU	Name	Oblig.	Dur.	Sem.	Compl.	Topics	Eval
l_1	Data Structures in Java	yes	2	2	3	Java, UML	4.5
l_2	Object Relational Mapping	yes	3	3	4	Java, UML	4.0
l_3	Software Architectures	no	3	4	3	UML	3.3
l_4	Project Management	yes	2	4	2	UML, Management	5.0
l_5	Agile Processes	no	1	3	2	Management	3.0
l_6	Object Oriented Analysis	yes	2	2	3	UML, Management	4.7
l_7	Object Oriented Design	yes	2	2	3	Java, UML	4.0
l_8	UML and the UP	no	3	3	2	UML, Management	2.0
l_9	Class Diagrams	yes	4	3	3	UML	3.0
l_{10}	OO Complexity Metrics	no	3	4	2	Quality	5.0

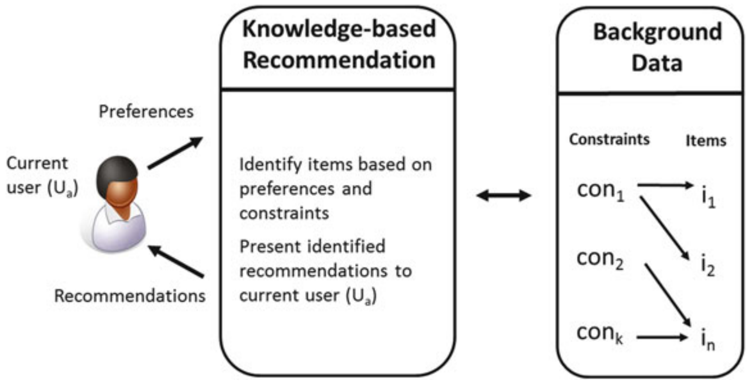
Knowledge-based Recommendation
Identify items based on preferences and constraints
Present identified recommendations to current user (U_a)
Background Data
Constraints
Items
con_1
con_2
con_k
i_1
i_2
i_n
Preferences
Current user (U_a)
Recommendations

Fig. 2.3 Knowledge-based recommendation (KBR) dataflow: users are entering their preferences and receive recommendations based on the interpretation of a set of rules (constraints)

Knowledge-based recommendation (see Fig. 2.3) relies on the following background data: (a) a set of rules (constraints) or similarity metrics and (b) a set of items. Depending on the given user requirements, rules (constraints) describe which items have to be recommended. The current user U_a articulates his/her requirements (preferences) in terms of item property specifications which are internally as well represented in terms of rules (constraints). In our example, constraints are represented solely by user requirements, no further constraint types are included (e.g., constraints that explicitly specify compatibility or incompatibility relationships). An example of such a constraint is $topics = Java$. It denotes the fact that the user is primarily interested in Java-related learning units. For a detailed discussion of further constraint types, we refer the reader to Felfernig et al. [16]. Constraints are interpreted and the resulting items are presented to the user.[2] A detailed discussion of reasoning mechanisms that are used in knowledge-based recommendation can be found, for example, in Felfernig et al. [16, 17, 22].

In order to determine a recommendation in the context of knowledge-based recommendation scenarios, a *recommendation task* has to be solved.

Definition 2.1. A *recommendation task* is a tuple (R, I) where R represents a set of user requirements and I represents a set of items (in our case: software engineering learning units $l_i \in LU$). The goal is to identify those items in I which fulfill the given user requirements (preferences).

A solution for a recommendation task (also denoted as recommendation) can be defined as follows.

[2]Knowledge-based recommendation approaches based on the determination of similarities between items will be discussed in Sect. 2.7.

Definition 2.2. A *solution for a recommendation task* (R, I) is a set $S \subseteq$ I such that $\forall l_i \in S : l_i \in \sigma_{(R)} I$ where σ is the selection operator of a conjunctive query [17], R represents a set of selection criteria (represented as constraints), and I represents an item table (see, for example, Table 2.7). If we want to restrict the set of item properties shown to the user in a result set (recommendation), we have to additionally include projection criteria π as follows: $\pi_{(attributes(I))}(\sigma_{(R)} I)$.

In our example, we show how to determine a solution for a given recommendation task based on a conjunctive query where user requirements are used as selection criteria (constraints) on an item table I. If we assume that the user requirements are represented by the set $R = \{r_1 : semester \leq 3, r_2 : topics =$ Java$\}$ and the item table I consists of the elements shown in Table 2.7, then $\pi_{(LU)}(\sigma_{(semester \leq 3 \, \wedge \, topics = \text{Java})} I) = \{l_1, l_2, l_7\}$, i.e., these three items are consistent with the given set of requirements.

Ranking Items. Up to this point we only know which items can be recommended to a user. One widespread approach to rank items is to define a utility scheme which serves as a basis for the application of multi-attribute utility theory (MAUT).[3] Alternative items can be evaluated and ranked with respect to a defined set of interest dimensions. In the domain of e-learning units, example interest dimensions of users could be *time effort* (time needed to consume the learning unit) and *quality* (quality of the learning unit). The first step to establish a MAUT scheme is to relate the interest dimensions to properties of the given set of items. A simple example of such a mapping is shown in Table 2.8. In this example, we assume that obligatory learning units (learning units that have to be consumed within the scope of a study path) trigger more time efforts than nonobligatory ones, a longer duration of a learning unit is correlated with higher time efforts, and low complexity correlates with lower time efforts. In this context, lower time efforts for a learning unit are associated with a higher utility. Furthermore, we assume that the more advanced the semester, the higher is the quality of the learning unit (e.g., in terms of education degree). The better the overall evaluation (*eval*), the higher the quality of a learning unit (e.g., in terms of the used pedagogical approach).

We are now able to determine the user-specific utility of each individual item. The calculation of *item* utilities for a specific user U_a can be based on Eq. (2.6).

$$\text{utility}(U_a, item) = \sum_{d \in Dimensions} \text{contribution}(item, d) \times \text{weight}(U_a, d) \qquad (2.6)$$

If we assume that the current user U_a assigns a weight of 0.2 to the dimension *time effort* (weight$(U_a, time\ effort) = 0.2$) and a weight of 0.8 to the dimension *quality* (weight$(U_a, quality) = 0.8$), then the user-specific utilities of the individual items (l_i) are the ones shown in Table 2.9.

[3] A detailed discussion of the application of MAUT in knowledge-based recommendation scenarios can be found in Ardissono et al. [1] and Felfernig et al. [16, 18].

