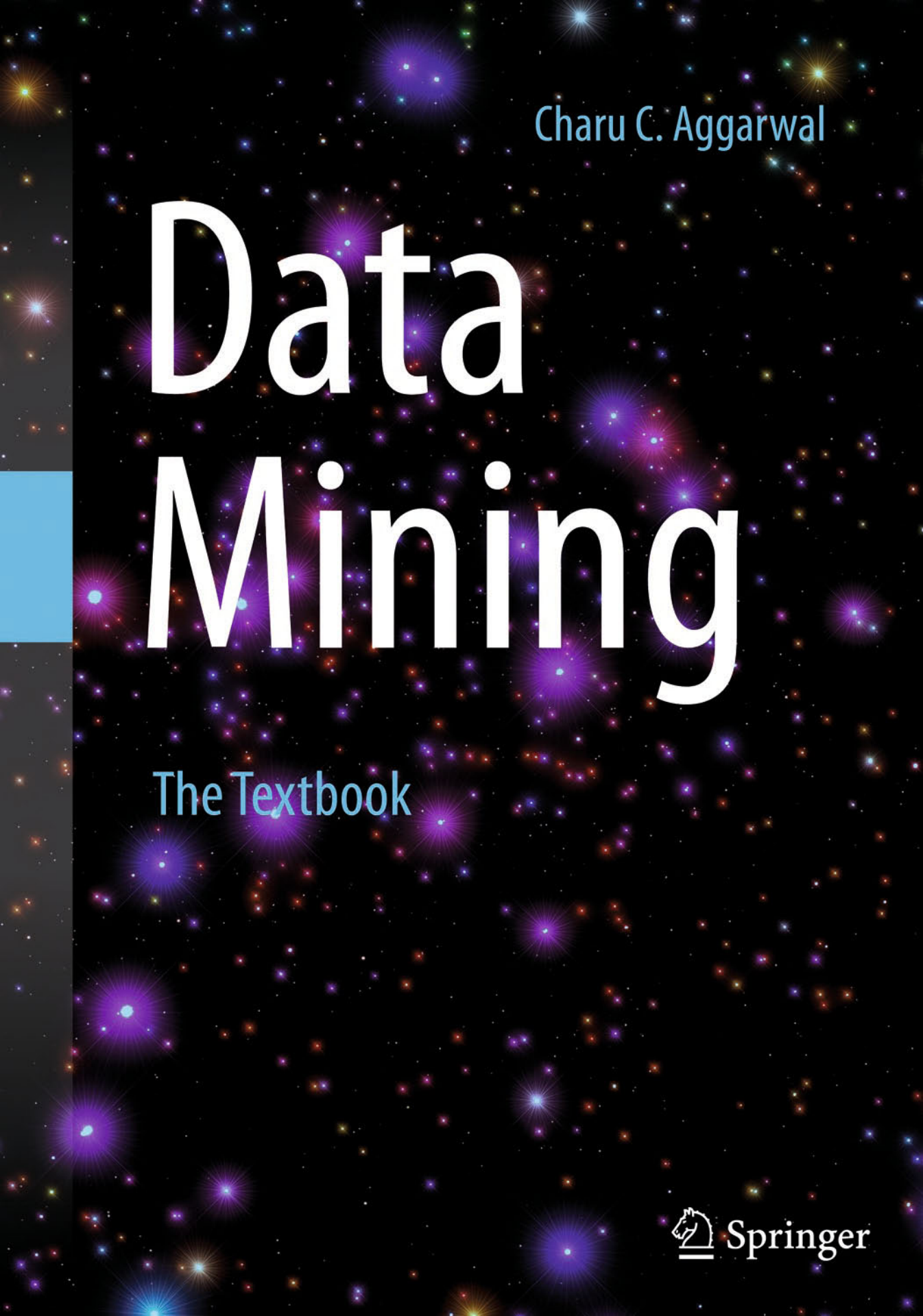

Charu C. Aggarwal
Data Mining
The Textbook
Springer

Data Mining: The Textbook

Charu C. Aggarwal

Data Mining

The Textbook

Springer

Charu C. Aggarwal
IBM T.J. Watson Research Center
Yorktown Heights
New York
USA

A solution manual for this book is available on Springer.com.

ISBN 978-3-319-14141-1 ISBN 978-3-319-14142-8 (eBook)
DOI 10.1007/978-3-319-14142-8

Library of Congress Control Number: 2015930833

Springer Cham Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To my wife Lata,
and my daughter Sayani

Contents

Preface

"*Data is the new oil.*"– Clive Humby

The field of data mining has seen rapid strides over the past two decades, especially from the perspective of the computer science community. While data analysis has been studied extensively in the conventional field of probability and statistics, *data mining* is a term coined by the computer science-oriented community. For computer scientists, issues such as scalability, usability, and computational implementation are extremely important.

The emergence of data science as a discipline requires the development of a book that goes beyond the traditional focus of books on only the fundamental data mining courses. Recent years have seen the emergence of the job description of "data scientists," who try to glean knowledge from vast amounts of data. In typical applications, the data types are so heterogeneous and diverse that the fundamental methods discussed for a multidimensional data type may not be effective. Therefore, more emphasis needs to be placed on the different data types and the applications that arise in the context of these different data types. A comprehensive data mining book must explore the different aspects of data mining, starting from the fundamentals, and then explore the complex data types, and their relationships with the fundamental techniques. While fundamental techniques form an excellent basis for the further study of data mining, they do not provide a complete picture of the true complexity of data analysis. This book studies these advanced topics without compromising the presentation of fundamental methods. Therefore, this book may be used for both introductory and advanced data mining courses. Until now, no single book has addressed all these topics in a comprehensive and integrated way.

The textbook assumes a basic knowledge of probability, statistics, and linear algebra, which is taught in most undergraduate curricula of science and engineering disciplines. Therefore, the book can also be used by industrial practitioners, who have a working knowledge of these basic skills. While stronger mathematical background is helpful for the more advanced chapters, it is not a prerequisite. Special chapters are also devoted to different aspects of data mining, such as text data, time-series data, discrete sequences, and graphs. This kind of specialized treatment is intended to capture the wide diversity of problem domains in which a data mining problem might arise.

The chapters of this book fall into one of three categories:

- **The fundamental chapters:** Data mining has four main "super problems," which correspond to clustering, classification, association pattern mining, and outlier anal-

ysis. These problems are so important because they are used repeatedly as building blocks in the context of a wide variety of data mining applications. As a result, a large amount of emphasis has been placed by data mining researchers and practitioners to design effective and efficient methods for these problems. These chapters comprehensively discuss the vast diversity of methods used by the data mining community in the context of these super problems.

- **Domain chapters:** These chapters discuss the specific methods used for different *domains* of data such as text data, time-series data, sequence data, graph data, and spatial data. Many of these chapters can also be considered application chapters, because they explore the specific characteristics of the problem in a particular domain.
- **Application chapters:** Advancements in hardware technology and software platforms have lead to a number of data-intensive applications such as streaming systems, Web mining, social networks, and privacy preservation. These topics are studied in detail in these chapters. The domain chapters are also focused on many different kinds of applications that arise in the context of those data types.

Suggestions for the Instructor

The book was specifically written to enable the teaching of both the basic data mining and advanced data mining courses from a single book. It can be used to offer various types of data mining courses with different emphases. Specifically, the courses that could be offered with various chapters are as follows:

- **Basic data mining course and fundamentals:** The basic data mining course should focus on the fundamentals of data mining. Chapters 1, 2, 3, 4, 6, 8, and 10 can be covered. In fact, the material in these chapters is more than what is possible to teach in a single course. Therefore, instructors may need to select topics of their interest from these chapters. Some portions of Chaps. 5, 7, 9, and 11 can also be covered, although these chapters are really meant for an advanced course.
- **Advanced course (fundamentals):** Such a course would cover advanced topics on the fundamentals of data mining and assume that the student is already familiar with Chaps. 1–3, and parts of Chaps. 4, 6, 8, and 10. The course can then focus on Chaps. 5, 7, 9, and 11. Topics such as ensemble analysis are useful for the advanced course. Furthermore, some topics from Chaps. 4, 6, 8, and 10, which were not covered in the basic course, can be used. In addition, Chap. 20 on privacy can be offered.
- **Advanced course (data types):** Advanced topics such as text mining, time series, sequences, graphs, and spatial data may be covered. The material should focus on Chaps. 13, 14, 15, 16, and 17. Some parts of Chap. 19 (e.g., graph clustering) and Chap. 12 (data streaming) can also be used.
- **Advanced course (applications):** An application course overlaps with a data type course but has a different focus. For example, the focus in an application-centered course would be more on the modeling aspect than the algorithmic aspect. Therefore, the same materials in Chaps. 13, 14, 15, 16, and 17 can be used while skipping specific details of algorithms. With less focus on specific algorithms, these chapters can be covered fairly quickly. The remaining time should be allocated to three very important chapters on data streams (Chap. 12), Web mining (Chap. 18), and social network analysis (Chap. 19).

The book is written in a simple style to make it accessible to undergraduate students and industrial practitioners with a limited mathematical background. Thus, the book will serve both as an introductory text and as an advanced text for students, industrial practitioners, and researchers.

Throughout this book, a vector or a multidimensional data point (including categorical attributes), is annotated with a bar, such as $\overline{X}$ or $\overline{y}$. A vector or multidimensional point may be denoted by either small letters or capital letters, as long as it has a bar. Vector dot products are denoted by centered dots, such as $\overline{X} \cdot \overline{Y}$. A matrix is denoted in capital letters without a bar, such as R. Throughout the book, the $n \times d$ data matrix is denoted by D, with n points and d dimensions. The individual data points in D are therefore d-dimensional row vectors. On the other hand, vectors with one component for each data point are usually n-dimensional column vectors. An example is the n-dimensional column vector $\overline{y}$ of class variables of n data points.

Acknowledgments

I would like to thank my wife and daughter for their love and support during the writing of this book. The writing of a book requires significant time, which is taken away from family members. This book is the result of their patience with me during this time.

I would also like to thank my manager Nagui Halim for providing the tremendous support necessary for the writing of this book. His professional support has been instrumental for my many book efforts in the past and present.

During the writing of this book, I received feedback from many colleagues. In particular, I received feedback from Kanishka Bhaduri, Alain Biem, Graham Cormode, Hongbo Deng, Amit Dhurandhar, Bart Goethals, Alexander Hinneburg, Ramakrishnan Kannan, George Karypis, Dominique LaSalle, Abdullah Mueen, Guojun Qi, Pierangela Samarati, Saket Sathe, Karthik Subbian, Jiliang Tang, Deepak Turaga, Jilles Vreeken, Jieping Ye, and Peixiang Zhao. I would like to thank them for their constructive feedback and suggestions. Over the years, I have benefited from the insights of numerous collaborators. These insights have influenced this book directly or indirectly. I would first like to thank my long-term collaborator Philip S. Yu for my years of collaboration with him. Other researchers with whom I have had significant collaborations include Tarek F. Abdelzaher, Jing Gao, Quanquan Gu, Manish Gupta, Jiawei Han, Alexander Hinneburg, Thomas Huang, Nan Li, Huan Liu, Ruoming Jin, Daniel Keim, Arijit Khan, Latifur Khan, Mohammad M. Masud, Jian Pei, Magda Procopiuc, Guojun Qi, Chandan Reddy, Jaideep Srivastava, Karthik Subbian, Yizhou Sun, Jiliang Tang, Min-Hsuan Tsai, Haixun Wang, Jianyong Wang, Min Wang, Joel Wolf, Xifeng Yan, Mohammed Zaki, ChengXiang Zhai, and Peixiang Zhao.

I would also like to thank my advisor James B. Orlin for his guidance during my early years as a researcher. While I no longer work in the same area, the legacy of what I learned from him is a crucial part of my approach to research. In particular, he taught me the importance of intuition and simplicity of thought in the research process. These are more important aspects of research than is generally recognized. This book is written in a simple and intuitive style, and is meant to improve accessibility of this area to both researchers and practitioners.

I would also like to thank Lata Aggarwal for helping me with some of the figures drawn using Microsoft Powerpoint.

Author Biography

Charu C. Aggarwal is a Distinguished Research Staff Member (DRSM) at the IBM T. J. Watson Research Center in Yorktown Heights, New York. He completed his B.S. from IIT Kanpur in 1993 and his Ph.D. from the Massachusetts Institute of Technology in 1996.

He has worked extensively in the field of data mining. He has published more than 250 papers in refereed conferences and journals and authored over 80 patents. He is author or editor of 14 books, including the first comprehensive book on outlier analysis, which is written from a computer science point of view. Because of the commercial value of his patents, he has thrice been designated a Master Inventor at IBM. He is a recipient of an IBM Corporate Award (2003) for his work on bio-terrorist threat detection in data streams, a recipient of the IBM Outstanding Innovation Award (2008) for his scientific contributions to privacy technology, a recipient of the IBM Outstanding Technical Achievement Award (2009) for his work on data streams, and a recipient of an IBM Research Division Award (2008) for his contributions to System S. He also received the EDBT 2014 Test of Time Award for his work on condensation-based privacy-preserving data mining.

He has served as the general co-chair of the IEEE Big Data Conference, 2014, and as an associate editor of the IEEE Transactions on Knowledge and Data Engineering from 2004 to 2008. He is an associate editor of the ACM Transactions on Knowledge Discovery from Data, an action editor of the Data Mining and Knowledge Discovery Journal, editor-in-chief of the ACM SIGKDD Explorations, and an associate editor of the Knowledge and Information Systems Journal. He serves on the advisory board of the Lecture Notes on Social Networks, a publication by Springer. He has served as the vice-president of the SIAM Activity Group on Data Mining. He is a fellow of the ACM and the IEEE, for "contributions to knowledge discovery and data mining algorithms."

Chapter 1

An Introduction to Data Mining

"*Education is not the piling on of learning, information, data, facts, skills, or abilities – that's training or instruction – but is rather making visible what is hidden as a seed.*"—Thomas More

1.1 Introduction

Data mining is the study of collecting, cleaning, processing, analyzing, and gaining useful insights from data. A wide variation exists in terms of the problem domains, applications, formulations, and data representations that are encountered in real applications. Therefore, "data mining" is a broad umbrella term that is used to describe these different aspects of data processing.

In the modern age, virtually all automated systems generate some form of data either for diagnostic or analysis purposes. This has resulted in a deluge of data, which has been reaching the order of petabytes or exabytes. Some examples of different kinds of data are as follows:

- *World Wide Web:* The number of documents on the indexed Web is now on the order of billions, and the invisible Web is much larger. User accesses to such documents create Web access logs at servers and customer behavior profiles at commercial sites. Furthermore, the linked structure of the Web is referred to as the *Web graph*, which is itself a kind of data. These different types of data are useful in various applications. For example, the Web documents and link structure can be mined to determine associations between different topics on the Web. On the other hand, user access logs can be mined to determine frequent patterns of accesses or unusual patterns of possibly unwarranted behavior.

- *Financial interactions:* Most common transactions of everyday life, such as using an automated teller machine (ATM) card or a credit card, can create data in an automated way. Such transactions can be mined for many useful insights such as fraud or other unusual activity.

C. C. Aggarwal, *Data Mining: The Textbook*, DOI 10.1007/978-3-319-14142-8_1

- *User interactions:* Many forms of user interactions create large volumes of data. For example, the use of a telephone typically creates a record at the telecommunication company with details about the duration and destination of the call. Many phone companies routinely analyze such data to determine relevant patterns of behavior that can be used to make decisions about network capacity, promotions, pricing, or customer targeting.

- *Sensor technologies and the Internet of Things:* A recent trend is the development of low-cost wearable sensors, smartphones, and other smart devices that can communicate with one another. By one estimate, the number of such devices exceeded the number of people on the planet in 2008 [30]. The implications of such massive data collection are significant for mining algorithms.

The deluge of data is a direct result of advances in technology and the computerization of every aspect of modern life. It is, therefore, natural to examine whether one can extract *concise* and possibly *actionable* insights from the available data for application-specific goals. This is where the task of data mining comes in. The raw data may be arbitrary, unstructured, or even in a format that is not immediately suitable for automated processing. For example, manually collected data may be drawn from heterogeneous sources in different formats and yet somehow needs to be processed by an automated computer program to gain insights.

To address this issue, data mining analysts use a pipeline of processing, where the raw data are collected, cleaned, and transformed into a standardized format. The data may be stored in a commercial database system and finally processed for insights with the use of analytical methods. In fact, while data mining often conjures up the notion of analytical algorithms, the reality is that the vast majority of work is related to the data preparation portion of the process. This pipeline of processing is conceptually similar to that of an actual mining process from a mineral ore to the refined end product. The term "mining" derives its roots from this analogy.