Table 2.8 Contributions of item properties to the dimensions *time effort* and *quality*

Item property	**Time effort** (1–10)	**Quality** (1–10)
obligatory = yes	4	-
obligatory = no	7	-
duration = 1	10	-
duration = 2	5	-
duration = 3	1	-
duration = 4	1	-
complexity = 2	8	-
complexity = 3	5	-
complexity = 4	2	-
semester = 2	-	3
semester = 3	-	5
semester = 4	-	7
eval = 0–2	-	2
eval = >2–3	-	5
eval = >3–4	-	8
eval = >4	-	10

Table 2.9 Item-specific utility for user U_a (i.e., utility(U_a, l_i)) assuming the personal preferences for *time effort* = 0.2 and *quality* = 0.8. In this scenario, item l_4 has the highest utility for user U_a

LU	**Time effort**	**Quality**	**Utility**
l_1	14	13	$2.8+10.4=13.2$
l_2	7	13	$1.4+10.4=11.8$
l_3	13	15	$2.6+12.0=14.6$
l_4	17	17	$3.4+13.6=\mathbf{17.0}$
l_5	25	10	$5.0+8.0=13.0$
l_6	14	13	$2.8+10.4=13.2$
l_7	14	11	$2.8+8.8=11.6$
l_8	16	7	$3.2+5.6=8.8$
l_9	10	10	$2.0+8.0=10.0$
l_{10}	16	17	$3.2+13.6=16.8$

Dealing with Inconsistencies. Due to the logical nature of knowledge-based recommendation problems, we have to deal with scenarios where no solution (recommendation) can be identified for a given set of user requirements, i.e., $\sigma_{(R)} I = \emptyset$. In such situations we are interested in proposals for requirements changes such that a solution (recommendation) can be identified. For example, if a user is interested in learning units with a duration of 4 h, related to management, and a complexity level > 3, then no solution can be provided for the given set of requirements $R = \{r_1 : \textit{duration} = 4, r_2 : \textit{topics} = \text{management}, r_3 : \textit{complexity} > 3\}$.

User support in such situations can be based on the concepts of conflict detection [34] and model-based diagnosis [13, 15, 51]. A conflict (or conflict set)

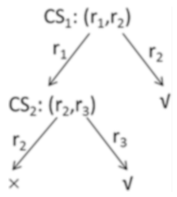

Fig. 2.4 Determination of the complete set of diagnoses (hitting sets) Δ_i for the given conflict sets $CS_1 = \{r_1, r_2\}$ and $CS_2 = \{r_2, r_3\}$: $\Delta_1 = \{r_2\}$ and $\Delta_2 = \{r_1, r_3\}$

with regard to an item set I in a given set of requirements R can be defined as follows.

Definition 2.3. A *conflict set* is a set $CS \subseteq R$ such that $\sigma_{(CS)} I = \emptyset$. CS is *minimal* if there does not exist a conflict set CS' with $CS' \subset CS$.

In our running example we are able to determine the following minimal conflict sets CS_i: $CS_1 : \{r_1, r_2\}$, $CS_2 : \{r_2, r_3\}$. We will not discuss algorithms that support the determination of minimal conflict sets but refer the reader to the work of Junker [34] who introduces a divide-and-conquer-based algorithm with a logarithmic complexity in terms of the needed number of consistency checks.

Based on the identified minimal conflict sets, we are able to determine the corresponding (minimal) diagnoses. A diagnosis for a given set of requirements which is inconsistent with the underlying item table can be defined as follows.

Definition 2.4. A *diagnosis* for a set of requirements $R = \{r_1, r_2, \ldots, r_n\}$ is a set $\Delta \subseteq R$ such that $\sigma_{(R-\Delta)} I \neq \emptyset$. A diagnosis Δ is *minimal* if there does not exist a diagnosis Δ' with $\Delta' \subset \Delta$.

In other words, a diagnosis (also called a *hitting set*) is a minimal set of requirements that have to be deleted from R such that a solution can be found for $R - \Delta$. The determination of the complete set of diagnoses for a set of requirements inconsistent with the underlying item table (the corresponding conjunctive query results in $\emptyset$) is based on the construction of hitting set trees [51]. An example of the determination of minimal diagnoses is depicted in Fig. 2.4. There are two possibilities of resolving the conflict set CS_1. If we decide to delete the requirement r_2, $\sigma_{(\{r_1, r_3\})} I \neq \emptyset$, i.e., a diagnosis has been identified ($\Delta_1 = \{r_2\}$) and—as a consequence—all CS_i have been resolved. Choosing the other alternative and resolving CS_1 by deleting r_1 does not result in a diagnosis since the conflict CS_2 is not resolved. Resolving CS_2 by deleting r_2 does not result in a minimal diagnosis, since r_2 already represents a diagnosis. The second (and last) minimal diagnosis that can be identified in our running example is $\Delta_2 = \{r_1, r_3\}$. For a detailed discussion of the underlying algorithm and analysis we refer the reader to Reiter [51]. Note that a diagnosis provides a hint to which requirements have to be changed. For a discussion of how requirement repairs (change proposals) are calculated, we refer the reader to Felfernig et al. [17].

Table 2.10 Examples of hybrid recommendation approaches ($RECS$ = set of recommenders, s = recommender-individual prediction, score = item score)

Method	Description	Example formula
weighted	predictions of individual recommenders are summed up	$\text{score}(item) = \Sigma_{rec \in RECS}\, s(item, rec)$
mixed	recommender-individual predictions are combined into one recommendation result	$\text{score}(item) = \text{zipper-function}(item, RECS)$
cascade	the predictions of one recommender are used as input for the next recommender	$\text{score}(item) = \text{score}(item, rec_n)$ $\text{score}(item, rec_i) = \begin{cases} s(item, rec_1)\ , & \text{if } i = 1 \\ s(item, rec_i) \times \text{score}(item, rec_{i-1})\ , & \text{otherwise.} \end{cases}$

2.5 Hybrid Recommendations

After having discussed the three basic recommendation approaches of collaborative filtering, content-based filtering, and knowledge-based recommendation, we will now present some possibilities to combine these basic types.

The motivation for hybrid recommendations is the opportunity to achieve a better accuracy [6]. There are different approaches to evaluate the accuracy of recommendation algorithms. These approaches (see also Avazpour et al. [2] and Tosun Mısırlı et al. [58] in Chaps. 10 and 13, respectively) can be categorized into *predictive accuracy metrics* such as the mean absolute error (MAE), *classification accuracy metrics* such as precision and recall, and *rank accuracy metrics* such as Kendall's Tau. For a discussion of accuracy metrics we refer the reader also to Gunawardana and Shani [28] and Jannach et al. [33].

We now take a look at example design types of hybrid recommendation approaches [6, 33] which are *weighted*, *mixed*, and *cascade* (see Table 2.10). These approaches will be explained on the basis of our running example. The basic assumption in the following is that individual recommendation approaches return a list of *five* recommended items where each item has an assigned (recommender-individual) prediction out of $\{1.0, 2.0, 3.0, 4.0, 5.0\}$. For a more detailed discussion of hybridization strategies, we refer the reader to Burke [6] and Jannach et al. [33].

Weighted. Weighted hybrid recommendation is based on the idea of deriving recommendations by combining the results (predictions) computed by individual recommenders. A corresponding example is depicted in Table 2.11 where the

Table 2.11 Example of *weighted* hybrid recommendation: individual predictions are integrated into one score. Item l_8 receives the best overall score (9.0)

Items	l_1	l_2	l_3	l_4	l_5	l_6	l_7	l_8	l_9	l_{10}
$s(l_i, \textit{collaborative filtering})$	1.0	3.0	–	5.0	–	2.0	–	4.0	–	–
$s(l_i, \textit{content-based filtering})$	–	1.0	2.0	–	–	3.0	4.0	5.0	–	–
score(l_i)	1.0	4.0	2.0	5.0	0.0	5.0	4.0	**9.0**	0.0	0.0
ranking(l_i)	7	4	6	2	8	3	5	**1**	9	10

Table 2.12 Example of *mixed* hybrid recommendation. Individual predictions are integrated into one score conform the zipper principle (best collaborative filtering prediction receives score = 10, best content-based filtering prediction receives score = 9 and so forth)

Items	l_1	l_2	l_3	l_4	l_5	l_6	l_7	l_8	l_9	l_{10}
$s(l_i, \textit{collaborative filtering})$	1.0	3.0	–	5.0	–	2.0	–	4.0	–	–
$s(l_i, \textit{content-based filtering})$	–	1.0	2.0	–	–	3.0	4.0	5.0	–	–
score(l_i)	4.0	8.0	5.0	**10.0**	0.0	6.0	7.0	9.0	0.0	0.0
ranking(l_i)	7	3	6	**1**	8	5	4	2	9	10

individual item scores of a collaborative and a content-based recommender are summed up. Item l_8 receives the highest overall score (9.0) and is ranked highest by the weighted hybrid recommender.[4]

Mixed. Mixed hybrid recommendation is based on the idea that predictions of individual recommenders are shown in one integrated result. For example, the results of a collaborative filtering and a content-based recommender can be ranked as sketched in Table 2.12. Item scores can be determined, for example, on the basis of the zipper principle, i.e., the item with highest collaborative filtering prediction value receives the highest overall score (10.0), the item with best content-based filtering prediction value receives the second best overall score, and so forth.

Cascade. The basic idea of cascade-based hybridization is that recommenders in a pipe of recommenders exploit the recommendation of the upstream recommender as a basis for deriving their own recommendation. The knowledge-based recommendation approach presented in Sect. 2.4 is an example of a cascade-based hybrid recommendation approach. First, items that are consistent with the given requirements are preselected by a conjunctive query Q. We can assume, for example, that $s(item, Q) = 1.0$ if the item has been selected and $s(item, Q) = 0.0$ if the item has not been selected. In our case, the set of requirements $R = \{r_1 : semester \leq 3, r_2 : topics = Java\}$ in the running example leads to the selection of the items $\{l_1, l_2, l_7\}$. Thereafter, these items are ranked conform to

[4]If two or more items have the same overall score, a possibility is to force a decision by lot; where needed, this approach can also be applied by other hybrid recommendation approaches.

their utility for the current user (utility-based ranking U). The utility-based ranking U would determine the item order utility(l_1) $>$ utility(l_2) $>$ utility(l_7) assuming that the current user assigns a weight of 0.8 to the interest dimension *quality* (weight(U_a,*quality*) $= 0.8$) and a weight of 0.2 to the interest dimensions *time effort* (weight(U_a,*time effort*) $= 0.2$). In this example the recommender Q is the first one and the results of Q are forwarded to the utility-based recommender.

Other examples of hybrid recommendation approaches include the following [6]. *Switching* denotes an approach where—depending on the current situation—a specific recommendation approach is chosen. For example, if a user has a low level of product knowledge, then a critiquing-based recommender will be chosen (see Sect. 2.7). Vice versa, if the user is an expert, an interface will be provided where the user is enabled to explicitly state his/her preferences on a detailed level. *Feature combination* denotes an approach where different data sources are exploited by a single recommender. For example, a recommendation algorithm could exploit semantic item knowledge in combination with item ratings (see Table 2.7). For an in-depth discussion of hybrid recommenders, we refer the reader to Burke [6] and Jannach et al. [33].

2.6 Hints for Practitioners

In this section we provide several hints for practitioners who are interested in developing recommendation systems.

2.6.1 Usage of Algorithms

The three basic approaches of collaborative filtering, content-based filtering, and knowledge-based recommendation exploit different sources of recommendation knowledge and have different strengths and weaknesses (see Table 2.13). Collaborative filtering (CF) and content-based filtering (CBF) are easy to set up (only basic item information is needed, e.g., item name and picture), whereas knowledge-based recommendation requires a more detailed specification of item properties (and in many cases also additional constraints). Both CF and CBF are more adaptive in the sense that new ratings are automatically taken into account in future activations of the recommendation algorithm. In contrast, utility schemes in knowledge-based recommendation (see, for example, Table 2.9) have to be adapted manually (if no additional learning support is available [21]).

Serendipity effects are interpreted as a kind of accident of being confronted with something useful although no related search has been triggered by the user. They can primarily be achieved when using CF approaches. Due to the fact that content-based filtering does not take into account the preferences (ratings) of other users, no such effects can be achieved. Achieving serendipity effects for the users based on KBR is possible in principle, however, restricted to and depending on

Table 2.13 Summary of the strengths and weaknesses of collaborative filtering (CF), content-based filtering (CBF), and knowledge-based recommendation (KBR)

Property	CF	CBF	KBR
easy setup	yes	yes	no
adaptivity	yes	yes	no
serendipity effects	yes	no	no
ramp-up problem	yes	yes	no
transparency	no	no	yes
high-involvement items	no	no	yes

the creativity of the knowledge engineer (who is able to foresee such effects when defining recommendation rules). The *ramp-up problem* (also called the cold start problem) denotes a situation where there is the need to provide initial rating data before the algorithm is able to determine reasonable recommendations. Ramp-up problems exist with both CF and CBF: in CF users have to rate a set of items before the algorithm is able to determine the nearest neighbors; in CBF, the user has to specify interesting/relevant items before the algorithm is able to determine items that are similar to those that have already been rated by the user.

Finally, transparency denotes the degree to which recommendations can be explained to users. Explanations in CF systems solely rely on the interpretation of the relationship to nearest neighbors, for example, *users who purchased item X also purchased item Y*. CBF algorithms explain their recommendations in terms of the similarity of the recommended item to items already purchased by the user: *we recommend Y since you already purchased X which is quite similar to Y*. Finally—due to the fact that they rely on deep knowledge—KBR is able to provide deep explanations which take into account semantic item knowledge. An example of such an explanation is diagnoses that explain the reasons as to why a certain set of requirements does not allow the calculation of a solution. Other types of explanations exist: why a certain item has been included in the recommendation and why a certain question has been asked to the user [16, 24].

Typically, CF and CBF algorithms are used for recommending low-involvement items[5] such as movies, books, and news articles. In contrast, knowledge-based recommender functionalities are used for the recommendation of high-involvement items such as financial services, cars, digital cameras, and apartments. In the latter case, ratings are provided with a low frequency which makes these domains less accessible to CF and CBF approaches. For example, user preferences regarding a car could significantly change within a couple of years without being detected by the recommender system, whereas such preference shifts are detected by collaborative and content-based recommendation approaches due to the fact that purchases occur more frequently and—as a consequence—related ratings are available for the

[5]The impact of a wrong decision (selection) is rather low, therefore users invest less evaluation effort in a purchase situation.

recommender system. For an overview of heuristics and rules related to the selection of recommendation approaches, we refer the reader to Burke and Ramezani [9].