From an analytical perspective, data mining is challenging because of the wide disparity in the problems and data types that are encountered. For example, a commercial product recommendation problem is very different from an intrusion-detection application, even at the level of the input data format or the problem definition. Even within related classes of problems, the differences are quite significant. For example, a product recommendation problem in a multidimensional database is very different from a social recommendation problem due to the differences in the underlying data type. Nevertheless, in spite of these differences, data mining applications are often closely connected to one of four "super-problems" in data mining: association pattern mining, clustering, classification, and outlier detection. These problems are so important because they are used as building blocks in a majority of the applications in some indirect form or the other. This is a useful abstraction because it helps us conceptualize and structure the field of data mining more effectively.

The data may have different formats or *types*. The type may be quantitative (e.g., age), categorical (e.g., ethnicity), text, spatial, temporal, or graph-oriented. Although the most common form of data is multidimensional, an increasing proportion belongs to more complex data types. While there is a conceptual portability of algorithms between many data types at a very high level, this is not the case from a practical perspective. The reality is that the precise data type may affect the behavior of a particular algorithm significantly. As a result, one may need to design refined variations of the basic approach for multidimensional data, so that it can be used effectively for a different data type. Therefore, this book will dedicate different chapters to the various data types to provide a better understanding of how the processing methods are affected by the underlying data type.

A major challenge has been created in recent years due to increasing data volumes. The prevalence of continuously collected data has led to an increasing interest in the field of *data streams*. For example, Internet traffic generates large streams that cannot even be stored effectively unless significant resources are spent on storage. This leads to unique challenges from the perspective of processing and analysis. In cases where it is not possible to explicitly store the data, all the processing needs to be performed in real time.

This chapter will provide a broad overview of the different technologies involved in preprocessing and analyzing different types of data. The goal is to study data mining from the perspective of different problem abstractions and data types that are frequently encountered. Many important applications can be converted into these abstractions.

This chapter is organized as follows. Section 1.2 discusses the data mining process with particular attention paid to the data preprocessing phase in this section. Different data types and their formal definition are discussed in Sect. 1.3. The major problems in data mining are discussed in Sect. 1.4 at a very high level. The impact of data type on problem definitions is also addressed in this section. Scalability issues are addressed in Sect. 1.5. In Sect. 1.6, a few examples of applications are provided. Section 1.7 gives a summary.

1.2 The Data Mining Process

As discussed earlier, the data mining process is a pipeline containing many phases such as data cleaning, feature extraction, and algorithmic design. In this section, we will study these different phases. The workflow of a typical data mining application contains the following phases:

1. *Data collection:* Data collection may require the use of specialized hardware such as a sensor network, manual labor such as the collection of user surveys, or software tools such as a Web document crawling engine to collect documents. While this stage is highly application-specific and often outside the realm of the data mining analyst, it is critically important because good choices at this stage may significantly impact the data mining process. After the collection phase, the data are often stored in a database, or, more generally, a *data warehouse* for processing.

2. *Feature extraction and data cleaning:* When the data are collected, they are often not in a form that is suitable for processing. For example, the data may be encoded in complex logs or free-form documents. In many cases, different types of data may be arbitrarily mixed together in a free-form document. To make the data suitable for processing, it is essential to transform them into a format that is friendly to data mining algorithms, such as multidimensional, time series, or semistructured format. The multidimensional format is the most common one, in which different *fields* of the data correspond to the different measured properties that are referred to as *features*, *attributes*, or *dimensions*. It is crucial to extract relevant features for the mining process. The feature extraction phase is often performed in parallel with data cleaning, where missing and erroneous parts of the data are either estimated or corrected. In many cases, the data may be extracted from multiple sources and need to be *integrated* into a unified format for processing. The final result of this procedure is a nicely structured data set, which can be effectively used by a computer program. After the feature extraction phase, the data may again be stored in a database for processing.

3. *Analytical processing and algorithms:* The final part of the mining process is to design effective analytical methods from the processed data. In many cases, it may not be

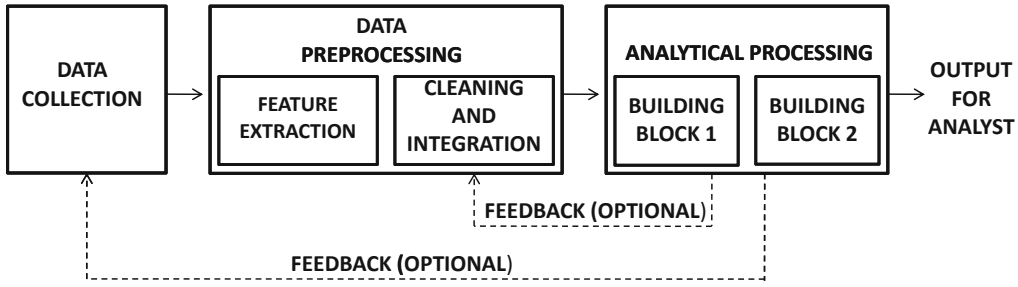


Figure 1.1: The data processing pipeline

possible to directly use a standard data mining problem, such as the four "superproblems" discussed earlier, for the application at hand. However, these four problems have such wide coverage that *many* applications can be broken up into components that use these different building blocks. This book will provide examples of this process.

The overall data mining process is illustrated in Fig. 1.1. Note that the analytical block in Fig. 1.1 shows multiple building blocks representing the design of the solution to a particular application. This part of the algorithmic design is dependent on the skill of the analyst and often uses one or more of the four major problems as a building block. This is, of course, not always the case, but it is frequent enough to merit special treatment of these four problems within this book. To explain the data mining process, we will use an example from a recommendation scenario.

Example 1.2.1 *Consider a scenario in which a retailer has Web logs corresponding to customer accesses to Web pages at his or her site. Each of these Web pages corresponds to a product, and therefore a customer access to a page may often be indicative of interest in that particular product. The retailer also stores demographic profiles for the different customers. The retailer wants to make targeted product recommendations to customers using the customer demographics and buying behavior.*

Sample Solution Pipeline In this case, the first step for the analyst is to collect the relevant data from two different sources. The first source is the set of Web logs at the site. The second is the demographic information within the retailer database that were collected during Web registration of the customer. Unfortunately, these data sets are in a very different format and cannot easily be used together for processing. For example, consider a sample log entry of the following form:

```
98.206.207.157 - - [31/Jul/2013:18:09:38 -0700] "GET /productA.htm
HTTP/1.1" 200 328177 "-" "Mozilla/5.0 (Mac OS X) AppleWebKit/536.26
(KHTML, like Gecko) Version/6.0 Mobile/10B329 Safari/8536.25"
"retailer.net"
```

The log may contain hundreds of thousands of such entries. Here, a customer at IP address 98.206.207.157 has accessed productA.htm. The customer from the IP address can be identified using the previous login information, by using cookies, or by the IP address itself, but this may be a noisy process and may not always yield accurate results. The analyst would need to design algorithms for deciding how to filter the different log entries and use only those which provide accurate results as a part of the *cleaning and extraction* process. Furthermore, the raw log contains a lot of additional information that is not necessarily

of any use to the retailer. In the *feature extraction* process, the retailer decides to create one record for each customer, with a specific choice of features extracted from the Web page accesses. For each record, an attribute corresponds to the number of accesses to each product description. Therefore, the raw logs need to be processed, and the accesses need to be aggregated during this *feature extraction* phase. Attributes are added to these records for the retailer's database containing demographic information in a *data integration phase*. Missing entries from the demographic records need to be estimated for further *data cleaning*. This results in a single data set containing attributes for the customer demographics and customer accesses.

At this point, the analyst has to decide how to use this cleaned data set for making recommendations. He or she decides to determine similar groups of customers, and make recommendations on the basis of the buying behavior of these similar groups. In particular, the *building block* of clustering is used to determine similar groups. For a given customer, the most frequent items accessed by the customers in that group are recommended. This provides an example of the entire data mining pipeline. As you will learn in Chap. 18, there are many elegant ways of performing the recommendations, some of which are more effective than the others depending on the specific definition of the problem. Therefore, the entire data mining process is an art form, which is based on the skill of the analyst, and cannot be fully captured by a single technique or building block. In practice, this skill can be learned only by working with a diversity of applications over different scenarios and data types.

1.2.1 The Data Preprocessing Phase

The data preprocessing phase is perhaps the most crucial one in the data mining process. Yet, it is rarely explored to the extent that it deserves because most of the focus is on the analytical aspects of data mining. This phase begins after the collection of the data, and it consists of the following steps:

1. *Feature extraction:* An analyst may be confronted with vast volumes of raw documents, system logs, or commercial transactions with little guidance on how these raw data should be transformed into meaningful database features for processing. This phase is highly dependent on the analyst to be able to abstract out the features that are most relevant to a particular application. For example, in a credit-card fraud detection application, the amount of a charge, the repeat frequency, and the location are often good indicators of fraud. However, many other features may be poorer indicators of fraud. Therefore, extracting the right features is often a skill that requires an understanding of the specific application domain at hand.

2. *Data cleaning:* The extracted data may have erroneous or missing entries. Therefore, some records may need to be dropped, or missing entries may need to be estimated. Inconsistencies may need to be removed.

3. *Feature selection and transformation:* When the data are very high dimensional, many data mining algorithms do not work effectively. Furthermore, many of the high-dimensional features are noisy and may add errors to the data mining process. Therefore, a variety of methods are used to either remove irrelevant features or transform the current set of features to a new data space that is more amenable for analysis. Another related aspect is data *transformation*, where a data set with a particular set of attributes may be transformed into a data set with another set of attributes of the same or a different type. For example, an attribute, such as age, may be partitioned into ranges to create discrete values for analytical convenience.

The data cleaning process requires statistical methods that are commonly used for missing data estimation. In addition, erroneous data entries are often removed to ensure more accurate mining results. The topics of data cleaning is addressed in Chap. 2 on data preprocessing.

Feature selection and transformation should not be considered a part of data preprocessing because the feature selection phase is often highly dependent on the specific analytical problem being solved. In some cases, the feature selection process can even be tightly integrated with the specific algorithm or methodology being used, in the form of a *wrapper model* or *embedded model*. Nevertheless, the feature selection phase is usually performed before applying the specific algorithm at hand.

1.2.2 The Analytical Phase

The vast majority of this book will be devoted to the analytical phase of the mining process. A major challenge is that each data mining application is unique, and it is, therefore, difficult to create general and reusable techniques across different applications. Nevertheless, many data mining formulations are repeatedly used in the context of different applications. These correspond to the major "superproblems" or building blocks of the data mining process. It is dependent on the skill and experience of the analyst to determine how these different formulations may be used in the context of a particular data mining application. Although this book can provide a good overview of the fundamental data mining models, the ability to apply them to real-world applications can only be learned with practical experience.

1.3 The Basic Data Types

One of the interesting aspects of the data mining process is the wide variety of data types that are available for analysis. There are two broad types of data, of varying complexity, for the data mining process:

1. *Nondependency-oriented data:* This typically refers to simple data types such as multidimensional data or text data. These data types are the simplest and most commonly encountered. In these cases, the data records do not have any specified dependencies between either the data items or the attributes. An example is a set of demographic records about individuals containing their age, gender, and ZIP code.

2. *Dependency-oriented data:* In these cases, implicit or explicit relationships may exist between data items. For example, a social network data set contains a set of *vertices* (data items) that are connected together by a set of *edges* (relationships). On the other hand, time series contains implicit dependencies. For example, two successive values collected from a sensor are likely to be related to one another. Therefore, the time attribute implicitly specifies a dependency between successive readings.

In general, dependency-oriented data are more challenging because of the complexities created by preexisting relationships between data items. Such dependencies between data items need to be incorporated directly into the analytical process to obtain contextually meaningful results.

Table 1.1: An example of a multidimensional data set

Name	Age	Gender	Race	ZIP code
John S.	45	M	African American	05139
Manyona L.	31	F	Native American	10598
Sayani A.	11	F	East Indian	10547
Jack M.	56	M	Caucasian	10562
Wei L.	63	M	Asian	90210

1.3.1 Nondependency-Oriented Data

This is the simplest form of data and typically refers to *multidimensional data*. This data typically contains a set of *records*. A record is also referred to as a *data point*, *instance*, *example*, *transaction*, *entity*, *tuple*, *object*, or *feature-vector*, depending on the application at hand. Each record contains a set of *fields*, which are also referred to as *attributes*, *dimensions*, and *features*. These terms will be used interchangeably throughout this book. These fields describe the different properties of that record. Relational database systems were traditionally designed to handle this kind of data, even in their earliest forms. For example, consider the demographic data set illustrated in Table 1.1. Here, the demographic properties of an individual, such as age, gender, and ZIP code, are illustrated. A multidimensional data set is defined as follows:

Definition 1.3.1 (Multidimensional Data) *A multidimensional data set $\mathcal{D}$ is a set of n records, $\overline{X_1} \ldots \overline{X_n}$, such that each record $\overline{X_i}$ contains a set of d features denoted by $(x_i^1 \ldots x_i^d)$.*

Throughout the early chapters of this book, we will work with multidimensional data because it is the simplest form of data and establishes the broader principles on which the more complex data types can be processed. More complex data types will be addressed in later chapters of the book, and the impact of the dependencies on the mining process will be explicitly discussed.

1.3.1.1 Quantitative Multidimensional Data

The attributes in Table 1.1 are of two different types. The age field has values that are numerical in the sense that they have a natural ordering. Such attributes are referred to as *continuous*, *numeric*, or *quantitative*. Data in which all fields are quantitative is also referred to as *quantitative data* or *numeric data*. Thus, when each value of x_i^j in Definition 1.3.1 is quantitative, the corresponding data set is referred to as quantitative multidimensional data. In the data mining literature, this particular subtype of data is considered the most common, and many algorithms discussed in this book work with this subtype of data. This subtype is particularly convenient for analytical processing because it is much easier to work with quantitative data from a statistical perspective. For example, the mean of a set of quantitative records can be expressed as a simple average of these values, whereas such computations become more complex in other data types. Where possible and effective, many data mining algorithms therefore try to convert different kinds of data to quantitative values before processing. This is also the reason that many algorithms discussed in this (or virtually any other) data mining textbook assume a quantitative multidimensional representation. Nevertheless, in real applications, the data are likely to be more complex and may contain a mixture of different data types.

1.3.1.2 Categorical and Mixed Attribute Data

Many data sets in real applications may contain categorical attributes that take on *discrete unordered* values. For example, in Table 1.1, the attributes such as gender, race, and ZIP code, have discrete values without a natural ordering among them. If each value of x_i^j in Definition 1.3.1 is categorical, then such data are referred to as *unordered discrete-valued* or *categorical*. In the case of *mixed attribute* data, there is a combination of categorical and numeric attributes. The full data in Table 1.1 are considered mixed-attribute data because they contain both numeric and categorical attributes.

The attribute corresponding to gender is special because it is categorical, but with only two possible values. In such cases, it is possible to impose an artificial ordering between these values and use algorithms designed for numeric data for this type. This is referred to as *binary* data, and it can be considered a special case of either numeric or categorical data. Chap. 2 will explain how binary data form the "bridge" to transform numeric or categorical attributes into a common format that is suitable for processing in many scenarios.

1.3.1.3 Binary and Set Data

Binary data can be considered a special case of either multidimensional categorical data or multidimensional quantitative data. It is a special case of multidimensional categorical data, in which each categorical attribute may take on one of at most two discrete values. It is also a special case of multidimensional quantitative data because an ordering exists between the two values. Furthermore, binary data is also a representation of setwise data, in which each attribute is treated as a set element indicator. A value of 1 indicates that the element should be included in the set. Such data is common in market basket applications. This topic will be studied in detail in Chaps. 4 and 5.

1.3.1.4 Text Data

Text data can be viewed either as a string, or as multidimensional data, depending on how they are represented. In its raw form, a text document corresponds to a *string*. This is a dependency-oriented data type, which will be described later in this chapter. Each string is a sequence of characters (or words) corresponding to the document. However, text documents are rarely represented as strings. This is because it is difficult to directly use the ordering between words in an efficient way for large-scale applications, and the additional advantages of leveraging the ordering are often limited in the text domain.

In practice, a *vector-space representation* is used, where the frequencies of the words in the document are used for analysis. Words are also sometimes referred to as *terms*. Thus, the precise ordering of the words is lost in this representation. These frequencies are typically normalized with statistics such as the length of the document, or the frequencies of the individual words in the collection. These issues will be discussed in detail in Chap. 13 on text data. The corresponding $n \times d$ data matrix for a text collection with n documents and d terms is referred to as a *document-term matrix*.

When represented in vector-space form, text data can be considered multidimensional quantitative data, where the attributes correspond to the words, and the values correspond to the frequencies of these attributes. However, this kind of quantitative data is special because most attributes take on zero values, and only a few attributes have nonzero values. This is because a single document may contain only a relatively small number of words out of a dictionary of size 10^5. This phenomenon is referred to as *data sparsity*, and it significantly impacts the data mining process. The direct use of a quantitative data mining

algorithm is often unlikely to work with sparse data without appropriate modifications. The sparsity also affects how the data are represented. For example, while it is possible to use the representation suggested in Definition 1.3.1, this is not a practical approach. Most values of x_i^j in Definition 1.3.1 are 0 for the case of text data. Therefore, it is inefficient to explicitly maintain a d-dimensional representation in which most values are 0. A bag-of-words representation is used containing only the words in the document. In addition, the frequencies of these words are explicitly maintained. This approach is typically more efficient. Because of data sparsity issues, text data are often processed with specialized methods. Therefore, text mining is often studied as a separate subtopic within data mining. Text mining methods are discussed in Chap. 13.

1.3.2 Dependency-Oriented Data

Most of the aforementioned discussion in this chapter is about the multidimensional scenario, where it is assumed that the data records can be treated independently of one another. In practice, the different data values may be (implicitly) related to each other temporally, spatially, or through explicit network relationship links between the data items. The knowledge about *preexisting* dependencies greatly changes the data mining process because data mining is all about finding relationships between data items. The presence of preexisting dependencies therefore changes the *expected* relationships in the data, and what may be considered *interesting* from the perspective of these expected relationships. Several types of dependencies may exist that may be either *implicit* or *explicit*:

1. *Implicit dependencies:* In this case, the dependencies between data items are not explicitly specified but are known to "typically" exist in that domain. For example, consecutive temperature values collected by a sensor are likely to be extremely similar to one another. Therefore, if the temperature value recorded by a sensor at a particular time is significantly different from that recorded at the next time instant then this is extremely unusual and may be interesting for the data mining process. This is different from multidimensional data sets where each data record is treated as an independent entity.

2. *Explicit dependencies:* This typically refers to graph or network data in which edges are used to specify explicit relationships. Graphs are a very powerful abstraction that are often used as an intermediate representation to solve data mining problems in the context of other data types.

In this section, the different dependency-oriented data types will be discussed in detail.

1.3.2.1 Time-Series Data

Time-series data contain values that are typically generated by continuous measurement over time. For example, an environmental sensor will measure the temperature continuously, whereas an electrocardiogram (ECG) will measure the parameters of a subject's heart rhythm. Such data typically have *implicit* dependencies built into the values received over time. For example, the adjacent values recorded by a temperature sensor will usually vary smoothly over time, and this factor needs to be explicitly used in the data mining process.

The nature of the temporal dependency may vary significantly with the application. For example, some forms of sensor readings may show periodic patterns of the measured

attribute over time. An important aspect of time-series mining is the extraction of such dependencies in the data. To formalize the issue of dependencies caused by temporal correlation, the attributes are classified into two types:

1. *Contextual attributes:* These are the attributes that define the *context* on the basis of which the implicit dependencies occur in the data. For example, in the case of sensor data, the time stamp at which the reading is measured may be considered the contextual attribute. Sometimes, the time stamp is not explicitly used, but a position index is used. While the time-series data type contains only one contextual attribute, other data types may have more than one contextual attribute. A specific example is *spatial data*, which will be discussed later in this chapter.

2. *Behavioral attributes:* These represent the values that are measured in a particular context. In the sensor example, the temperature is the behavioral attribute value. It is possible to have more than one behavioral attribute. For example, if multiple sensors record readings at synchronized time stamps, then it results in a multidimensional time-series data set.

The contextual attributes typically have a strong impact on the dependencies between the behavioral attribute values in the data. Formally, time-series data are defined as follows:

Definition 1.3.2 (Multivariate Time-Series Data) *A time series of length n and dimensionality d contains d numeric features at each of n time stamps $t_1 \ldots t_n$. Each time-stamp contains a component for each of the d series. Therefore, the set of values received at time stamp t_i is $\overline{Y_i} = (y_i^1 \ldots y_i^d)$. The value of the jth series at time stamp t_i is y_i^j.*

For example, consider the case where two sensors at a particular location monitor the temperature and pressure every second for a minute. This corresponds to a multidimensional series with $d = 2$ and $n = 60$. In some cases, the time stamps $t_1 \ldots t_n$ may be replaced by index values from 1 through n, especially when the time-stamp values are equally spaced apart.

Time-series data are relatively common in many sensor applications, forecasting, and financial market analysis. Methods for analyzing time series are discussed in Chap. 14.

1.3.2.2 Discrete Sequences and Strings

Discrete sequences can be considered the categorical analog of time-series data. As in the case of time-series data, the contextual attribute is a time stamp or a position index in the ordering. The behavioral attribute is a categorical value. Therefore, discrete sequence data are defined in a similar way to time-series data.

Definition 1.3.3 (Multivariate Discrete Sequence Data) *A discrete sequence of length n and dimensionality d contains d discrete feature values at each of n different time stamps $t_1 \ldots t_n$. Each of the n components $\overline{Y_i}$ contains d discrete behavioral attributes $(y_i^1 \ldots y_i^d)$, collected at the ith time-stamp.*

For example, consider a sequence of Web accesses, in which the Web page address and the originating IP address of the request are collected for 100 different accesses. This represents a discrete sequence of length $n = 100$ and dimensionality $d = 2$. A particularly common case in sequence data is the *univariate* scenario, in which the value of d is 1. Such sequence data are also referred to as *strings*.

It should be noted that the aforementioned definition is almost identical to the time-series case, with the main difference being that discrete sequences contain categorical attributes. In theory, it is possible to have series that are mixed between categorical and numerical data. Another important variation is the case where a sequence does not contain categorical attributes, but a *set* of any number of unordered categorical values. For example, supermarket transactions may contain a sequence of sets of items. Each set may contain any number of items. Such setwise sequences are not really multivariate sequences, but are univariate sequences, in which each element of the sequence is a *set* as opposed to a unit element. Thus, discrete sequences can be defined in a wider variety of ways, as compared to time-series data because of the ability to define sets on discrete elements.

In some cases, the contextual attribute may not refer to time explicitly, but it might be a position based on physical placement. This is the case for biological sequence data. In such cases, the time stamp may be replaced by an index representing the position of the value in the string, counting the leftmost position as 1. Some examples of common scenarios in which sequence data may arise are as follows:

- *Event logs:* A wide variety of computer systems, Web servers, and Web applications create event logs on the basis of user activity. An example of an event log is a sequence of user actions at a financial Web site:

  ```
  Login Password Login Password Login Password ....
  ```

  This particular sequence may represent a scenario where a user is attempting to break into a password-protected system, and it may be interesting from the perspective of anomaly detection.

- *Biological data:* In this case, the sequences may correspond to strings of nucleotides or amino acids. The ordering of such units provides information about the characteristics of protein function. Therefore, the data mining process can be used to determine interesting patterns that are reflective of different biological properties.

Discrete sequences are often more challenging for mining algorithms because they do not have the smooth value continuity of time-series data. Methods for sequence mining are discussed in Chap. 15.

1.3.2.3 Spatial Data

In spatial data, many nonspatial attributes (e.g., temperature, pressure, image pixel color intensity) are measured at spatial locations. For example, sea-surface temperatures are often collected by meteorologists to forecast the occurrence of hurricanes. In such cases, the spatial coordinates correspond to contextual attributes, whereas attributes such as the temperature correspond to the behavioral attributes. Typically, there are two spatial attributes. As in the case of time-series data, it is also possible to have multiple behavioral attributes. For example, in the sea-surface temperature application, one might also measure other behavioral attributes such as the pressure.

Definition 1.3.4 (Spatial Data) *A d-dimensional spatial data record contains d behavioral attributes and one or more contextual attributes containing the spatial location. Therefore, a d-dimensional spatial data set is a set of d dimensional records $\overline{X_1} \ldots \overline{X_n}$, together with a set of n locations $L_1 \ldots L_n$, such that the record $\overline{X_i}$ is associated with the location L_i.*

The aforementioned definition provides broad flexibility in terms of how record $\overline{X_i}$ and location L_i may be defined. For example, the behavioral attributes in record $\overline{X_i}$ may be numeric or categorical, or a mixture of the two. In the meteorological application, $\overline{X_i}$ may contain the temperature and pressure attributes at location L_i. Furthermore, L_i may be specified in terms of precise spatial coordinates, such as latitude and longitude, or in terms of a logical location, such as the city or state.

Spatial data mining is closely related to time-series data mining, in that the behavioral attributes in most commonly studied spatial applications are continuous, although some applications may use categorical attributes as well. Therefore, value continuity is observed across contiguous spatial locations, just as value continuity is observed across contiguous time stamps in time-series data.

Spatiotemporal Data

A particular form of spatial data is spatiotemporal data, which contains both spatial and temporal attributes. The precise nature of the data also depends on which of the attributes are contextual and which are behavioral. Two kinds of spatiotemporal data are most common:

1. *Both spatial and temporal attributes are contextual:* This kind of data can be viewed as a direct generalization of both spatial data and temporal data. This kind of data is particularly useful when the spatial and temporal dynamics of particular behavioral attributes are measured simultaneously. For example, consider the case where the variations in the sea-surface temperature need to be measured over time. In such cases, the temperature is the behavioral attribute, whereas the spatial and temporal attributes are contextual.

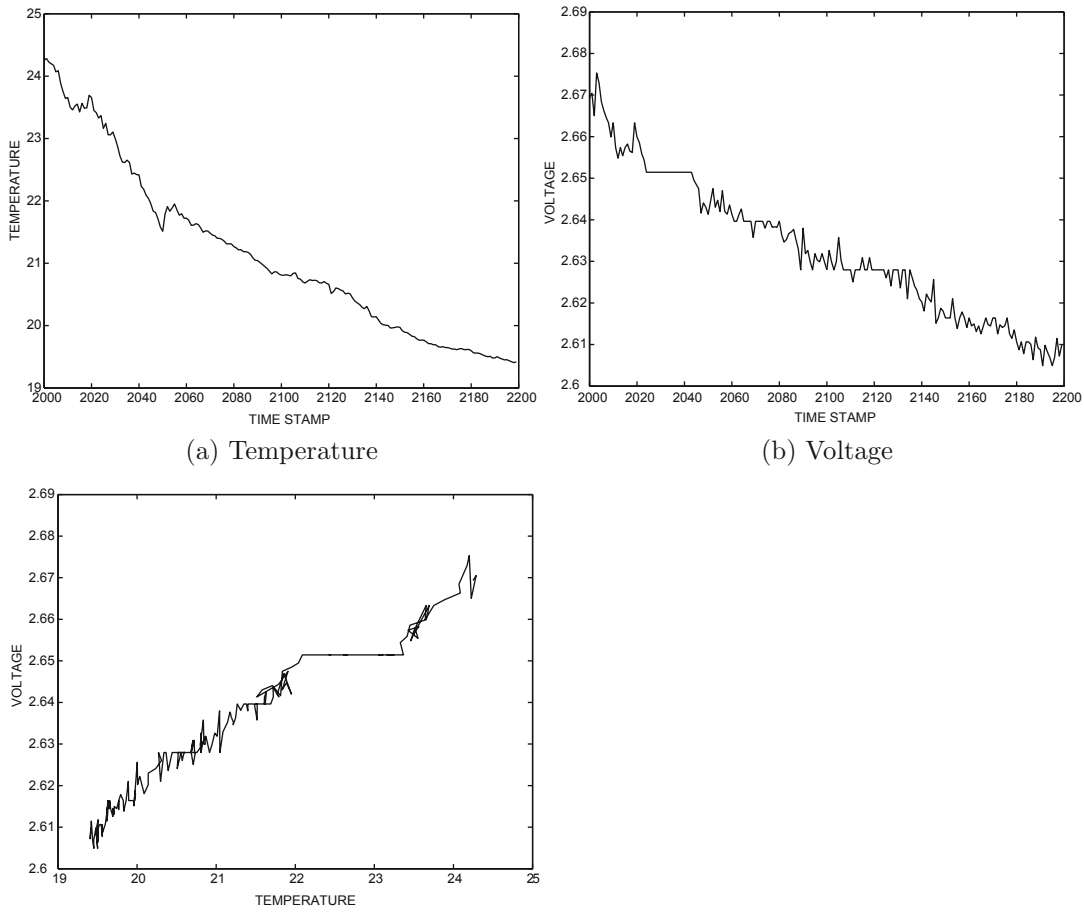
2. *The temporal attribute is contextual, whereas the spatial attributes are behavioral:* Strictly speaking, this kind of data can also be considered time-series data. However, the spatial nature of the behavioral attributes also provides better interpretability and more focused analysis in many scenarios. The most common form of this data arises in the context of *trajectory analysis*.

It should be pointed out that any 2- or 3-dimensional time-series data can be mapped onto trajectories. This is a useful transformation because it implies that trajectory mining algorithms can also be used for 2- or 3-dimensional time-series data. For example, the *Intel Research Berkeley data set* [556] contains readings from a variety of sensors. An example of a pair of readings from a temperature and voltage sensor are illustrated in Figs. 1.2a and b, respectively. The corresponding temperature–voltage trajectory is illustrated in Fig. 1.2c. Methods for spatial and spatiotemporal data mining are discussed in Chap. 16.

1.3.2.4 Network and Graph Data

In network and graph data, the data values may correspond to nodes in the network, whereas the relationships among the data values may correspond to the edges in the network. In some cases, attributes may be associated with nodes in the network. Although it is also possible to associate attributes with edges in the network, it is much less common to do so.

Definition 1.3.5 (Network Data) *A network $G = (N, A)$ contains a set of nodes N and a set of edges A, where the edges in A represent the relationships between the nodes. In*



(a) Temperature

(b) Voltage

(c) Temperature-voltage trajectory

Figure 1.2: Mapping of multivariate time series to trajectory data

some cases, an attribute set $\overline{X_i}$ may be associated with node i, or an attribute set $\overline{Y_{ij}}$ may be associated with edge (i, j).

The edge (i, j) may be directed or undirected, depending on the application at hand. For example, the Web graph may contain directed edges corresponding to directions of hyperlinks between pages, whereas friendships in the Facebook social network are undirected.

A second class of graph mining problems is that of a database containing many small graphs such as chemical compounds. The challenges in these two classes of problems are very different. Some examples of data that are represented as graphs are as follows:

- *Web graph:* The nodes correspond to the Web pages, and the edges correspond to hyperlinks. The nodes have text attributes corresponding to the content in the page.
- *Social networks:* In this case, the nodes correspond to social network actors, whereas the edges correspond to friendship links. The nodes may have attributes corresponding to social page content. In some specialized forms of social networks, such as email or

chat-messenger networks, the edges may have content associated with them. This content corresponds to the communication between the different nodes.

- *Chemical compound databases:* In this case, the nodes correspond to the elements and the edges correspond to the chemical bonds between the elements. The structures in these chemical compounds are very useful for identifying important reactive and pharmacological properties of these compounds.

Network data are a very general representation and can be used for solving many similarity-based applications on other data types. For example, multidimensional data may be converted to network data by creating a node for each record in the database, and representing similarities between nodes by edges. Such a representation is used quite often for many similarity-based data mining applications, such as clustering. It is possible to use community detection algorithms to determine clusters in the network data and then map them back to multidimensional data. Some spectral clustering methods, discussed in Chap. 19, are based on this principle. This generality of network data comes at a price. The development of mining algorithms for network data is generally more difficult. Methods for mining network data are discussed in Chaps. 17, 18, and 19.

1.4 The Major Building Blocks: A Bird's Eye View

As discussed in the introduction Sect. 1.1, four problems in data mining are considered fundamental to the mining process. These problems correspond to clustering, classification, association pattern mining, and outlier detection, and they are encountered repeatedly in the context of many data mining applications. What makes these problems so special? Why are they encountered repeatedly? To answer these questions, one must understand the nature of the typical relationships that data scientists often try to extract from the data.