2.6.2 Recommendation Environments

Recommendation is an artificial intelligence (AI) technology successfully applied in different commercial contexts [20]. As recommendation algorithms and heuristics are regarded as a major intellectual property of a company, recommender systems are often not developed on the basis of standard solutions but are rather based on proprietary solutions that are tailored to the specific situation of the company. Despite this situation, there exist a few recommendation environments that can be exploited for the development of different recommender applications.

Strands is a company that provides recommendation technologies covering the whole range of collaborative, content-based, and knowledge-based recommendation approaches. MyMediaLite is an open-source library that can be used for the development of collaborative filtering-based recommender systems. LensKit [11] is an open-source toolkit that supports the development and evaluation of recommender systems—specifically it includes implementations of different collaborative filtering algorithms. A related development is MovieLens which is a noncommercial movie recommendation platform. The MovieLens dataset (user $\times$ item ratings) is publicly available and popular dataset for evaluating new algorithmic developments. Apache Mahout is a machine learning environment that also includes recommendation functionalities such as user-based and item-based collaborative filtering.

Open-source constraint libraries such as Choco and Jacop can be exploited for the implementation of knowledge-based recommender applications. WeeVis is a Wiki-based environment for the development of knowledge-based recommender applications—resulting recommender applications can be deployed on different handheld platforms such as iOS, Android, and Windows 8. Finally, Choicla is a group recommendation platform that allows the definition and execution of group recommendation tasks (see Sect. 2.7).

2.7 Further Algorithmic Approaches

We examine two further algorithmic approaches here: general critiquing-based recommendations and group recommendations.

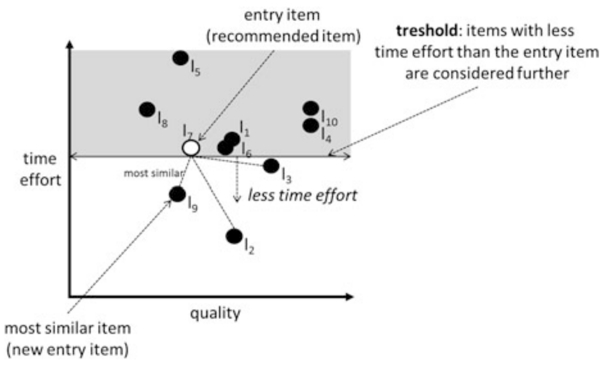

Fig. 2.5 Example of a critiquing scenario. The entry item I_7 is shown to the user. The user specifies the critique "less time effort." The new entry item is I_9 since it is consistent with the critique and the item most similar to I_7

2.7.1 Critiquing-Based Recommendation

There are two basic approaches to support item identification in the context of knowledge-based recommendation.

First, *search-based* approaches require the explicit specification of search criteria and the recommendation algorithm is in charge of identifying a set of corresponding recommendations [16,57] (see also Sect. 2.4). If no solution can be found for a given set of requirements, the recommendation engine determines diagnoses that indicate potential changes such that a solution (recommendation) can be identified. Second, *navigation-based* approaches support the navigation in the item space where in each iteration a reference item is presented to the user and the user either accepts the (recommended) item or searches for different solutions by specifying *critiques*. Critiques are simple criteria that are used for determining new recommendations that take into account the (changed) preferences of the current user. Examples of such critiques in the context of our running example are *less time efforts* and *higher quality* (see Fig. 2.5). Critiquing-based recommendation systems are useful in situations where users are not experts in the item domain and prefer to specify their requirements on the level of critiques [35]. If users are knowledgeable in the item domain, the application of search-based approaches makes more sense. For an in-depth discussion of different variants of critiquing-based recommendation, we refer the reader to [8, 10, 27, 41, 46, 52].

2.7.2 Group Recommendation

Due to the increasing popularity of social platforms and online communities, group recommendation systems are becoming an increasingly important technology

Table 2.14 Example of group recommendation: selection of a learning unit for a group. The recommendation (l_7) is based on the *least misery* heuristic

Items	l_1	l_2	l_3	l_4	l_5	l_6	l_7	l_8	l_9	l_{10}
alex	1.0	3.0	1.0	5.0	4.0	2.0	4.0	2.0	1.0	4.0
dorothy	5.0	1.0	2.0	1.0	4.0	3.0	4.0	2.0	2.0	3.0
peter	2.0	4.0	2.0	5.0	3.0	5.0	4.0	3.0	2.0	2.0
ann	3.0	4.0	5.0	2.0	1.0	1.0	3.0	3.0	3.0	4.0
least misery	1.0	1.0	1.0	1.0	1.0	1.0	**3.0**	2.0	1.0	2.0

[29, 44]. Example application domains of group recommendation technologies include tourism [47] (e.g., *which hotels or tourist destinations should be visited by a group?*) and interactive television [43] (*which sequence of television programs will be accepted by a group?*). In the majority, group recommendation algorithms are related to simple items such as hotels, tourist destinations, and television programs. The application of group recommendation in the context of our running example is shown in Table 2.14 (selection of a learning unit for a group).

The group recommendation task is to figure out a recommendation that will be accepted by the whole group. The group decision heuristics applied in the context is *least misery* which returns the lowest voting for alternative l_i as group recommendation. For example, the *least misery* value for alternative l_7 is 3.0 which is the highest value of all possible alternatives, i.e., the first recommendation for the group is l_7. Other examples of group recommendation heuristics are *most pleasure* (the group recommendation is the item with the most individual votes) and *majority voting* (the voting for an individual solution is defined by the majority of individual user votes: the group recommendation is the item with the highest majority value). Group recommendation technologies for high-involvement items (see Sect. 2.6) are the exception of the rule [e.g., 31, 55]. First applications of group recommendation technologies in the software engineering context are reported in Felfernig et al. [25]. An in-depth discussion of different types of group recommendation algorithms can be found in O'Connor et al. [48], Jameson and Smyth [32], and Masthoff [44].

2.8 Conclusion

This chapter provides an introduction to the recommendation approaches of collaborative filtering, content-based filtering, knowledge-based recommendation, and different hybrid variants thereof. While collaborative filtering-based approaches exploit ratings of nearest neighbors, content-based filtering exploits categories and/or extracted keywords for determining recommendations. Knowledge-based recommenders should be used, for example, for products where there is a need to encode the recommendation knowledge in terms of constraints. Beside algorithmic approaches, we discussed criteria to be taken into account when deciding about which recommendation technology to use in a certain application context. Furthermore, we provided an overview of environments that can be exploited for recommender application development.