Consider a multidimensional database $\mathcal{D}$ with n records, and d attributes. Such a database $\mathcal{D}$ may be represented as an $n \times d$ matrix D, in which each row corresponds to one record and each column corresponds to a dimension. We generally refer to this matrix as the *data matrix*. This book will use the notation of a data matrix D, and a database $\mathcal{D}$ interchangeably. Broadly speaking, data mining is all about finding summary relationships between the entries in the data matrix that are either unusually frequent or unusually infrequent. Relationships between data items are one of two kinds:

- *Relationships between columns:* In this case, the frequent or infrequent relationships between the values in a particular row are determined. This maps into either the positive or negative association pattern mining problem, though the former is more commonly studied. In some cases, one particular column of the matrix is considered more important than other columns because it represents a target attribute of the data mining analyst. In such cases, one tries to determine how the relationships in the other columns relate to this special column. Such relationships can be used to predict the value of this special column, when the value of that special column is unknown. This problem is referred to as *data classification*. A mining process is referred to as *supervised* when it is based on treating a particular attribute as special and predicting it.

- *Relationships between rows:* In these cases, the goal is to determine subsets of rows, in which the values in the corresponding columns are related. In cases where these subsets are similar, the corresponding problem is referred to as *clustering*. On the other hand,

when the entries in a row are very different from the corresponding entries in other rows, then the corresponding row becomes interesting as an unusual data point, or as an *anomaly*. This problem is referred to as *outlier analysis*. Interestingly, the clustering problem is closely related to that of classification, in that the latter can be considered a supervised version of the former. The discrete values of a special column in the data correspond to the group identifiers of different *desired* or *supervised* groups of application-specific similar records in the data. For example, when the special column corresponds to whether or not a customer is interested in a particular product, this represents the two groups in the data that one is interested in *learning*, with the use of *supervision*. The term "supervision" refers to the fact that the special column is used to direct the data mining process in an application-specific way, just as a teacher may supervise his or her student toward a specific goal.

Thus, these four problems are important because they seem to cover an exhaustive range of scenarios representing different kinds of positive, negative, supervised, or unsupervised relationships between the entries of the data matrix. These problems are also related to one another in a variety of ways. For example, association patterns may be considered indirect representations of (overlapping) clusters, where each pattern corresponds to a cluster of data points of which it is a subset.

It should be pointed out that the aforementioned discussion assumes the (most commonly encountered) multidimensional data type, although these problems continue to retain their relative importance for more complex data types. However, the more complex data types have a wider variety of problem formulations associated with them because of their greater complexity. This issue will be discussed in detail later in this section.

It has consistently been observed that many application scenarios determine such relationships between rows and columns of the data matrix as an intermediate step. This is the reason that a good understanding of these building-block problems is so important for the data mining process. Therefore, the first part of this book will focus on these problems in detail before generalizing to complex scenarios.

1.4.1 Association Pattern Mining

In its most primitive form, the association pattern mining problem is defined in the context of *sparse binary databases*, where the data matrix contains only 0/1 entries, and most entries take on the value of 0. Most customer transaction databases are of this type. For example, if each column in the data matrix corresponds to an item, and a customer transaction represents a row, the (i, j)th entry is 1, if customer transaction i contains item j as one of the items that was bought. A particularly commonly studied version of this problem is the frequent pattern mining problem or, more generally, the association pattern mining problem. In terms of the binary data matrix, the frequent pattern mining problem may be formally defined as follows:

Definition 1.4.1 (Frequent Pattern Mining) *Given a binary $n \times d$ data matrix D, determine all subsets of columns such that all the values in these columns take on the value of 1 for at least a fraction s of the rows in the matrix. The relative frequency of a pattern is referred to as its support. The fraction s is referred to as the minimum support.*

Patterns that satisfy the minimum support requirement are often referred to as *frequent patterns*, or *frequent itemsets*. Frequent patterns represent an important class of association patterns. Many other definitions of relevant association patterns are possible that do not use

absolute frequencies but use other statistical quantifications such as the χ^2 measure. These measures often lead to generation of more *interesting* rules from a statistical perspective. Nevertheless, this particular definition of association pattern mining has become the most popular one in the literature because of the ease in developing algorithms for it. This book therefore refers to this problem as *association pattern mining* as opposed to *frequent pattern mining*.

For example, if the columns of the data matrix D corresponding to *Bread*, *Butter*, and *Milk* take on the value of 1 together frequently in a customer transaction database, then it implies that these items are often bought together. This is very useful information for the merchant from the perspective of physical placement of the items in the store, or from the perspective of product promotions. Association pattern mining is not restricted to the case of binary data and can be easily generalized to quantitative and numeric attributes by using appropriate data transformations, which will be discussed in Chap. 4.

Association pattern mining was originally proposed in the context of *association rule mining*, where an additional step was included based on a measure known as the *confidence* of the rule. For example, consider two sets of items A and B. The confidence of the rule $A \Rightarrow B$ is defined as the fraction of transactions containing A, which also contain B. In other words, the confidence is obtained by dividing the support of the pattern $A \cup B$ with the support of pattern A. A combination of support and confidence is used to define association rules.

Definition 1.4.2 (Association Rules) *Let A and B be two sets of items. The rule $A \Rightarrow B$ is said to be valid at support level s and confidence level c, if the following two conditions are satisfied:*

1. *The support of the item set A is at least s.*
2. *The confidence of $A \Rightarrow B$ is at least c.*

By incorporating supervision in association rule mining algorithms, it is possible to provide solutions for the classification problem. Many variations of association pattern mining are also related to clustering and outlier analysis. This is a natural consequence of the fact that horizontal and vertical analysis of the data matrix are often related to one another. In fact, many variations of the association pattern mining problem are used as a subroutine to solve the clustering, outlier analysis, and classification problems. These issues will be discussed in Chaps. 4 and 5.

1.4.2 Data Clustering

A rather broad and informal definition of the clustering problem is as follows:

Definition 1.4.3 (Data Clustering) *Given a data matrix D (database $\mathcal{D}$), partition its rows (records) into sets $\mathcal{C}_1 \ldots \mathcal{C}_k$, such that the rows (records) in each cluster are "similar" to one another.*

We have intentionally provided an informal definition here because clustering allows a wide variety of definitions of similarity, some of which are not cleanly defined in closed form by a similarity function. A clustering problem can often be defined as an optimization problem, in which the variables of the optimization problem represent cluster memberships of data points, and the objective function maximizes a concrete mathematical quantification of intragroup similarity in terms of these variables.

An important part of the clustering process is the design of an appropriate similarity function for the computation process. Clearly, the computation of similarity depends heavily on the underlying data type. The issue of similarity computation will be discussed in detail in Chap. 3. Some examples of relevant applications are as follows:

- *Customer segmentation:* In many applications, it is desirable to determine customers that are similar to one another in the context of a variety of product promotion tasks. The segmentation phase plays an important role in this process.
- *Data summarization:* Because clusters can be considered similar groups of records, these similar groups can be used to create a summary of the data.
- *Application to other data mining problems:* Because clustering is considered an unsupervised version of classification, it is often used as a building block to solve the latter. Furthermore, this problem is also used in the context of the outlier analysis problem, as discussed below.

The data clustering problem is discussed in detail in Chaps. 6 and 7.

1.4.3 Outlier Detection

An outlier is a data point that is significantly different from the remaining data. Hawkins formally defined [259] the concept of an outlier as follows:
"*An outlier is an observation that deviates so much from the other observations as to arouse suspicions that it was generated by a different mechanism.*"

Outliers are also referred to as *abnormalities*, *discordants*, *deviants*, or *anomalies* in the data mining and statistics literature. In most applications, the data are created by one or more generating processes that can either reflect activity in the system or observations collected about entities. When the generating process behaves in an unusual way, it results in the creation of outliers. Therefore, an outlier often contains useful information about abnormal characteristics of the systems and entities that impact the data-generation process. The recognition of such unusual characteristics provides useful application-specific insights. The outlier detection problem is informally defined in terms of the data matrix as follows:

Definition 1.4.4 (Outlier Detection) *Given a data matrix D, determine the rows of the data matrix that are very different from the remaining rows in the matrix.*

The outlier detection problem is related to the clustering problem by complementarity. This is because outliers correspond to dissimilar data points from the main groups in the data. On the other hand, the main groups in the data are clusters. In fact, a simple methodology to determine outliers uses clustering as an intermediate step. Some examples of relevant applications are as follows:

- *Intrusion-detection systems:* In many networked computer systems, different kinds of data are collected about the operating system calls, network traffic, or other activity in the system. These data may show unusual behavior because of malicious activity. The detection of such activity is referred to as intrusion detection.
- *Credit card fraud:* Unauthorized use of credit cards may show different patterns, such as a buying spree from geographically obscure locations. Such patterns may show up as outliers in credit card transaction data.

- *Interesting sensor events:* Sensors are often used to track various environmental and location parameters in many real applications. The sudden changes in the underlying patterns may represent events of interest. Event detection is one of the primary motivating applications in the field of sensor networks.

- *Medical diagnosis:* In many medical applications, the data are collected from a variety of devices such as magnetic resonance imaging (MRI), positron emission tomography (PET) scans, or electrocardiogram (ECG) time series. Unusual patterns in such data typically reflect disease conditions.

- *Law enforcement:* Outlier detection finds numerous applications in law enforcement, especially in cases where unusual patterns can only be discovered over time through multiple actions of an entity. The identification of fraud in financial transactions, trading activity, or insurance claims typically requires the determination of unusual patterns in the data generated by the actions of the criminal entity.

- *Earth science:* A significant amount of spatiotemporal data about weather patterns, climate changes, or land-cover patterns is collected through a variety of mechanisms such as satellites or remote sensing. Anomalies in such data provide significant insights about hidden human or environmental trends that may have caused such anomalies.

The outlier detection problem is studied in detail in Chaps. 8 and 9.

1.4.4 Data Classification

Many data mining problems are directed toward a specialized goal that is sometimes represented by the value of a particular feature in the data. This particular feature is referred to as the *class label*. Therefore, such problems are *supervised*, wherein the relationships of the remaining features in the data with respect to this special feature are *learned*. The data used to learn these relationships is referred to as the *training data*. The *learned model* may then be used to determine the estimated class labels for records, where the label is missing.

For example, in a target marketing application, each record may be tagged by a particular *label* that represents the interest (or lack of it) of the customer toward a particular product. The labels associated with customers may have been derived from the previous buying behavior of the customer. In addition, a set of features corresponding the customer demographics may also be available. The goal is to predict whether or not a customer, whose buying behavior is unknown, will be interested in a particular product by relating the demographic features to the class label. Therefore, a *training model* is constructed, which is then used to predict class labels. The classification problem is informally defined as follows:

Definition 1.4.5 (Data Classification) *Given an $n \times d$ training data matrix D (database $\mathcal{D}$), and a class label value in $\{1 \ldots k\}$ associated with each of the n rows in D (records in $\mathcal{D}$), create a training model $\mathcal{M}$, which can be used to predict the class label of a d-dimensional record $\overline{Y} \notin \mathcal{D}$.*

The record whose class label is unknown is referred to as the *test record*. It is interesting to examine the relationship between the clustering and the classification problems. In the case of the clustering problem, the data are partitioned into k groups on the basis of similarity. In the case of the classification problem, a (test) record is also categorized into one of k groups, except that this is achieved by learning a model from a training database $\mathcal{D}$, rather than on the basis of similarity. In other words, the supervision from the training data redefines the

notion of a group of "similar" records. Therefore, from a learning perspective, clustering is often referred to as *unsupervised learning* (because of the lack of a special training database to "teach" the model about the notion of an appropriate grouping), whereas the classification problem is referred to as *supervised learning*.

The classification problem is related to association pattern mining, in the sense that the latter problem is often used to solve the former. This is because if the entire training database (including the class label) is treated as an $n \times (d+1)$ matrix, then frequent patterns containing the class label in this matrix provide useful hints about the correlations of other features to the class label. In fact, many forms of classifiers, known as *rule-based classifiers*, are based on this broader principle.

The classification problem can be mapped to a specific version of the outlier detection problem, by incorporating supervision in the latter. While the outlier detection problem is assumed to be unsupervised by default, many variations of the problem are either partially or fully supervised. In supervised outlier detection, some examples of outliers are available. Thus, such data records are tagged to belong to a *rare class*, whereas the remaining data records belong to the *normal class*. Thus, the supervised outlier detection problem maps to a binary classification problem, with the caveat that the class labels are highly *imbalanced*.

The incorporation of supervision makes the classification problem unique in terms of its *direct* application specificity due to its use of application-specific class labels. Compared to the other major data mining problems, the classification problem is relatively self-contained. For example, the clustering and frequent pattern mining problem are more often used as intermediate steps in larger application frameworks. Even the outlier analysis problem is sometimes used in an exploratory way. On the other hand, the classification problem is often used directly as a stand-alone tool in many applications. Some examples of applications where the classification problem is used are as follows:

- *Target marketing:* Features about customers are related to their buying behavior with the use of a training model.
- *Intrusion detection:* The sequences of customer activity in a computer system may be used to predict the possibility of intrusions.
- *Supervised anomaly detection:* The rare class may be differentiated from the normal class when previous examples of outliers are available.

The data classification problem is discussed in detail in Chaps. 10 and 11.

1.4.5 Impact of Complex Data Types on Problem Definitions

The specific data type has a profound impact on the kinds of problems that may be defined. In particular, in dependency-oriented data types, the dependencies often play a critical role in the problem definition, the solution, or both. This is because the contextual attributes and dependencies are often fundamental to how the data may be evaluated. Furthermore, because complex data types are much richer, they allow the formulation of novel problem definitions that may not even exist in the context of multidimensional data. A tabular summary of the different variations of data mining problems for dependency-oriented data types is provided in Table 1.2. In the following, a brief review will be provided as to how the different problem definitions are affected by data type.

Table 1.2: Some examples of variation in problem definition with data type

Problem	Time series	Spatial	Sequence	Networks
Patterns	Motif-mining Periodic pattern	Colocation patterns	Sequential patterns Periodic Sequence	Structural patterns
	Trajectory patterns			
Clustering	Shape clusters	Spatial clusters	Sequence clusters	Community detection
	Trajectory clusters			
Outliers	Position outlier Shape outlier	Position outlier Shape outlier	Position outlier Combination outlier	Node outlier Linkage outlier Community outliers
	Trajectory outliers			
Classification	Position classification Shape classification	Position classification Shape classification	Position classification Sequence classification	Collective classification Graph classification
	Trajectory classification			

1.4.5.1 Pattern Mining with Complex Data Types

The association pattern mining problem generally determines the patterns from the underlying data in the form of sets; however, this is not the case when dependencies are present in the data. This is because the dependencies and relationships often impose ordering among data items, and the direct use of frequent pattern mining methods fails to recognize the relationships among the different data values. For example, when a larger number of time series are made available, they can be used to determine different kinds of *temporally* frequent patterns, in which a temporal ordering is imposed on the items in the pattern. Furthermore, because of the presence of the additional contextual attribute representing time, temporal patterns may be defined in a much richer way than a set-based pattern as in association pattern mining. The patterns may be temporally contiguous, as in *time-series motifs*, or they may be periodic, as in *periodic patterns*. Some of these methods for temporal pattern mining will be discussed in Chap. 14. A similar analogy exists for the case of discrete sequence mining, except that the individual pattern constituents are categorical, as opposed to continuous. It is also possible to define 2-dimensional motifs for the spatial scenario, and such a formulation is useful for image processing. Finally, structural patterns are commonly defined in networks that correspond to frequent subgraphs in the data. Thus, the dependencies between the nodes are included within the definition of the patterns.

1.4.5.2 Clustering with Complex Data Types

The techniques used for clustering are also affected significantly by the underlying data type. Most importantly, the similarity function is significantly affected by the data type. For example, in the case of time series, sequential, or graph data, the similarity between a pair of time series cannot be easily defined by using straightforward metrics such as the Euclidean metric. Rather, it is necessary to use other kinds of metrics, such as the edit distance or structural similarity. In the context of spatial data, trajectory clustering is particularly useful in finding the relevant patterns for mobile data, or for multivariate

time series. For network data, the clustering problem discovers densely connected groups of nodes, and is also referred to as *community detection*.

1.4.5.3 Outlier Detection with Complex Data Types

Dependencies can be used to define expected values of data items. Deviations from these expected values are outliers. For example, a sudden jump in the value of a time series will result in a position outlier at the specific spot at which the jump occurs. The idea in these methods is to use *prediction-based techniques* to forecast the value at that position. Significant deviation from the prediction is reported as a *position outlier*. Such outliers can be defined in the context of time-series, spatial, and sequential data, where significant deviations from the corresponding neighborhoods can be detected using autoregressive, Markovian, or other models. In the context of graph data, outliers may correspond to unusual properties of nodes, edges, or entire subgraphs. Thus, the complex data types show significant richness in terms of how outliers may be defined.