References

1. Ardissono, L., Felfernig, A., Friedrich, G., Goy, A., Jannach, D., Petrone, G., Schäfer, R., Zanker, M.: A framework for the development of personalized, distributed web-based configuration systems. AI Mag. **24**(3), 93–108 (2003)
2. Avazpour, I., Pitakrat, T., Grunske, L., Grundy, J.: Dimensions and metrics for evaluating recommendation systems. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering, Chap. 10. Springer, New York (2014)
3. Bachwani, R.: Preventing and diagnosing software upgrade failures. Ph.D. thesis, Rutgers University (2012)
4. Billsus, D., Pazzani, M.: Learning collaborative information filters. In: Proceedings of the International Conference on Machine Learning, pp. 46–54 (1998)
5. Burke, R.: Knowledge-based recommender systems. Encyclopedia Libr. Inform. Sci. **69**(32), 180–200 (2000)
6. Burke, R.: Hybrid recommender systems: Survey and experiments. User Model. User-Adapt. Interact. **12**(4), 331–370 (2002). DOI 10.1023/A:1021240730564
7. Burke, R., Felfernig, A., Goeker, M.: Recommender systems: An overview. AI Mag. **32**(3), 13–18 (2011)
8. Burke, R., Hammond, K., Yound, B.: The FindMe approach to assisted browsing. IEEE Expert **12**(4), 32–40 (1997). DOI 10.1109/64.608186
9. Burke, R., Ramezani, M.: Matching recommendation technologies and domains. In: Ricci, F., Rokach, L., Shapira, B., Kantor, P.B. (eds.) Recommender Systems Handbook, pp. 367–386. Springer, New York (2011). DOI 10.1007/978-0-387-85820-3_11
10. Chen, L., Pu, P.: Critiquing-based recommenders: Survey and emerging trends. User Model. User-Adapt. Interact. **22**(1–2), 125–150 (2012). DOI 10.1007/s11257-011-9108-6
11. Ekstrand, M.D., Ludwig, M., Kolb, J., Riedl, J.: LensKit: A modular recommender framework. In: Proceedings of the ACM Conference on Recommender Systems, pp. 349–350 (2011a). DOI 10.1145/2043932.2044001
12. Ekstrand, M.D., Riedl, J.T., Konstan, J.A.: Collaborative filtering recommender systems. Found. Trends Hum. Comput. Interact. **4**(2), 81–173 (2011b). DOI 10.1561/1100000009
13. Falkner, A., Felfernig, A., Haag, A.: Recommendation technologies for configurable products. AI Mag. **32**(3), 99–108 (2011)
14. Felfernig, A., Friedrich, G., Gula, B., Hitz, M., Kruggel, T., Melcher, R., Riepan, D., Strauss, S., Teppan, E., Vitouch, O.: Persuasive recommendation: Serial position effects in knowledge-based recommender systems. In: Proceedings of the International Conference of Persuasive Technology, *Lecture Notes in Computer Science*, vol. 4744, pp. 283–294 (2007a). DOI 10.1007/978-3-540-77006-0_34
15. Felfernig, A., Friedrich, G., Jannach, D., Stumptner, M.: Consistency-based diagnosis of configuration knowledge bases. Artif. Intell. **152**(2), 213–234 (2004). DOI 10.1016/S0004-3702(03)00117-6
16. Felfernig, A., Friedrich, G., Jannach, D., Zanker, M.: An integrated environment for the development of knowledge-based recommender applications. Int. J. Electron. Commerce **11**(2), 11–34 (2006a). DOI 10.2753/JEC1086-4415110201
17. Felfernig, A., Friedrich, G., Schubert, M., Mandl, M., Mairitsch, M., Teppan, E.: Plausible repairs for inconsistent requirements. In: Proceedings of the International Joint Conference on Artificial Intelligence, pp. 791–796 (2009)

18. Felfernig, A., Gula, B., Leitner, G., Maier, M., Melcher, R., Teppan, E.: Persuasion in knowledge-based recommendation. In: Proceedings of the International Conference on Persuasive Technology, *Lecture Notes in Computer Science*, vol. 5033, pp. 71–82 (2008). DOI 10.1007/978-3-540-68504-3_7
19. Felfernig, A., Isak, K., Szabo, K., Zachar, P.: The VITA financial services sales support environment. In: Proceedings of the Innovative Applications of Artificial Intelligence Conference, pp. 1692–1699 (2007b)
20. Felfernig, A., Jeran, M., Ninaus, G., Reinfrank, F., Reiterer, S.: Toward the next generation of recommender systems: Applications and research challenges. In: Multimedia Services in Intelligent Environments: Advances in Recommender Systems, *Smart Innovation, Systems and Technologies*, vol. 24, pp. 81–98. Springer, New York (2013a). DOI 10.1007/978-3-319-00372-6_5
21. Felfernig, A., Ninaus, G., Grabner, H., Reinfrank, F., Weninger, L., Pagano, D., Maalej, W.: An overview of recommender systems in requirements engineering. In: Managing Requirements Knowledge, Chap. 14, pp. 315–332. Springer, New York (2013b). DOI 10.1007/978-3-642-34419-0_14
22. Felfernig, A., Schubert, M., Reiterer, S.: Personalized diagnosis for over-constrained problems. In: Proceedings of the International Joint Conference on Artificial Intelligence, pp. 1990–1996 (2013c)
23. Felfernig, A., Shchekotykhin, K.: Debugging user interface descriptions of knowledge-based recommender applications. In: Proceedings of the International Conference on Intelligent User Interfaces, pp. 234–241 (2006). DOI 10.1145/1111449.1111499
24. Felfernig, A., Teppan, E., Gula, B.: Knowledge-based recommender technologies for marketing and sales. Int. J. Pattern Recogn. Artif. Intell. **21**(2), 333–354 (2006b). DOI 10.1142/S0218001407005417
25. Felfernig, A., Zehentner, C., Ninaus, G., Grabner, H., Maaleij, W., Pagano, D., Weninger, L., Reinfrank, F.: Group decision support for requirements negotiation. In: Advances in User Modeling, no. 7138 in Lecture Notes in Computer Science, pp. 105–116 (2012). DOI 10.1007/978-3-642-28509-7_11
26. Goldberg, D., Nichols, D., Oki, B., Terry, D.: Using collaborative filtering to weave an information tapestry. Comm. ACM **35**(12), 61–70 (1992). DOI 10.1145/138859.138867
27. Grasch, P., Felfernig, A., Reinfrank, F.: ReComment: Towards critiquing-based recommendation with speech interaction. In: Proceedings of the ACM Conference on Recommender Systems pp. 157–164 (2013)
28. Gunawardana, A., Shani, G.: A survey of accuracy evaluation metrics of recommendation tasks. J. Mach. Learn. Res. **10**, 2935–2962 (2009)
29. Hennig-Thurau, T., Marchand, A., Marx, P.: Can automated group recommender systems help consumers make better choices? J. Market. **76**(5), 89–109 (2012)
30. Herlocker, J., Konstan, J., Borchers, A., Riedl, J.: An algorithmic framework for performing collaborative filtering. In: Proceedings of the ACM SIGIR International Conference on Research and Development in Information Retrieval, pp. 230–237 (1999). DOI >10.1145/312624.312682
31. Jameson, A.: More than the sum of its members: Challenges for group recommender systems. In: Proceedings of the Working Conference on Advanced Visual Interfaces, pp. 48–54 (2004). DOI 10.1145/989863.989869
32. Jameson, A., Smyth, B.: Recommendation to groups. In: Brusilovsky, P., Kobsa, A., Nejdl, W. (eds.) The Adaptive Web: Methods and Strategies of Web Personalization, *Lecture Notes in Computer Science*, vol. 4321, Chap. 20, pp. 596–627. Springer, New York (2007). DOI 10.1007/978-3-540-72079-9_20
33. Jannach, D., Zanker, M., Felfernig, A., Friedrich, G.: Recommender Systems: An Introduction. Cambridge University Press, Cambridge (2010)
34. Junker, U.: QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In: Proceedings of the National Conference on Artifical Intelligence, pp. 167–172 (2004)

35. Knijnenburg, B., Reijmer, N., Willemsen, M.: Each to his own: How different users call for different interaction methods in recommender systems. In: Proceedings of the ACM Conference on Recommender Systems, pp. 141–148 (2011). DOI 10.1145/2043932.2043960
36. Konstan, J.A., Miller, B.N., Maltz, D., Herlocker, J.L., Gordon, L.R., Riedl, J.: GroupLens: Applying collaborative filtering to Usenet news. Comm. ACM **40**(3), 77–87 (1997). DOI 10.1145/245108.245126
37. Koren, Y., Bell, R., Volinsky, C.: Matrix factorization techniques for recommender systems. Computer **42**(8), 30–37 (2009). DOI 10.1109/MC.2009.263
38. Lim, S., Quercia, D., Finkelstein, A.: StakeNet: Using social networks to analyse the stakeholders of large-scale software projects. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 295–304 (2010). DOI 10.1145/1806799.1806844
39. Linden, G., Smith, B., York, J.: Amazon.com recommendations: Item-to-item collaborative filtering. IEEE Internet Comput. **7**(1), 76–80 (2003). DOI 10.1109/MIC.2003.1167344
40. Maalej, W., Sahm, A.: Assisting engineers in switching artifacts by using task semantic and interaction history. In: Proceedings of the International Workshop on Recommendation Systems for Software Engineering, pp. 59–63 (2010). DOI 10.1145/1808920.1808935
41. Mandl, M., Felfernig, A.: Improving the performance of unit critiquing. In: Proceedings of the International Conference on User Modeling, Adaptation, and Personalization, pp. 176–187 (2012). DOI 10.1007/978-3-642-31454-4_15
42. Mandl, M., Felfernig, A., Teppan, E., Schubert, M.: Consumer decision making in knowledge-based recommendation. J. Intell. Inform. Syst. **37**(1), 1–22 (2010). DOI 10.1007/s10844-010-0134-3
43. Masthoff, J.: Group modeling: Selecting a sequence of television items to suit a group of viewers. User Model. User-Adapt. Interact. **14**(1), 37–85 (2004). DOI 10.1023/B:USER.0000010138.79319.fd
44. Masthoff, J.: Group recommender systems: Combining individual models. In: Ricci, F., Rokach, L., Shapira, B., Kantor, P. (eds.) Recommender Systems Handbook, Chap. 21, pp. 677–702. Springer, New York (2011). DOI 10.1007/978-0-387-85820-3_21
45. McCarey, F., Ó Cinnéide, M., Kushmerick, N.: RASCAL: A recommender agent for agile reuse. Artif. Intell. Rev. **24**(3–4), 253–276 (2005). DOI 10.1007/s10462-005-9012-8
46. McCarthy, K., Reilly, J., McGinty, L., Smyth, B.: On the dynamic generation of compound critiques in conversational recommender systems. In: Proceedings of the International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems, *Lecture Notes in Computer Science*, vol. 3137, pp. 176–184 (2004)
47. McCarthy, K., Salamo, M., Coyle, L., McGinty, L., Smyth, B., Nixon, P.: Group recommender systems: A critiquing based approach. In: Proceedings of the International Conference on Intelligent User Interfaces, pp. 267–269 (2006). DOI 10.1145/1111449.1111506
48. O'Connor, M., Cosley, D., Konstan, J., Riedl, J.: PolyLens: A recommender system for groups of users. In: Proceedings of the European Conference on Computer Supported Cooperative Work, pp. 199–218 (2001). DOI 10.1007/0-306-48019-0_11
49. Pazzani, M., Billsus, D.: Learning and revising user profiles: The identification of interesting web sites. Mach. Learn. **27**(3), 313–331 (1997). DOI 10.1023/A:1007369909943
50. Peischl, B., Zanker, M., Nica, M., Schmid, W.: Constraint-based recommendation for software project effort estimation. J. Emerg. Tech. Web Intell. **2**(4), 282–290 (2010). DOI 10.4304/jetwi.2.4.282-290
51. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. **32**(1), 57–95 (1987). DOI 10.1016/0004-3702(87)90062-2
52. Ricci, F., Nguyen, Q.: Acqiring and revising preferences in a critiquing-based mobile recommender systems. IEEE Intell. Syst. **22**(3), 22–29 (2007). DOI 10.1109/MIS.2007.43
53. Robillard, M.P., Walker, R.J., Zimmermann, T.: Recommendation systems for software engineering. IEEE Software **27**(4), 80–86 (2010). DOI 10.1109/MS.2009.161
54. Sarwar, B., Karypis, G., Konstan, J., Riedl, J.: Item-based collaborative filtering recommendation algorithms. In: Proceedings of the International Conference on the World Wide Web, pp. 285–295 (2001). DOI 10.1145/371920.372071

55. Stettinger, M., Ninaus, G., Jeran, M., Reinfrank, F., Reiterer, S.: WE-DECIDE: A decision support environment for groups of users. In: Proceedings of the International Conference on Industrial, Engineering, and Other Applications of Applied Intelligent Systems, pp. 382–391 (2013). DOI 10.1007/978-3-642-38577-3_39
56. Takács, G., Pilászy, I., Németh, B., Tikk, D.: Scalable collaborative filtering approaches for large recommender systems. J. Mach. Learn. Res. **10**, 623–656 (2009)
57. Tiihonen, J., Felfernig, A.: Towards recommending configurable offerings. Int. J. Mass Customization **3**(4), 389–406 (2010). DOI 10.1504/IJMASSC.2010.037652
58. Tosun Mısırlı, A., Bener, A., Çağlayan, B., Çalıklı, G., Turhan, B.: Field studies: A methodology for construction and evaluation of recommendation systems in software engineering. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering, Chap. 13. Springer, New York (2014)
59. Tsunoda, M., Kakimoto, T., Ohsugi, N., Monden, A., Matsumoto, K.: Javawock: A Java class recommender system based on collaborative filtering. In: Proceedings of the International Conference on Software Engineering and Knowledge Engineering, pp. 491–497 (2005)

Chapter 3
Data Mining

A Tutorial

Tim Menzies

Abstract Recommendation systems find and summarize patterns in the structure of some data or in how we visit that data. Such summarizing can be implemented by *data mining* algorithms. While the rest of this book focuses specifically on recommendation systems in software engineering, this chapter provides a more general tutorial introduction to data mining.