1.4.5.4 Classification with Complex Data Types

The classification problem also shows a significant amount of variation in the different complex data types. For example, class labels can be attached to specific positions in a series, or they can be attached to the entire series. When the class labels are attached to a specific position in the series, this can be used to perform supervised event detection, where the first occurrence of an event-specific label (e.g., the breakdown of a machine as suggested by the underlying temperature and pressure sensor) of a particular series represents the occurrence of the event. For the case of network data, the labels may be attached to individual nodes in a very large network, or to entire graphs in a collection of multiple graphs. The former case corresponds to the classification of nodes in a social network, and is also referred to as *collective classification*. The latter case corresponds to the chemical compound classification problem, in which labels are attached to compounds on the basis of their chemical properties.

1.5 Scalability Issues and the Streaming Scenario

Scalability is an important concern in many data mining applications due to the increasing sizes of the data in modern-day applications. Broadly speaking, there are two important scenarios for scalability:

1. The data are stored on one or more machines, but it is too large to process efficiently. For example, it is easy to design efficient algorithms in cases where the entire data can be maintained in main memory. When the data are stored on disk, it is important to be design the algorithms in such a way that random access to the disk is minimized. For very large data sets, big data frameworks, such as MapReduce, may need to be used. This book will touch upon this kind of scalability at the level of disk-resident processing, where needed.

2. The data are generated continuously over time in high volume, and it is not practical to store it entirely. This scenario is that of *data streams*, in which the data need to be processed with the use of an online approach.

The latter scenario requires some further exposition. The streaming scenario has become increasingly popular because of advances in data collection technology that can collect large amounts of data over time. For example, simple transactions of everyday life such as using a credit card or the phone may lead to automated data collection. In such cases, the volume of the data is so large that it may be impractical to store directly. Rather, all algorithms must be executed in a single pass over the data. The major challenges that arise in the context of data stream processing are as follows:

1. *One-pass constraint:* The algorithm needs to process the entire data set in one pass. In other words, after a data item has been processed and the relevant summary insights have been gleaned, the raw item is discarded and is no longer available for processing. The amount of data that may be processed at a given time depends on the storage available for retaining segments of the data.
2. *Concept drift:* In most applications, the data distribution changes over time. For example, the pattern of sales in a given hour of a day may not be similar to that at another hour of the day. This leads to changes in the output of the mining algorithms as well.

It is often challenging to design algorithms for such scenarios because of the varying rates at which the patterns in the data may change over time and the continuously evolving patterns in the underlying data. Methods for stream mining are addressed in Chap. 12.

1.6 A Stroll Through Some Application Scenarios

In this section, some common application scenarios will be discussed. The goal is to illustrate the wide diversity of problems and applications, and how they might map onto some of the building blocks discussed in this chapter.

1.6.1 Store Product Placement

The application scenario may be stated as follows:

Application 1.6.1 (Store Product Placement) *A merchant has a set of d products together with previous transactions from the customers containing baskets of items bought together. The merchant would like to know how to place the product on the shelves to increase the likelihood that items that are frequently bought together are placed on adjacent shelves.*

This problem is closely related to frequent pattern mining because the analyst can use the frequent pattern mining problem to determine groups of items that are frequently bought together at a particular support level. An important point to note here is that the determination of the frequent patterns, while providing useful insights, does not provide the merchant with precise guidance in terms of how the products may be placed on the different shelves. This situation is quite common in data mining. The building block problems often do not directly solve the problem at hand. In this particular case, the merchant may choose from a variety of heuristic ideas in terms of how the products may be stocked on the different shelves. For example, the merchant may already have an existing placement, and may use the frequent patterns to create a numerical score for the quality of the placement. This placement can be successively optimized by making incremental changes to the current placement. With an appropriate initialization methodology, the frequent pattern mining approach can be leveraged as a very useful subroutine for the problem. These parts of data mining are often application-specific and show such wide variations across different domains that they can only be learned through practical experience.

1.6.2 Customer Recommendations

This is a very commonly encountered problem in the data mining literature. Many variations of this problem exist, depending on the kind of input data available to that application. In the following, we will examine a particular instantiation of the recommendation problem and a straw-man solution.

Application 1.6.2 (Product Recommendations) *A merchant has an $n \times d$ binary matrix D representing the buying behavior of n customers across d items. It is assumed that the matrix is sparse, and therefore each customer may have bought only a few items. It is desirable to use the product associations to make recommendations to customers.*

This problem is a simple version of the collaborative filtering problem that is widely studied in the data mining and recommendation literature. There are literally hundreds of solutions to the vanilla version of this problem, and we provide three sample examples of varying complexity below:

1. A simple solution is to use association rule mining at particular levels of support and confidence. For a particular customer, the relevant rules are those in which all items in the left-hand side were previously bought by this customer. Items that appear frequently on the right-hand side of the relevant rules are reported.

2. The previous solution does not use the similarity across different customers to make recommendations. A second solution is to determine the most similar rows to a target customer, and then recommend the most common item occurring in these similar rows.

3. A final solution is to use clustering to create segments of similar customers. Within each similar segment, association pattern mining may be used to make recommendations.

Thus, there can be multiple ways of solving a particular problem corresponding to different analytical paths. These different paths may use different kinds of building blocks, which are all useful in different parts of the data mining process.

1.6.3 Medical Diagnosis

Medical diagnosis has become a common application in the context of data mining. The data types in medical diagnosis tend to be complex, and may correspond to image, time-series, or discrete sequence data. Thus, dependency-oriented data types tend to be rather common in medical diagnosis applications. A particular case is that of ECG readings from heart patients.

Application 1.6.3 (Medical ECG Diagnosis) *Consider a set of ECG time series that are collected from different patients. It is desirable to determine the anomalous series from this set.*

This application can be mapped to different problems, depending upon the nature of the input data available. For example, consider the case where no previous examples of anomalous ECG series are available. In such cases, the problem can be mapped to the outlier detection problem. A time series that differs significantly from the remaining series in the data may be considered an outlier. However, the solution methodology changes significantly

if previous examples of normal and anomalous series are available. In such cases, the problem maps to a classification problem on time-series data. Furthermore, the class labels are likely to be imbalanced because the number of abnormal series are usually far fewer than the number of normal series.

1.6.4 Web Log Anomalies

Web logs are commonly collected at the hosts of different Web sites. Such logs can be used to detect unusual, suspicious, or malicious activity at the site. Financial institutions regularly analyze the logs at their site to detect intrusion attempts.

Application 1.6.4 (Web Log Anomalies) *A set of Web logs is available. It is desired to determine the anomalous sequences from the Web logs.*

Because the data are typically available in the form of raw logs, a significant amount of data cleaning is required. First, the raw logs need to be transformed into sequences of symbols. These sequences may then need to be decomposed into smaller windows to analyze the sequences at a particular level of granularity. Anomalous sequences may be determined by using a sequence clustering algorithm, and then determining the sequences that do not lie in these clusters [5]. If it is desired to find specific positions that correspond to anomalies, then more sophisticated methods such as Markovian models may be used to determine the anomalies [5].

As in the previous case, the analytical phase of this problem can be modeled differently, depending on whether or not examples of Web log anomalies are available. If no previous examples of Web log anomalies are available, then this problem maps to the unsupervised temporal outlier detection problem. Numerous methods for solving the unsupervised case for the temporal outlier detection problem are introduced in [5]. The topic is also briefly discussed in Chaps. 14 and 15 of this book. On the other hand, when examples of previous anomalies are available, then the problem maps to the rare class-detection problem. This problem is discussed in [5] as well, and in Chap. 11 of this book.

1.7 Summary

Data mining is a complex and multistage process. These different stages are data collection, preprocessing, and analysis. The data preprocessing phase is highly application-specific because the different formats of the data require different algorithms to be applied to them. The processing phase may include data integration, cleaning, and feature extraction. In some cases, feature selection may also be used to sharpen the data representation. After the data have been converted to a convenient format, a variety of analytical algorithms can be used.

A number of data mining building blocks are often used repeatedly in a wide variety of application scenarios. These correspond to the frequent pattern mining, clustering, outlier analysis, and classification problems, respectively. The final design of a solution for a particular data mining problem is dependent on the skill of the analyst in mapping the application to the different building blocks, or in using novel algorithms for a specific application. This book will introduce the fundamentals required for gaining such analytical skills.

1.8 Bibliographic Notes

The problem of data mining is generally studied by multiple research communities corresponding to statistics, data mining, and machine learning. These communities are highly overlapping and often share many researchers in common. The machine learning and statistics communities generally approach data mining from a theoretical and statistical perspective. Some good books written in this context may be found in [95, 256, 389]. However, because the machine learning community is generally focused on supervised learning methods, these books are mostly focused on the classification scenario. More general data mining books, which are written from a broader perspective, may be found in [250, 485, 536]. Because the data mining process often has to interact with databases, a number of relevant database textbooks [434, 194] provide knowledge about data representation and integration issues.

A number of books have also been written on each of the major areas of data mining. The frequent pattern mining problem and its variations have been covered in detail in [34]. Numerous books have been written on the topic of data clustering. A well-known data clustering book [284] discusses the classical techniques from the literature. Another book [219] discusses the more recent methods for data clustering, although the material is somewhat basic. The most recent book [32] in the literature provides a very comprehensive overview of the different data clustering algorithms. The problem of data classification has been addressed in the standard machine learning books [95, 256, 389]. The classification problem has also been studied extensively by the pattern recognition community [189]. More recent surveys on the topic may be found in [33]. The problem of outlier detection has been studied in detail in [89, 259]. These books are, however, written from a statistical perspective and do not address the problem from the perspective of the computer science community. The problem has been addressed from the perspective of the computer science community in [5].

1.9 Exercises

1. An analyst collects surveys from different participants about their likes and dislikes. Subsequently, the analyst uploads the data to a database, corrects erroneous or missing entries, and designs a recommendation algorithm on this basis. Which of the following actions represent data collection, data preprocessing, and data analysis? (a) Conducting surveys and uploading to database, (b) correcting missing entries, (c) designing a recommendation algorithm.

2. What is the data type of each of the following kinds of attributes (a) *Age*, (b) *Salary*, (c) *ZIP code*, (d) *State of residence*, (e) *Height*, (f) *Weight*?

3. An analyst obtains medical notes from a physician for data mining purposes, and then transforms them into a table containing the medicines prescribed for each patient. What is the data type of (a) the original data, and (b) the transformed data? (c) What is the process of transforming the data to the new format called?

4. An analyst sets up a sensor network in order to measure the temperature of different locations over a period. What is the data type of the data collected?

5. The same analyst as discussed in Exercise 4 above finds another database from a different source containing pressure readings. She decides to create a single database

containing her own readings and the pressure readings. What is the process of creating such a single database called?

6. An analyst processes Web logs in order to create records with the ordering information for Web page accesses from different users. What is the type of this data?

7. Consider a data object corresponding to a set of nucleotides arranged in a certain order. What is this type of data?

8. It is desired to partition customers into similar groups on the basis of their demographic profile. Which data mining problem is best suited to this task?

9. Suppose in Exercise 8, the merchant already knows for *some* of the customers whether or not they have bought widgets. Which data mining problem would be suited to the task of identifying groups among the remaining customers, who *might* buy widgets in the future?

10. Suppose in Exercise 9, the merchant also has information for other items bought by the customers (beyond widgets). Which data mining problem would be best suited to finding sets of items that are often bought together with widgets?

11. Suppose that a small number of customers lie about their demographic profile, and this results in a mismatch between the buying behavior and the demographic profile, as suggested by comparison with the remaining data. Which data mining problem would be best suited to finding such customers?

Chapter 2

Data Preparation

"*Success depends upon previous preparation, and without such preparation there is sure to be failure.*"—Confucius

2.1 Introduction

The raw format of real data is usually widely variable. Many values may be missing, inconsistent across different data sources, and erroneous. For the analyst, this leads to numerous challenges in using the data effectively. For example, consider the case of evaluating the interests of consumers from their activity on a social media site. The analyst may first need to determine the types of activity that are valuable to the mining process. The activity might correspond to the interests entered by the user, the comments entered by the user, and the set of friendships of the user along with their interests. All these pieces of information are diverse and need to be collected from different databases within the social media site. Furthermore, some forms of data, such as raw logs, are often not directly usable because of their unstructured nature. In other words, useful features need to be extracted from these data sources. Therefore, a *data preparation phase* is needed.

The data preparation phase is a multistage process that comprises several individual steps, some or all of which may be used in a given application. These steps are as follows:

1. *Feature extraction and portability:* The raw data is often in a form that is not suitable for processing. Examples include raw logs, documents, semistructured data, and possibly other forms of heterogeneous data. In such cases, it may be desirable to derive meaningful features from the data. Generally, features with good semantic interpretability are more desirable because they simplify the ability of the analyst to understand intermediate results. Furthermore, they are usually better tied to the goals of the data mining application at hand. In some cases where the data is obtained from multiple sources, it needs to be integrated into a single database for processing. In addition, some algorithms may work only with a specific data type, whereas the data may contain heterogeneous types. In such cases, *data type portability* becomes

C. C. Aggarwal, *Data Mining: The Textbook*, DOI 10.1007/978-3-319-14142-8_2

important where attributes of one type are transformed to another. This results in a more homogeneous data set that can be processed by existing algorithms.

2. *Data cleaning:* In the data cleaning phase, missing, erroneous, and inconsistent entries are removed from the data. In addition, some missing entries may also be estimated by a process known as *imputation*.

3. *Data reduction, selection, and transformation:* In this phase, the size of the data is reduced through data subset selection, feature subset selection, or data transformation. The gains obtained in this phase are twofold. First, when the size of the data is reduced, the algorithms are generally more efficient. Second, if irrelevant features or irrelevant records are removed, the quality of the data mining process is improved. The first goal is achieved by generic sampling and dimensionality reduction techniques. To achieve the second goal, a highly problem-specific approach must be used for feature selection. For example, a feature selection approach that works well for clustering may not work well for classification.

Some forms of feature selection are tightly integrated with the problem at hand. Later chapters on specific problems such as clustering and classification will contain detailed discussions on feature selection.

This chapter is organized as follows. The feature extraction phase is discussed in Sect. 2.2. The data cleaning phase is covered in Sect. 2.3. The data reduction phase is explained in Sect. 2.4. A summary is given in Sect. 2.5.

2.2 Feature Extraction and Portability

The first phase of the data mining process is creating a set of features that the analyst can work with. In cases where the data is in raw and unstructured form (e.g., raw text, sensor signals), the relevant features need to be extracted for processing. In other cases where a heterogeneous mixture of features is available in different forms, an "off-the-shelf" analytical approach is often not available to process such data. In such cases, it may be desirable to transform the data into a uniform representation for processing. This is referred to as *data type porting*.

2.2.1 Feature Extraction

The first phase of feature extraction is a crucial one, though it is very application specific. In some cases, feature extraction is closely related to the concept of data type portability, where low-level features of one type may be transformed to higher-level features of another type. The nature of feature extraction depends on the domain from which the data is drawn:

1. *Sensor data:* Sensor data is often collected as large volumes of low-level signals, which are massive. The low-level signals are sometimes converted to higher-level features using wavelet or Fourier transforms. In other cases, the time series is used directly after some cleaning. The field of signal processing has an extensive literature devoted to such methods. These technologies are also useful for porting time-series data to multidimensional data.

2. *Image data:* In its most primitive form, image data are represented as pixels. At a slightly higher level, color histograms can be used to represent the features in different segments of an image. More recently, the use of *visual words* has become more

popular. This is a semantically rich representation that is similar to document data. One challenge in image processing is that the data are generally very high dimensional. Thus, feature extraction can be performed at different levels, depending on the application at hand.

3. *Web logs:* Web logs are typically represented as text strings in a prespecified format. Because the fields in these logs are clearly specified and separated, it is relatively easy to convert Web access logs into a multidimensional representation of (the relevant) categorical and numeric attributes.

4. *Network traffic:* In many intrusion-detection applications, the characteristics of the network packets are used to analyze intrusions or other interesting activity. Depending on the underlying application, a variety of features may be extracted from these packets, such as the number of bytes transferred, the network protocol used, and so on.

5. *Document data:* Document data is often available in raw and unstructured form, and the data may contain rich linguistic relations between different entities. One approach is to remove stop words, stem the data, and use a bag-of-words representation. Other methods use *entity extraction* to determine linguistic relationships.