3.1 Introduction

A recommendation system finds and summarizes patterns in some structure (and those patterns can include how, in the past, users have explored that structure). One way to find those patterns is to use *data mining algorithms*.

The rest of this book focuses specifically on recommendation systems in software engineering (RSSEs). But, just to get us started, this chapter is a tutorial introduction to data mining algorithms:

- This chapter covers C4.5, K-means, Apriori, AdaBoost, kNN, naive Bayesian, CART, and SVM.
- Also mentioned will be random forests, DBScan, canopy clustering, mini-batch K-means, simple single-pass K-means, GenIc, the Fayyad–Irani discretizer, InfoGain, TF–IDF, PDDP, PCA, and LSI.
- There will also be some discussion on how to use the above for text mining.

T. Menzies (✉)
Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV, USA
e-mail: tim@menzies.us

Data mining is a very active field. Hence, any summary of that field must be incomplete. Therefore this chapter ends with some suggested readings for those who want to read more about this exciting field.

Every learning method is biased in some way, and it is important to understand those biases. Accordingly, it is important to understand two biases of this chapter. Firstly, it will be the view of this chapter that *it is a mistake* to use data miners as black box tools. In that black box view, the learners are applied without any comprehension of their internal workings. To avoid that mistake, it is useful for data mining novices to reflect on these algorithms, as a menu of design options can be *mixed and matched and mashed-up* as required. Accordingly, where appropriate, this chapter will take care to show how parts of one learner might be used for another. Secondly, this chapter discusses *newer methods* such as CLIFF, WHERE, W2, and the QUICK active learner: work of the author, his collaborators, and/or his graduate students. Caveat emptor!

3.2 Different Learners for Different Data

Let us start at the very beginning (a very good place to start). When you read you begin with A-B-C. When you mine, you begin with data.

Different kinds of data miners work best of different kinds of data. Such data may be viewed as *tables* of *examples*:

- Tables have one column per *feature* and one row per example.
- The columns may be *numeric* (have numbers) or *discrete* (contain symbols).
- Also, some columns are *goals* (things we want to predict using the other columns).
- Finally, columns may contain *missing values*.

For example, in *text mining*, where there is one column per word and one row per document, the columns contain many missing values (since not all words appear in all documents) and there may be hundreds of thousands of columns.

While text mining applications can have many columns, *Big Data* applications can have any number of columns and millions to billions of rows. For such very large datasets, a complete analysis may be impossible. Hence, these might be sampled probabilistically (e.g., using the naive Bayesian algorithm discussed below).

On the other hand, when there are very few rows, data mining may fail since there are too few examples to support summarization. For such sparse tables, k-nearest neighbors (kNN) may be best. kNN makes conclusions about new examples by looking at their neighborhood in the space of old examples. Hence, kNN only needs a few (or even only one) similar examples to make conclusions.

If a table has no goal columns, then this is an *unsupervised* learning problem that might be addressed by (say) finding clusters of similar rows using, say, K-means or expectation maximization. An alternate approach, taken by the Apriori association rule learner, is to assume that every column is a goal and to look for what combinations of any values predict for any combination of any other.

If a table has one goal, then this is a *supervised* learning problem where the task is to find combinations of values from the other columns that predict for the goal values. Note that for datasets with one discrete goal feature, it is common to call that goal the *class* of the dataset.

For example, here is a table of data for a *simple data mining* problem:

outlook	temp	humidity	windy	play?
overcast	64	65	TRUE	yes
overcast	72	90	TRUE	yes
overcast	81	75	FALSE	yes
overcast	83	86	FALSE	yes
rainy	65	70	TRUE	no
rainy	71	91	TRUE	no
rainy	68	80	FALSE	yes
rainy	70	96	FALSE	yes
rainy	75	80	FALSE	yes
sunny	69	70	FALSE	yes
sunny	72	95	FALSE	no
sunny	75	70	TRUE	yes
sunny	80	90	TRUE	no
sunny	85	85	FALSE	no

In this table, we are trying to predict for the goal of `play?`, given a record of the weather. Each row is one example where we did or did not play golf (and the goal of data mining is to find what weather predicts for playing golf).

Note that `temp` and `humidity` are numeric columns and there are no missing values.

Such simple tables are characterized by just a few columns and not many rows (say, dozens to thousands). Traditionally, such simple data mining problems have been explored by C4.5 and CART. However, with some clever sampling of the data, it is possible to scale these traditional learners to Big Data problems [7, 8].

3.3 Association Rules

The *Apriori* learner seeks association rules, i.e., sets of ranges that are often found in the same row. First published in the early 1990s [1], Apriori is a classic recommendation algorithm for assisting shopper. It was initially developed to answer the *shopping basket* problem, i.e., "if a customer buys X, what else might they buy?"

Apriori can be used by, say, an online book store to make recommendations about what else a user might like to see. To use Apriori, all numeric values must be *discretized*, i.e., the numeric ranges replaced with a small number of discrete symbols. Later in this chapter, we discuss several ways to perform discretization but an *X% chop* is sometimes as good as anything else. In this approach, numeric feature values are sorted and then divided into X equal-sized bins. A standard default is $X = 10$, but the above table is very small, so we will use $X = 2$ to generate:

outlook	temp	humidity	windy	play?
overcast	over 73.5	over 82.5	FALSE	yes
overcast	up to 73.5	up to 82.5	TRUE	yes
overcast	up to 73.5	over 82.5	TRUE	yes
overcast	over 73.5	up to 82.5	FALSE	yes
rainy	over 73.5	up to 82.5	FALSE	yes
rainy	up to 73.5	over 82.5	TRUE	no
rainy	up to 73.5	up to 82.5	TRUE	no
rainy	up to 73.5	over 82.5	FALSE	yes
rainy	up to 73.5	up to 82.5	FALSE	yes
sunny	over 73.5	over 82.5	TRUE	no
sunny	over 73.5	over 82.5	FALSE	no
sunny	over 73.5	up to 82.5	TRUE	yes
sunny	up to 73.5	over 82.5	FALSE	no
sunny	up to 73.5	up to 82.5	FALSE	yes

In the discretized data, Apriori then looks for sets of ranges where the larger set is found often in the smaller. For example, one such rule in our table is:

```
play=yes ==> humidity=up to 82.5 & windy=FALSE
```

That is, sometimes when we `play`, humidity is high and there is no wind. Other associations in this dataset include:

```
humidity= up to 82.5 & windy=FALSE ==> play = no
humidity= over 82.5                ==> play = no
humidity= up to 82.5               ==> play = yes
temperature= up to 73.5            ==> outlook = rainy
outlook=overcast                   ==> play = yes
outlook=rainy                      ==> temperature = up to 73.5
play= yes                          ==> humidity = up to 82.5
play=no                            ==> humidity =  over 82.5
play=yes                           ==> outlook = overcast
```

Note that in association rule learning, the left- or right-hand side of the rule can contain one or more ranges. Also, while all the above are associations within our play data, some are much rarer than others. Apriori can generate any number of rules depending on a set of tuning parameters that define, say, the minimum number of examples needed before we can print a rule.

Formally, we say that an association rule learner takes as input D "transactions" of items I (e.g., see the above example table). As shown above, association rule learners return rules of the form $LHS \Rightarrow RHS$ where $LHS \subset I$ and $RHS \subset I$ and $LHS \cap RHS = \emptyset$. In the terminology of Apriori, an association rule $X \Rightarrow Y$ has *support* s if $s\%$ of D contains $X \wedge Y$, i.e., $s = \frac{|X \wedge Y|}{|D|}$, where $|\, X \wedge Y \,|$ denotes the number of transactions/examples in D containing both X and Y. The confidence c of an association rule is the percentage of transactions/examples containing X which also contain Y, i.e., $c = \frac{|X \wedge Y|}{|X|}$. As an example of these measures, consider the following rule:

```
play=yes ==> outlook = overcast
```

In this rule, *LHS* $=$ X $=$ `play=yes` and *RHS* $=$ Y $=$ `outlook=overcast`. Hence:

- support $= \frac{|X \wedge Y|}{|D|} = \frac{4}{14} = 0.29$
- confidence $= \frac{|X \wedge Y|}{|X|} = \frac{4}{9} = 0.44$.

Apriori was the first association rule pruning approach. When it was first proposed (1993), it was famous for its scalability. Running on a 33MHz machine with 64MB of RAM, Apriori was able to find associations in 838MB of data in under 100 s, which was quite a feat for those days. To achieve this, Apriori explored progressively larger combinations of ranges. Furthermore, the search for larger associations was constrained to smaller associations that occurred frequently. These *frequent itemsets* were grown incrementally and Apriori only explored itemsets of size N using items that occurred frequently of size $M < N$. Formally speaking, Apriori uses *support-based pruning*, i.e., when searching for rules with high confidence, sets of items $I_i, \ldots, I_k$ are examined only if all its subsets are above some minimum support value. After that, *confidence-based pruning* is applied to reject all rules that fall below some minimal threshold of adequate confidence.

3.3.1 Technical Aside: How to Discretize?

In the above example, we used a *discretization policy* before running Apriori. Such discretization is a useful technique for many other learning schemes (and we will return to discretization many times in this chapter).

For now, we just say that discretization need not be very clever [58]. For example, a 10 % chop is often as good as anything else (exception: for small tables of data like that shown above, it may be necessary to use fewer chops, just in case not enough information falls into each bin).

A newer method for discretization is to generate many small bins (e.g., 10 bins) then combine adjacent bins whose mean values are about the same. To apply this newer approach, we need some definition of "about the same" such as Hedges's test of Fig. 3.1.

3.4 Learning Trees

Apriori finds sets of interesting associations. For some applications this is useful but, when the query is more directed, another kind of learner may be more suited.

Hedges's test [28] explores two populations, each of which is characterized by its size, their mean, and standard deviation (denoted *n*, *mean*, and *sd*, respectively).
When testing if these two populations are different, we need to consider the following:

- If the standard deviation is large, then this *confuses* our ability to distinguish the bins.
- But if the sample size is large then we can *attenuate* the effects of the large standard deviation, i.e., the more we know about the sample, the more certain we are of the mean values.

Combining all that, we arrive at an informal measure of the difference between two means (note that this expression weights *confusion* by how many samples are trying to confuse us):

```
attenuate = n1 + n2
confusion = (n1*sd1 + n2*sd2) / attenuate
delta = abs(mean1 - mean2) / confusion
```

A more formally accepted version of the above, as endorsed by Kampenes et al. [31], is the following. To explain the difference between the above expression and Hedges's test, note the following.

- This test returns true if the *delta* is less than some "small" amount. The correct value of "small" is somewhat debatable but the values shown below are in the lower third of the "small" values seen in the 284 tests from the 64 experiments reviewed by Kampenes et al..
- A *c* term is added to handle small sample sizes (less than 20).
- Standard practice in statistics is to:
  - use $n-1$ in standard deviation calculations; and
  - use variance $\texttt{sd}^2$ rather than standard deviation.

```
function hedges(n1,mean1,sd1, n2,mean2,sd2) {
   small     = 0.17 # for a strict test. for a less severe
        test, use 0.38
   m1        = n1 - 1
   m2        = n2 - 1
   attenuate = m1 + m2
   confusion = sqrt( (m1 * (sd1)^2 +  m2 * (sd2)^2) /
       attenuate)
   delta     = abs(mean1 - mean2) / confusion
   c         = 1 - 3/(4*(m1 + n1) - 1)
   return delta * c < small
}
```

Fig. 3.1 A tutorial on Hedges's test of the effect size of the difference between two populations [28, 31]

3.4.1 C4.5

The *C4.5* decision tree learner [50] tries to ignore everything except the minimum combination of feature ranges that lead to different *decisions*. For example, if C4.5 reads the raw golf data (from Sect. 3.2), it would focus on the `play?` feature. It would then report what other feature ranges lead to such playful behavior. That report would take the form of the following tree:

```
outlook = sunny
|   humidity <= 75: yes
|   humidity > 75: no
outlook = overcast: yes
outlook = rainy
|   windy = TRUE: no
|   windy = FALSE: yes
```

To read this decision tree, note that subtrees are indented and that any line containing a colon (`:`) is a prediction. For example, the top branch of this tree says: "If `outlook` is `sunny` and `humidity` $\leq$ 75 then we will play golf." Note that this decision tree does not include `temp`, i.e., the temperature. This is not to say that golf playing behavior is unaffected by cold or heat. Rather, it is saying that, for this data, the other features are more important.

C4.5 looks for a feature value that simplifies the data. For example, consider the above table with five examples of `no` playing of golf and nine examples of `yes`, we played golf. Note that the *baseline* distributions in the table are `p1` $= 5/14$ and `p2` $= 9/14$ for `no` and `yes` (respectively). Now look at the middle of the above tree, at the branch `outlook = overcast`. C4.5 built this branch since within that region, the distributions are very simple indeed: all the rows where the outlook is overcast have `play? = yes`. That is, in this subtree `p1` $= 0$ and `p2` $= 100\,\%$.

Formally, we say that decision tree learners look for splits in the data that reduce the diversity of the data. This diversity is measured by the entropy equation discussed in Fig. 3.2. For example, in the golf example, the relative frequency of each class was `p1` $= 5/14$ and `p2` $= 9/14$. In that case:

```
e = entropy([5/14, 9/14])
  = -5/14 * log2(5/14) - 9/14 *log2(9/14) = 0.94
```

For the subtree selected by `outlook = overcast`, where `p1` $= 0$ and `p2` $= 100\,\%$, we ignore the zero value (since there is no information there) and compute:

```
n1 = 4
e1 = entropy([1]) = -1 * log2(1) = 0
```

Note that for the subtree with five rows selected by `outlook = sunny`, there are two `yes` and one `no`. That is:

```
n2 = 5
e2 = entropy([2/5, 3/5]) = 0.97
```

Also, and for the subtree with five rows selected by `outlook = rainy`, there are three `yes` and two `no`. Hence:

```
n3 = 5
e3 = entropy([3/5, 2/5]) = 0.97
```