   Named-entity recognition is an important subtask of information extraction. This approach locates and classifies atomic elements in text into predefined expressions of names of persons, organizations, locations, actions, numeric quantities, and so on. Clearly, the ability to identify such atomic elements is very useful because they can be used to understand the structure of sentences and complex events. Such an approach can also be used to populate a more conventional database of relational elements or as a sequence of atomic entities, which is more easily analyzed. For example, consider the following sentence:

   ```
   Bill Clinton lives in Chappaqua.
   ```

   Here, "Bill Clinton" is the name of a person, and "Chappaqua" is the name of a place. The word "lives" denotes an action. Each type of entity may have a different significance to the data mining process depending on the application at hand. For example, if a data mining application is mainly concerned with mentions of specific locations, then the word "Chappaqua" needs to be extracted.

   Popular techniques for named entity recognition include linguistic grammar-based techniques and statistical models. The use of grammar rules is typically very effective, but it requires work by experienced computational linguists. On the other hand, statistical models require a significant amount of training data. The techniques designed are very often domain-specific. The area of named entity recognition is vast in its own right, which is outside the scope of this book. The reader is referred to [400] for a detailed discussion of different methods for entity recognition.

Feature extraction is an art form that is highly dependent on the skill of the analyst to choose the features and their representation that are best suited to the task at hand. While this particular aspect of data analysis typically belongs to the domain expert, it is perhaps the most important one. If the correct features are not extracted, the analysis can only be as good as the available data.

2.2.2 Data Type Portability

Data type portability is a crucial element of the data mining process because the data is often heterogeneous, and may contain multiple types. For example, a demographic data set may contain both numeric and mixed attributes. A time-series data set collected from an electrocardiogram (ECG) sensor may have numerous other meta-information and text attributes associated with it. This creates a bewildering situation for an analyst who is now faced with the difficult challenge of designing an algorithm with an arbitrary combination of data types. The mixing of data types also restricts the ability of the analyst to use off-the-shelf tools for processing. Note that porting data types does lose representational accuracy and expressiveness in some cases. Ideally, it is best to customize the algorithm to the particular combination of data types to optimize results. This is, however, time-consuming and sometimes impractical.

This section will describe methods for converting between various data types. Because the numeric data type is the simplest and most widely studied one for data mining algorithms, it is particularly useful to focus on how different data types may be converted to it. However, other forms of conversion are also useful in many scenarios. For example, for similarity-based algorithms, it is possible to convert virtually any data type to a graph and apply graph-based algorithms to this representation. The following discussion, summarized in Table 2.1, will discuss various ways of transforming data across different types.

2.2.2.1 Numeric to Categorical Data: Discretization

The most commonly used conversion is from the numeric to the categorical data type. This process is known as *discretization*. The process of discretization divides the ranges of the numeric attribute into ϕ ranges. Then, the attribute is assumed to contain ϕ different categorical labeled values from 1 to ϕ, depending on the range in which the original attribute lies. For example, consider the age attribute. One could create ranges $[0, 10]$, $[11, 20]$, $[21, 30]$, and so on. The *symbolic* value for any record in the range $[11, 20]$ is "2" and the symbolic value for a record in the range $[21, 30]$ is "3". Because these are symbolic values, no ordering is assumed between the values "2" and "3". Furthermore, variations within a range are not distinguishable after discretization. Thus, the discretization process does lose some information for the mining process. However, for some applications, this loss of information is not too debilitating. One challenge with discretization is that the data may be nonuniformly distributed across the different intervals. For example, for the case of the salary attribute, a large subset of the population may be grouped in the $[40,000,\ 80,000]$ range, but very few will be grouped in the $[1,040,000,\ 1,080,000]$ range. Note that both ranges have the same size. Thus, the use of ranges of equal size may not be very helpful in discriminating between different data segments. On the other hand, many attributes, such as age, are not as nonuniformly distributed, and therefore ranges of equal size may work reasonably well. The discretization process can be performed in a variety of ways depending on application-specific goals:

1. *Equi-width ranges:* In this case, each range $[a, b]$ is chosen in such a way that $b - a$ is the same for each range. This approach has the drawback that it will not work for data sets that are distributed nonuniformly across the different ranges. To determine the actual values of the ranges, the minimum and maximum values of each attribute are determined. This range $[min, max]$ is then divided into ϕ ranges of equal length.

2. *Equi-log ranges:* Each range $[a, b]$ is chosen in such a way that $\log(b) - \log(a)$ has the same value. This kinds of range selection has the effect of geometrically increasing

Table 2.1: Portability of different data types

Source data type	Destination data type	Methods
Numeric	Categorical	Discretization
Categorical	Numeric	Binarization
Text	Numeric	Latent semantic analysis (LSA)
Time series	Discrete sequence	SAX
Time series	Numeric multidimensional	DWT, DFT
Discrete sequence	Numeric multidimensional	DWT, DFT
Spatial	Numeric multidimensional	2-d DWT
Graphs	Numeric multidimensional	MDS, spectral
Any type	Graphs	Similarity graph (Restricted applicability)

ranges $[a, a \cdot \alpha]$, $[a \cdot \alpha, a \cdot \alpha^2]$, and so on, for some $\alpha > 1$. This kind of range may be useful when the attribute shows an exponential distribution across a range. In fact, if the attribute frequency distribution for an attribute can be modeled in functional form, then a natural approach would be to select ranges $[a, b]$ such that $f(b) - f(a)$ is the same for some function $f(\cdot)$. The idea is to select this function $f(\cdot)$ in such a way that each range contains an approximately similar number of records. However, in most cases, it is hard to find such a function $f(\cdot)$ in closed form.

3. *Equi-depth ranges:* In this case, the ranges are selected so that each range has an equal number of records. The idea is to provide the same level of granularity to each range. An attribute can be divided into equi-depth ranges by first sorting it, and then selecting the division points on the sorted attribute value, such that each range contains an equal number of records.

The process of discretization can also be used to convert time-series data to discrete sequence data.

2.2.2.2 Categorical to Numeric Data: Binarization

In some cases, it is desirable to use numeric data mining algorithms on categorical data. Because binary data is a special form of both numeric and categorical data, it is possible to convert the categorical attributes to binary form and then use numeric algorithms on the binarized data. If a categorical attribute has ϕ different values, then ϕ different binary attributes are created. Each binary attribute corresponds to one possible value of the categorical attribute. Therefore, exactly one of the ϕ attributes takes on the value of 1, and the remaining take on the value of 0.

2.2.2.3 Text to Numeric Data

Although the vector-space representation of text can be considered a sparse numeric data set with very high dimensionality, this special numeric representation is not very amenable to conventional data mining algorithms. For example, one typically uses specialized similarity functions, such as the cosine, rather than the Euclidean distance for text data. This is the reason that text mining is a distinct area in its own right with its own family of specialized algorithms. Nevertheless, it is possible to convert a text collection into a form

that is more amenable to the use of mining algorithms for numeric data. The first step is to use *latent semantic analysis (LSA)* to transform the text collection to a nonsparse representation with lower dimensionality. Furthermore, after transformation, each document $\overline{X} = (x_1 \ldots x_d)$ needs to be scaled to $\frac{1}{\sqrt{\sum_{i=1}^{d} x_i^2}}(x_1 \ldots x_d)$. This scaling is necessary to ensure that documents of varying length are treated in a uniform way. After this scaling, traditional numeric measures, such as the Euclidean distance, work more effectively. *LSA* is discussed in Sect. 2.4.3.3 of this chapter. Note that *LSA* is rarely used in conjunction with this kind of scaling. Rather, traditional text mining algorithms are directly applied to the reduced representation obtained from *LSA*.

2.2.2.4 Time Series to Discrete Sequence Data

Time-series data can be converted to discrete sequence data using an approach known as *symbolic aggregate approximation (SAX)*. This method comprises two steps:

1. *Window-based averaging:* The series is divided into windows of length w, and the average time-series value over each window is computed.

2. *Value-based discretization:* The (already averaged) time-series values are discretized into a smaller number of approximately *equi-depth* intervals. This is identical to the equi-depth discretization of numeric attributes that was discussed earlier. The idea is to ensure that each symbol has an approximately equal frequency in the time series. The interval boundaries are constructed by assuming that the time-series values are distributed with a Gaussian assumption. The mean and standard deviation of the (windowed) time-series values are estimated in the data-driven manner to instantiate the parameters of the Gaussian distribution. The quantiles of the Gaussian distribution are used to determine the boundaries of the intervals. This is more efficient than sorting all the data values to determine quantiles, and it may be a more practical approach for a long (or streaming) time series. The values are discretized into a small number (typically 3 to 10) of intervals for the best results. Each such equi-depth interval is mapped to a symbolic value. This creates a symbolic representation of the time series, which is essentially a discrete sequence.

Thus, *SAX* might be viewed as an equi-depth discretization approach after window-based averaging.

2.2.2.5 Time Series to Numeric Data

This particular transformation is very useful because it enables the use of multidimensional algorithms for time-series data. A common method used for this conversion is the discrete wavelet transform (*DWT*). The wavelet transform converts the time series data to multidimensional data, as a set of coefficients that represent averaged differences between different portions of the series. If desired, a subset of the largest coefficients may be used to reduce the data size. This approach will be discussed in Sect. 2.4.4.1 on data reduction. An alternative method, known as the discrete Fourier transform (*DFT*), is discussed in Sect. 14.2.4.2 of Chap. 14. The common property of these transforms is that the various coefficients are no longer as dependency oriented as the original time-series values.

2.2.2.6 Discrete Sequence to Numeric Data

This transformation can be performed in two steps. The first step is to convert the discrete sequence to a *set* of (binary) time series, where the number of time series in this set is equal to the number of distinct symbols. The second step is to map each of these time series into a multidimensional vector using the wavelet transform. Finally, the features from the different series are combined to create a single multidimensional record.

To convert a sequence to a binary time series, one can create a binary string in which the value denotes whether or not a particular symbol is present at a position. For example, consider the following nucleotide sequence, which is drawn on four symbols:

```
ACACACTGTGACTG
```

This series can be converted into the following set of four binary time series corresponding to the symbols A, C, T, and G, respectively:

```
10101000001000
01010100000100
00000010100010
00000001010001
```

A wavelet transformation can be applied to each of these series to create a multidimensional set of features. The features from the four different series can be appended to create a single numeric multidimensional record.

2.2.2.7 Spatial to Numeric Data

Spatial data can be converted to numeric data by using the same approach that was used for time-series data. The main difference is that there are now two contextual attributes (instead of one). This requires modification of the wavelet transformation method. Section 2.4.4.1 will briefly discuss how the one-dimensional wavelet approach can be generalized when there are two contextual attributes. The approach is fairly general and can be used for any number of contextual attributes.

2.2.2.8 Graphs to Numeric Data

Graphs can be converted to numeric data with the use of methods such as multidimensional scaling (*MDS*) and spectral transformations. This approach works for those applications where the edges are weighted, and represent similarity or distance relationships between nodes. The general approach of *MDS* can achieve this goal, and it is discussed in Sect. 2.4.4.2. A spectral approach can also be used to convert a graph into a multidimensional representation. This is also a dimensionality reduction scheme that converts the structural information into a multidimensional representation. This approach will be discussed in Sect. 2.4.4.3.

2.2.2.9 Any Type to Graphs for Similarity-Based Applications

Many applications are based on the notion of similarity. For example, the clustering problem is defined as the creation of groups of similar objects, whereas the outlier detection problem is defined as one in which a subset of objects differing significantly from the remaining objects are identified. Many forms of classification models, such as nearest neighbor classifiers, are also dependent on the notion of similarity. The notion of pairwise similarity can

be best captured with the use of a *neighborhood graph*. For a given set of data objects $\mathcal{O} = \{O_1 \ldots O_n\}$, a neighborhood graph is defined as follows:

1. A single node is defined for each object in $\mathcal{O}$. This is defined by the node set N, containing n nodes where the node i corresponds to the object O_i.

2. An edge exists between O_i and O_j, if the distance $d(O_i, O_j)$ is less than a particular threshold ϵ. Alternatively, the k-nearest neighbors of each node may be used. Because the k-nearest neighbor relationship is not symmetric, this results in a directed graph. The directions on the edges are ignored, and the parallel edges are removed. The weight w_{ij} of the edge (i, j) is equal to a kernelized function of the distance between the objects O_i and O_j, so that larger weights indicate greater similarity. An example is the *heat kernel*:
$$w_{ij} = e^{-d(O_i,O_j)^2/t^2} \tag{2.1}$$
Here, t is a user-defined parameter.

A wide variety of data mining algorithms are available for network data. All these methods can also be used on the similarity graph. Note that the similarity graph can be crisply defined for data objects of any type, as long as an appropriate distance function can be defined. This is the reason that distance function design is so important for virtually any data type. The issue of distance function design will be addressed in Chap. 3. Note that this approach is useful only for applications that are based on the notion of similarity or distances. Nevertheless, many data mining problems are directed or indirectly related to notions of similarity and distances.

2.3 Data Cleaning

The data cleaning process is important because of the errors associated with the data collection process. Several sources of missing entries and errors may arise during the data collection process. Some examples are as follows:

1. Some data collection technologies, such as sensors, are inherently inaccurate because of the hardware limitations associated with collection and transmission. Sometimes sensors may drop readings because of hardware failure or battery exhaustion.

2. Data collected using scanning technologies may have errors associated with it because optical character recognition techniques are far from perfect. Furthermore, speech-to-text data is also prone to errors.

3. Users may not want to specify their information for privacy reasons, or they may specify incorrect values intentionally. For example, it has often been observed that users sometimes specify their birthday incorrectly on automated registration sites such as those of social networks. In some cases, users may choose to leave several fields empty.

4. A significant amount of data is created manually. Manual errors are common during data entry.

5. The entity in charge of data collection may not collect certain fields for some records, if it is too costly. Therefore, records may be incompletely specified.

The aforementioned issues may be a significant source of inaccuracy for data mining applications. Methods are needed to remove or correct missing and erroneous entries from the data. There are several important aspects of data cleaning:

1. *Handling missing entries:* Many entries in the data may remain unspecified because of weaknesses in data collection or the inherent nature of the data. Such missing entries may need to be estimated. The process of estimating missing entries is also referred to as *imputation*.

2. *Handling incorrect entries:* In cases where the same information is available from multiple sources, *inconsistencies* may be detected. Such inconsistencies can be removed as a part of the analytical process. Another method for detecting the incorrect entries is to use domain-specific knowledge about what is already known about the data. For example, if a person's height is listed as 6 m, it is most likely incorrect. More generally, data points that are inconsistent with the remaining data distribution are often noisy. Such data points are referred to as *outliers*. It is, however, dangerous to assume that such data points are always caused by errors. For example, a record representing credit card fraud is likely to be inconsistent with respect to the patterns in most of the (normal) data but should not be removed as "incorrect" data.

3. *Scaling and normalization:* The data may often be expressed in very different scales (e.g., age and salary). This may result in some features being inadvertently weighted too much so that the other features are implicitly ignored. Therefore, it is important to normalize the different features.

The following sections will discuss each of these aspects of data cleaning.

2.3.1 Handling Missing Entries

Missing entries are common in databases where the data collection methods are imperfect. For example, user surveys are often unable to collect responses to all questions. In cases where data contribution is voluntary, the data is almost always incompletely specified. Three classes of techniques are used to handle missing entries:

1. Any data record containing a missing entry may be eliminated entirely. However, this approach may not be practical when most of the records contain missing entries.

2. The missing values may be estimated or imputed. However, errors created by the imputation process may affect the results of the data mining algorithm.

3. The analytical phase is designed in such a way that it can work with missing values. Many data mining methods are inherently designed to work robustly with missing values. This approach is usually the most desirable because it avoids the additional biases inherent in the imputation process.

The problem of estimating missing entries is directly related to the classification problem. In the classification problem, a single attribute is treated specially, and the other features are used to estimate its value. In this case, the missing value can occur on any feature, and therefore the problem is more challenging, although it is fundamentally not different. Many of the methods discussed in Chaps. 10 and 11 for classification can also be used for missing value estimation. In addition, the matrix completion methods discussed in Sect. 18.5 of Chap. 18 may also be used.

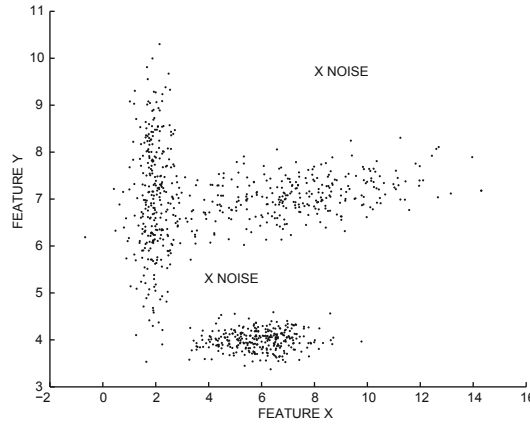


Figure 2.1: Finding noise by data-centric methods

In the case of dependency-oriented data, such as time series or spatial data, missing value estimation is much simpler. In this case, the behavioral attribute values of contextually nearby records are used for the imputation process. For example, in a time-series data set, the average of the values at the time stamp just before or after the missing attribute may be used for estimation. Alternatively, the behavioral values at the last n time-series data stamps can be linearly interpolated to determine the missing value. For the case of spatial data, the estimation process is quite similar, where the average of values at neighboring spatial locations may be used.

2.3.2 Handling Incorrect and Inconsistent Entries

The key methods that are used for removing or correcting the incorrect and inconsistent entries are as follows:

1. *Inconsistency detection:* This is typically done when the data is available from different sources in different formats. For example, a person's name may be spelled out in full in one source, whereas the other source may only contain the initials and a last name. In such cases, the key issues are duplicate detection and inconsistency detection. These topics are studied under the general umbrella of data integration within the database field.

2. *Domain knowledge:* A significant amount of domain knowledge is often available in terms of the ranges of the attributes or rules that specify the relationships across different attributes. For example, if the country field is "United States," then the city field cannot be "Shanghai." Many *data scrubbing* and *data auditing* tools have been developed that use such domain knowledge and constraints to detect incorrect entries.

3. *Data-centric methods:* In these cases, the *statistical behavior* of the data is used to detect outliers. For example, the two isolated data points in Fig. 2.1 marked as "noise" are outliers. These isolated points might have arisen because of errors in the data collection process. However, this may not always be the case because the anomalies may be the result of interesting behavior of the underlying system. Therefore, any detected outlier may need to be manually examined before it is discarded. The use of

data-centric methods for cleaning can sometimes be dangerous because they can result in the removal of useful knowledge from the underlying system. The outlier detection problem is an important analytical technique in its own right, and is discussed in detail in Chaps. 8 and 9.

The methods for addressing erroneous and inconsistent entries are generally highly domain specific.

2.3.3 Scaling and Normalization

In many scenarios, the different features represent different scales of reference and may therefore not be comparable to one another. For example, an attribute such as age is drawn on a very different scale than an attribute such as salary. The latter attribute is typically orders of magnitude larger than the former. As a result, any aggregate function computed on the different features (e.g., Euclidean distances) will be dominated by the attribute of larger magnitude.

To address this problem, it is common to use *standardization*. Consider the case where the jth attribute has mean μ_j and standard deviation σ_j. Then, the jth attribute value x_i^j of the ith record $\overline{X_i}$ may be normalized as follows:

$$z_i^j = \frac{x_i^j - \mu_j}{\sigma_j} \tag{2.2}$$

The vast majority of the normalized values will typically lie in the range $[-3, 3]$ under the normal distribution assumption.

A second approach uses *min-max scaling* to map all attributes to the range $[0, 1]$. Let min_j and max_j represent the minimum and maximum values of attribute j. Then, the jth attribute value x_i^j of the ith record $\overline{X_i}$ may be scaled as follows:

$$y_i^j = \frac{x_i^j - min_j}{max_j - min_j} \tag{2.3}$$

This approach is not effective when the maximum and minimum values are extreme value outliers because of some mistake in data collection. For example, consider the age attribute where a mistake in data collection caused an additional zero to be appended to an age, resulting in an age value of 800 years instead of 80. In this case, most of the scaled data along the age attribute will be in the range $[0, 0.1]$, as a result of which this attribute may be de-emphasized. Standardization is more robust to such scenarios.

2.4 Data Reduction and Transformation

The goal of data reduction is to represent it more compactly. When the data size is smaller, it is much easier to apply sophisticated and computationally expensive algorithms. The reduction of the data may be in terms of the number of rows (records) or in terms of the number of columns (dimensions). Data reduction does result in some loss of information. The use of a more sophisticated algorithm may sometimes compensate for the loss in information resulting from data reduction. Different types of data reduction are used in various applications:

1. *Data sampling:* The records from the underlying data are sampled to create a much smaller database. Sampling is generally much harder in the streaming scenario where the sample needs to be dynamically maintained.

2. *Feature selection:* Only a subset of features from the underlying data is used in the analytical process. Typically, these subsets are chosen in an application-specific way. For example, a feature selection method that works well for clustering may not work well for classification and vice versa. Therefore, this section will discuss the issue of feature subsetting only in a limited way and defer a more detailed discussion to later chapters.

3. *Data reduction with axis rotation:* The correlations in the data are leveraged to represent it in a smaller number of dimensions. Examples of such data reduction methods include principal component analysis (PCA), singular value decomposition (SVD), or latent semantic analysis (LSA) for the text domain.

4. *Data reduction with type transformation:* This form of data reduction is closely related to data type portability. For example, time series are converted to multidimensional data of a smaller size and lower complexity by discrete wavelet transformations. Similarly, graphs can be converted to multidimensional representations by using embedding techniques.

Each of the aforementioned aspects will be discussed in different segments of this section.

2.4.1 Sampling

The main advantage of sampling is that it is simple, intuitive, and relatively easy to implement. The type of sampling used may vary with the application at hand.

2.4.1.1 Sampling for Static Data

It is much simpler to sample data when the entire data is already available, and therefore the number of base data points is known in advance. In the unbiased sampling approach, a predefined fraction f of the data points is selected and retained for analysis. This is extremely simple to implement, and can be achieved in two different ways, depending upon whether or not replacement is used.

In sampling without replacement from a data set $\mathcal{D}$ with n records, a total of $\lceil n \cdot f \rceil$ records are randomly picked from the data. Thus, no duplicates are included in the sample, unless the original data set $\mathcal{D}$ also contains duplicates. In sampling with replacement from a data set $\mathcal{D}$ with n records, the records are sampled *sequentially and independently* from the *entire* data set $\mathcal{D}$ for a total of $\lceil n \cdot f \rceil$ times. Thus, duplicates are possible because the same record may be included in the sample over sequential selections. Generally, most applications do not use replacement because unnecessary duplicates can be a nuisance for some data mining applications, such as outlier detection. Some other specialized forms of sampling are as follows:

1. *Biased sampling:* In biased sampling, some parts of the data are intentionally emphasized because of their greater importance to the analysis. A classical example is that of temporal-decay bias where more recent records have a larger chance of being included in the sample, and stale records have a lower chance of being included. In exponential-decay bias, the probability $p(\overline{X})$ of sampling a data record $\overline{X}$, which was generated

δt time units ago, is proportional to an exponential decay function value regulated by the decay parameter λ:

$$p(\overline{X}) \propto e^{-\lambda \cdot \delta t} \tag{2.4}$$

Here e is the base of the natural logarithm. By using different values of λ, the impact of temporal decay can be regulated appropriately.

2. *Stratified sampling:* In some data sets, important parts of the data may not be sufficiently represented by sampling because of their rarity. A stratified sample, therefore, first partitions the data into a set of desired strata, and then independently samples from each of these strata based on predefined proportions in an application-specific way.

   For example, consider a survey that measures the economic diversity of the lifestyles of different individuals in the population. Even a sample of 1 million participants may not capture a billionaire because of their relative rarity. However, a stratified sample (by income) will independently sample a predefined fraction of participants from each income group to ensure greater robustness in analysis.

Numerous other forms of biased sampling are possible. For example, in density-biased sampling, points in higher-density regions are weighted less to ensure greater representativeness of the rare regions in the sample.

2.4.1.2 Reservoir Sampling for Data Streams

A particularly interesting form of sampling is that of *reservoir sampling* for data streams. In reservoir sampling, a sample of k points is *dynamically* maintained from a data stream. Recall that a stream is of an extremely large volume, and therefore one cannot store it on a disk to sample it. Therefore, for each incoming data point in the stream, one must use a set of efficiently implementable operations to *maintain* the sample.

In the static case, the probability of including a data point in the sample is k/n where k is the sample size, and n is the number of points in the "data set." In this case, the "data set" is not static and cannot be stored on disk. Furthermore, the value of n is constantly increasing as more points arrive and previous data points (outside the sample) have already been discarded. Thus, the sampling approach works with *incomplete knowledge* about the previous history of the stream at any given moment in time. In other words, for each incoming data point in the stream, we need to *dynamically* make two simple *admission control* decisions:

1. What sampling rule should be used to decide whether to include the newly incoming data point in the sample?
2. What rule should be used to decide how to eject a data point from the sample to "make room" for the newly inserted data point?

Fortunately, it is relatively simple to design an algorithm for reservoir sampling in data streams [498]. For a reservoir of size k, the first k data points in the stream are used to initialize the reservoir. Subsequently, for the nth incoming stream data point, the following two admission control decisions are applied:

1. Insert the nth incoming stream data point into the reservoir with probability k/n.
2. If the newly incoming data point was inserted, then eject one of the old k data points at random to make room for the newly arriving point.

It can be shown that the aforementioned rule maintains an unbiased reservoir sample from the data stream.

Lemma 2.4.1 *After n stream points have arrived, the probability of any stream point being included in the reservoir is the same, and is equal to k/n.*

Proof: This result is easy to show by induction. At initialization of the first k data points, the theorem is trivially true. Let us (inductively) assume that it is also true after $(n-1)$ data points have been received, and therefore the probability of each point being included in the reservoir is $k/(n-1)$. The probability of the arriving point being included in the stream is k/n, and therefore the lemma holds true for the arriving data point. It remains to prove the result for the remaining points in the data stream. There are two *disjoint* case events that can arise for an incoming data point, and the final probability of a point being included in the reservoir is the sum of these two cases:

I: The incoming data point is not inserted into the reservoir. The probability of this is $(n-k)/n$. Because the original probability of any point being included in the reservoir by the inductive assumption, is $k/(n-1)$, the overall probability of a point being included in the reservoir *and* Case I event, is the multiplicative value of $p_1 = \frac{k(n-k)}{n(n-1)}$.

II: The incoming data point is inserted into the reservoir. The probability of Case II is equal to insertion probability k/n of incoming data points. Subsequently, existing reservoir points are retained with probability $(k-1)/k$ because exactly one of them is ejected. Because the inductive assumption implies that any of the earlier points in the data stream was originally present in the reservoir with probability $k/(n-1)$, it implies that the probability of a point being included in the reservoir *and* Case II event is given by the product p_2 of the three aforementioned probabilities:

$$p_2 = \left(\frac{k}{n}\right)\left(\frac{k-1}{k}\right)\left(\frac{k}{n-1}\right) = \frac{k(k-1)}{n(n-1)} \tag{2.5}$$

Therefore, the total probability of a stream point being retained in the reservoir after the nth data point arrival is given by the sum of p_1 and p_2. It can be shown that this is equal to k/n. ■

It is possible to extend reservoir sampling to cases where temporal bias is present in the data stream. In particular, the case of exponential bias has been addressed in [35].

2.4.2 Feature Subset Selection

A second method for data preprocessing is feature subset selection. Some features can be discarded when they are known to be irrelevant. Which features are relevant? Clearly, this decision depends on the application at hand. There are two primary types of feature selection:

1. *Unsupervised feature selection:* This corresponds to the removal of noisy and redundant attributes from the data. Unsupervised feature selection is best defined in terms of its impact on clustering applications, though the applicability is much broader. It is difficult to comprehensively describe such feature selection methods without using the clustering problem as a proper context. Therefore, a discussion of methods for unsupervised feature selection is deferred to Chap. 6 on data clustering.

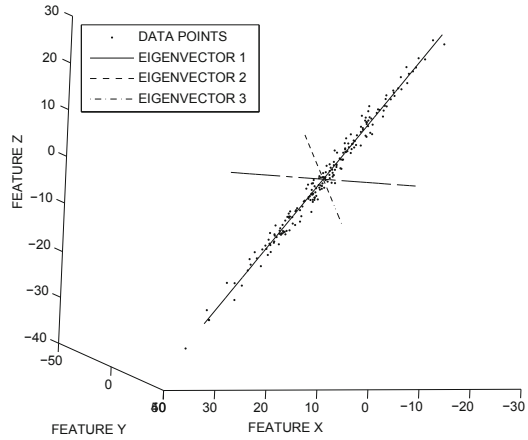


Figure 2.2: Highly correlated data represented in a small number of dimensions in an axis system that is rotated appropriately

2. *Supervised feature selection:* This type of feature selection is relevant to the problem of data classification. In this case, only the features that can predict the class attribute effectively are the most relevant. Such feature selection methods are often closely integrated with analytical methods for classification. A detailed discussion is deferred to Chap. 10 on data classification.

Feature selection is an important part of the data mining process because it defines the quality of the input data.

2.4.3 Dimensionality Reduction with Axis Rotation

In real data sets, a significant number of correlations exist among different attributes. In some cases, hard constraints or rules between attributes may uniquely define some attributes in terms of others. For example, the date of birth of an individual (represented quantitatively) is perfectly correlated with his or her age. In most cases, the correlations may not be quite as perfect, but significant dependencies may still exist among the different features. Unfortunately, real data sets contain many such redundancies that escape the attention of the analyst during the initial phase of data creation. These correlations and constraints correspond to implicit redundancies because they imply that knowledge of some subsets of the dimensions can be used to predict the values of the other dimensions. For example, consider the 3-dimensional data set illustrated in Fig. 2.2. In this case, if the axis is rotated to the orientation illustrated in the figure, the correlations and redundancies in the newly transformed feature values are removed. As a result of this redundancy removal, the entire data can be (approximately) represented along a 1-dimensional line. Thus, the *intrinsic dimensionality* of this 3-dimensional data set is 1. The other two axes correspond to the low-variance dimensions. If the data is represented as coordinates in the new axis system illustrated in Fig. 2.2, then the coordinate values along these low-variance dimensions will not vary much. Therefore, after the axis system has been rotated, these dimensions can be removed without much information loss.

A natural question arises as to how the correlation-removing axis system such as that in Fig. 2.2 may be determined in an automated way. Two natural methods to achieve this goal

are those of *principal component analysis (PCA)* and *singular value decomposition (SVD)*. These two methods, while not exactly identical at the definition level, are closely related. Although the notion of principal component analysis is intuitively easier to understand, *SVD* is a more general framework and can be used to perform *PCA* as a special case.

2.4.3.1 Principal Component Analysis

PCA is generally applied after subtracting the mean of the data set from each data point. However, it is also possible to use it without mean centering, as long as the mean of the data is separately stored. This operation is referred to as *mean centering*, and it results in a data set centered at the origin. The goal of *PCA* is to rotate the data into an axis-system where the greatest amount of variance is captured in a small number of dimensions. It is intuitively evident from the example of Fig. 2.2 that such an axis system is affected by the correlations between attributes. An important observation, which we will show below, is that the variance of a data set along a particular direction can be expressed directly in terms of its covariance matrix.

Let C be the $d \times d$ symmetric covariance matrix of the $n \times d$ data matrix D. Thus, the (i, j)th entry c_{ij} of C denotes the covariance between the ith and jth columns (dimensions) of the data matrix D. Let μ_i represent the mean along the ith dimension. Specifically, if x_k^m be the mth dimension of the kth record, then the value of the covariance entry c_{ij} is as follows:

$$c_{ij} = \frac{\sum_{k=1}^{n} x_k^i x_k^j}{n} - \mu_i \mu_j \quad \forall i, j \in \{1 \ldots d\} \tag{2.6}$$

Let $\overline{\mu} = (\mu_1 \ldots \mu_d)$ is the d-dimensional row vector representing the means along the different dimensions. Then, the aforementioned $d \times d$ computations of Eq. 2.6 for different values of i and j can be expressed compactly in $d \times d$ matrix form as follows:

$$C = \frac{D^T D}{n} - \overline{\mu}^T \overline{\mu} \tag{2.7}$$

Note that the d diagonal entries of the matrix C correspond to the d variances. The covariance matrix C is positive semi-definite, because it can be shown that for any d-dimensional column vector $\overline{v}$, the value of $\overline{v}^T C \overline{v}$ is equal to the variance of the 1-dimensional projection $D\overline{v}$ of the data set D on $\overline{v}$.

$$\overline{v}^T C \overline{v} = \frac{(D\overline{v})^T D\overline{v}}{n} - (\overline{\mu}\,\overline{v})^2 = \text{Variance of 1-dimensional points in } D\overline{v} \geq 0 \tag{2.8}$$

In fact, the goal of *PCA* is to successively determine orthonormal vectors $\overline{v}$ maximizing $\overline{v}^T C \overline{v}$. How can one determine such directions? Because the covariance matrix is symmetric and positive semidefinite, it can be diagonalized as follows:

$$C = P \Lambda P^T \tag{2.9}$$

The columns of the matrix P contain the orthonormal eigenvectors of C, and Λ is a diagonal matrix containing the nonnegative eigenvalues. The entry Λ_{ii} is the eigenvalue corresponding to the ith eigenvector (or column) of the matrix P. These eigenvectors represent successive orthogonal solutions[1] to the aforementioned optimization model maximizing the variance $\overline{v}^T C \overline{v}$ along the unit direction $\overline{v}$.

[1]Setting the gradient of the Lagrangian relaxation $\overline{v}^T C \overline{v} - \lambda(||\overline{v}||^2 - 1)$ to 0 is equivalent to the eigenvector condition $C\overline{v} - \lambda\overline{v} = 0$. The variance along an eigenvector is $\overline{v}^T C \overline{v} = \overline{v}^T \lambda \overline{v} = \lambda$. Therefore, one should include the orthonormal eigenvectors in decreasing order of eigenvalue λ to maximize preserved variance in reduced subspace.

An interesting property of this diagonalization is that both the eigenvectors and eigenvalues have a geometric interpretation in terms of the underlying data distribution. Specifically, if the axis system of data representation is rotated to the orthonormal set of eigenvectors in the columns of P, then it can be shown that all $\binom{d}{2}$ covariances of the newly transformed feature values are zero. In other words, *the greatest variance-preserving directions are also the correlation-removing directions*. Furthermore, the eigenvalues represent the variances of the data along the corresponding eigenvectors. In fact, the diagonal matrix Λ is the new covariance matrix after axis rotation. Therefore, eigenvectors with large eigenvalues preserve greater variance, and are also referred to as *principal components*. Because of the nature of the optimization formulation used to derive this transformation, a new axis system containing only the eigenvectors with the largest eigenvalues is optimized to *retaining the maximum variance in a fixed number of dimensions*. For example, the scatter plot of Fig. 2.2 illustrates the various eigenvectors, and it is evident that the eigenvector with the largest variance is all that is needed to create a variance-preserving representation. It generally suffices to retain only a small number of eigenvectors with large eigenvalues.

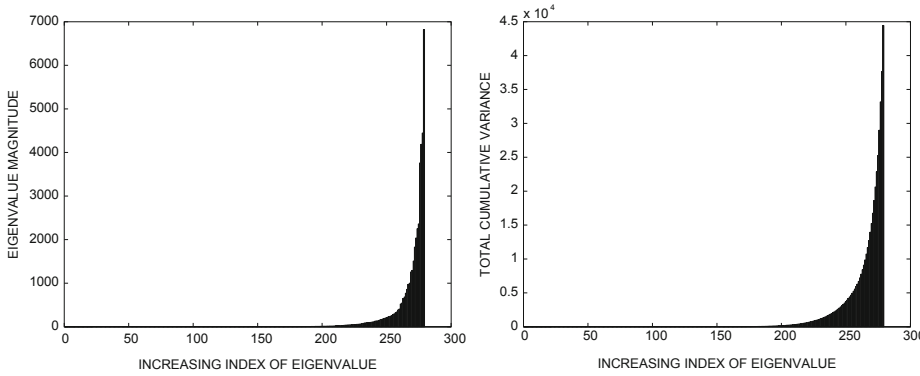
Without loss of generality, it can be assumed that the columns of P (and corresponding diagonal matrix Λ) are arranged from left to right in such a way that they correspond to decreasing eigenvalues. Then, the transformed data matrix D' in the new coordinate system after axis rotation to the orthonormal columns of P can be algebraically computed as the following linear transformation:

$$D' = DP \tag{2.10}$$

While the transformed data matrix D' is also of size $n \times d$, only its first (leftmost) $k \ll d$ columns will show significant variation in values. Each of the remaining $(d - k)$ columns of D' will be approximately equal to the mean of the data in the rotated axis system. For mean-centered data, the values of these $(d - k)$ columns will be almost 0. Therefore, the dimensionality of the data can be reduced, and only the first k columns of the transformed data matrix D' may need to be retained[2] for representation purposes. Furthermore, it can be confirmed that the covariance matrix of the transformed data $D' = DP$ is the diagonal matrix Λ by applying the covariance definition of Eq. 2.7 to DP (transformed data) and $\overline{\mu}P$ (transformed mean) instead of D and $\overline{\mu}$, respectively. The resulting covariance matrix can be expressed in terms of the original covariance matrix C as $P^T C P$. Substituting $C = P \Lambda P^T$ from Eq. 2.9 shows equivalence because $P^T P = P P^T = I$. In other words, correlations have been removed from the transformed data because Λ is diagonal.

The variance of the data set defined by projections along top-k eigenvectors is equal to the sum of the k corresponding eigenvalues. In many applications, the eigenvalues show a precipitous drop-off after the first few values. For example, the behavior of the eigenvalues for the 279-dimensional *Arrythmia* data set from the *UCI Machine Learning Repository* [213] is illustrated in Fig. 2.3. Figure 2.3a shows the absolute magnitude of the eigenvalues in increasing order, whereas Fig. 2.3b shows the total amount of variance retained in the top-k eigenvalues. Figure 2.3b can be derived by using the cumulative sum of the smallest eigenvalues in Fig. 2.3a. It is interesting to note that the 215 smallest eigenvalues contain less than 1 % of the total variance in the data and can therefore be removed with little change to the results of similarity-based applications. Note that the *Arrythmia* data set is not a very strongly correlated data set along many pairs of dimensions. Yet, the dimensionality reduction is drastic because of the *cumulative* effect of the correlations across many dimensions.

[2]The means of the remaining columns also need be stored if the data set is not mean centered.



(a) Magnitude of Eigenvalues (Increasing Index): *Arrythmia*

(b) Variance in smallest k Eigenvalues: *Arrythmia*

Figure 2.3: Variance retained with increasing number of eigenvalues for the *Arrythmia* data set

The eigenvectors of the matrix C may be determined by using any numerical method discussed in [295] or by an off-the-shelf eigenvector solver. *PCA* can be extended to discovering nonlinear embeddings with the use of a method known as the *kernel trick*. Refer to Sect. 10.6.4.1 of Chap. 10 for a brief description of kernel *PCA*.

2.4.3.2 Singular Value Decomposition

Singular value decomposition (*SVD*) is closely related to principal component analysis (*PCA*). However, these distinct methods are sometimes confused with one another because of the close relationship. Before beginning the discussion of *SVD*, we state how it is related to *PCA*. *SVD* is more general than *PCA* because it provides *two* sets of basis vectors instead of one. *SVD* provides basis vectors of both the rows and columns of the data matrix, whereas *PCA* only provides basis vectors of the rows of the data matrix. Furthermore, *SVD* provides the same basis as *PCA* for the rows of the data matrix in certain special cases:

SVD provides the same basis vectors and data transformation as PCA for data sets in which the mean of each attribute is 0.

The basis vectors of *PCA* are invariant to mean-translation, whereas those of *SVD* are not. When the data are not mean centered, the basis vectors of *SVD* and *PCA* will not be the same, and qualitatively different results may be obtained. *SVD* is often applied without mean centering to sparse nonnegative data such as document-term matrices. A formal way of defining *SVD* is as a decomposable product of (or *factorization* into) three matrices:

$$D = Q\Sigma P^T \tag{2.11}$$

Here, Q is an $n \times n$ matrix with orthonormal columns, which are the *left singular vectors*. Σ is an $n \times d$ diagonal matrix containing the *singular values*, which are always nonnegative and, by convention, arranged in nonincreasing order. Furthermore, P is a $d \times d$ matrix with orthonormal columns, which are the *right singular vectors*. Note that the diagonal matrix Σ is rectangular rather than square, but it is referred to as diagonal because only entries of the

form Σ_{ii} are nonzero. It is a fundamental fact of linear algebra that such a decomposition always exists, and a proof may be found in [480]. The number of nonzero diagonal entries of Σ is equal to the rank of the matrix D, which is at most $\min\{n, d\}$. Furthermore, because of the orthonormality of the singular vectors, both $P^T P$ and $Q^T Q$ are identity matrices. We make the following observations:

1. The columns of matrix Q, which are also the left singular vectors, are the orthonormal eigenvectors of DD^T. This is because $DD^T = Q\Sigma(P^T P)\Sigma^T Q^T = Q\Sigma\Sigma^T Q^T$. Therefore, the square of the nonzero singular values, which are diagonal entries of the $n \times n$ diagonal matrix $\Sigma\Sigma^T$, represent the nonzero eigenvalues of DD^T.

2. The columns of matrix P, which are also the right singular vectors, are the orthonormal eigenvectors of $D^T D$. The square of the nonzero singular values, which are represented in diagonal entries of the $d \times d$ diagonal matrix $\Sigma^T\Sigma$, are the nonzero eigenvalues of $D^T D$. Note that the nonzero eigenvalues of DD^T and $D^T D$ are the same. The matrix P is particularly important because it provides the basis vectors, which are analogous to the eigenvectors of the covariance matrix in *PCA*.

3. Because the covariance matrix of mean-centered data is $\frac{D^T D}{n}$ (cf. Eq. 2.7) and the right singular vectors of *SVD* are eigenvectors of $D^T D$, it follows that the eigenvectors of *PCA* are the same as the right-singular vectors of *SVD* for mean-centered data. Furthermore, the squared singular values in *SVD* are n times the eigenvalues of *PCA*. This equivalence shows why *SVD* and *PCA* can provide the same transformation for mean-centered data.

4. Without loss of generality, it can be assumed that the diagonal entries of Σ are arranged in decreasing order, and the columns of matrix P and Q are also ordered accordingly. Let P_k and Q_k be the truncated $d \times k$ and $n \times k$ matrices obtained by selecting the first k columns of P and Q, respectively. Let Σ_k be the $k \times k$ square matrix containing the top k singular values. Then, the *SVD* factorization yields an *approximate* d-dimensional data representation of the original data set D:

$$D \approx Q_k \Sigma_k P_k^T \tag{2.12}$$

   The columns of P_k represent a k-dimensional basis system for a reduced representation of the data set. The dimensionality reduced data set in this k-dimensional basis system is given by the $n \times k$ data set $D'_k = DP_k = Q_k\Sigma_k$, as in Eq. 2.10 of *PCA*. Each of the n rows of D'_k contain the k coordinates of each transformed data point in this new axis system. Typically, the value of k is much smaller than both n and d. Furthermore, unlike *PCA*, the rightmost $(d-k)$ columns of the full d-dimensional transformed data matrix $D' = DP$ will be approximately 0 (rather than the data mean), whether the data are mean centered or not. In general, *PCA* projects the data on a low-dimensional hyperplane passing through the data mean, whereas *SVD* projects the data on a low-dimensional hyperplane passing through the origin. *PCA* captures as much of the variance (or, squared Euclidean distance about the *mean*) of the data as possible, whereas *SVD* captures as much of the aggregate squared Euclidean distance about the *origin* as possible. This method of approximating a data matrix is referred to as *truncated SVD*.

In the following, we will show that truncated *SVD* maximizes the aggregate squared Euclidean distances (or *energy*) of the transformed data points about the origin. Let $\overline{v}$ be a

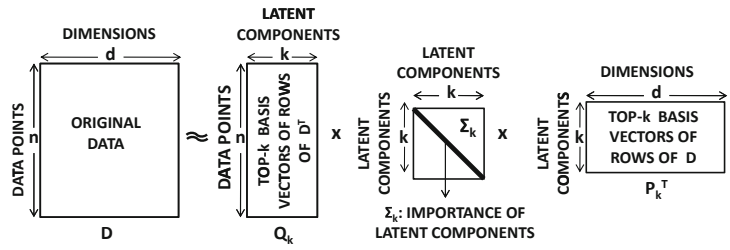


Figure 2.4: Complementary basis properties of matrix factorization in *SVD*

d-dimensional column vector and $D\overline{v}$ be the projection of the data set D on $\overline{v}$. Consider the problem of determining the *unit* vector $\overline{v}$ such that the sum of squared Euclidean distances $(D\overline{v})^T(D\overline{v})$ of the projected data points from the origin is maximized. Setting the gradient of the Lagrangian relaxation $\overline{v}^T D^T D\overline{v} - \lambda(||\overline{v}||^2 - 1)$ to 0 is equivalent to the eigenvector condition $D^T D\overline{v} - \lambda\overline{v} = 0$. Because the right singular vectors are eigenvectors of $D^T D$, it follows that the eigenvectors (right singular vectors) with the k largest eigenvalues (squared singular values) provide a basis that maximizes the preserved energy in the transformed and reduced data matrix $D'_k = DP_k = Q_k\Sigma_k$. Because the *energy*, which is the sum of squared Euclidean distances from the origin, is invariant to axis rotation, the energy in D'_k is the same as that in $D'_k P_k^T = Q_k\Sigma_k P_k^T$. Therefore, *$k$-rank SVD is a maximum energy-preserving factorization*. This result is known as the *Eckart–Young theorem*.

The total preserved energy of the projection $D\overline{v}$ of the data set D along unit right-singular vector $\overline{v}$ with singular value σ is given by $(D\overline{v})^T(D\overline{v})$, which can be simplified as follows:

$$(D\overline{v})^T(D\overline{v}) = \overline{v}^T(D^T D\overline{v}) = \overline{v}^T(\sigma^2\overline{v}) = \sigma^2$$

Because the energy is defined as a linearly separable sum along orthonormal directions, the preserved energy in the data projection along the top-k singular vectors is equal to the sum of the squares of the top-k singular values. Note that the total energy in the data set D is always equal to the sum of the squares of all the nonzero singular values. It can be shown that maximizing the preserved energy is the same as minimizing the squared error[3] (or *lost* energy) of the k-rank approximation. This is because the sum of the energy in the preserved subspace and the lost energy in the complementary (discarded) subspace is always a constant, which is equal to the energy in the original data set D.

When viewed purely in terms of eigenvector analysis, *SVD* provides two different perspectives for understanding the transformed and reduced data. The transformed data matrix can either be viewed as the *projection* DP_k of the data matrix D on the top k basis eigenvectors P_k of the $d \times d$ *scatter matrix* $D^T D$, or it can *directly* be viewed as the scaled eigenvectors $Q_k\Sigma_k = DP_k$ of the $n \times n$ *dot-product similarity matrix* DD^T. While it is generally computationally expensive to extract the eigenvectors of an $n \times n$ similarity matrix, such an approach also generalizes to nonlinear dimensionality reduction methods where notions of linear basis vectors do not exist in the original space. In such cases, the dot-product similarity matrix is replaced with a more complex similarity matrix in order to extract a nonlinear embedding (cf. Table 2.3).

SVD is more general than *PCA* and can be used to simultaneously determine a subset of k basis vectors for the data matrix and its transpose with the maximum energy. The latter can be useful in understanding complementary transformation properties of D^T.

[3]The squared error is the sum of squares of the entries in the error matrix $D - Q_k\Sigma_k P_k^T$.

The orthonormal columns of Q_k provide a k-dimensional basis system for (approximately) transforming "data points" corresponding to the rows of D^T, and the matrix $D^T Q_k = P_k \Sigma_k$ contains the corresponding coordinates. For example, in a user-item ratings matrix, one may wish to determine either a reduced representation of the users, or a reduced representation of the items. *SVD* provides the basis vectors for both reductions. Truncated *SVD* expresses the data in terms of k dominant *latent components*. The ith latent component is expressed in the ith basis vectors of both D and D^T, and its relative importance in the data is defined by the ith singular value. By decomposing the matrix product $Q_k \Sigma_k P_k^T$ into column vectors of Q_k and P_k (i.e., dominant basis vectors of D^T and D), the following additive sum of the k latent components can be obtained:

$$Q_k \Sigma_k P_k^T = \sum_{i=1}^{k} \overline{q_i} \sigma_i \overline{p_i}^T = \sum_{i=1}^{k} \sigma_i (\overline{q_i}\ \overline{p_i}^T) \tag{2.13}$$

Here $\overline{q_i}$ is the ith column of Q, $\overline{p_i}$ is the ith column of P, and σ_i is the ith diagonal entry of Σ. Each latent component $\sigma_i (\overline{q_i}\ \overline{p_i}^T)$ is an $n \times d$ matrix with rank 1 and energy σ_i^2. This decomposition is referred to as *spectral decomposition*. The relationships of the reduced basis vectors to *SVD* matrix factorization are illustrated in Fig. 2.4.

An example of a rank-2 truncated *SVD* of a toy 6×6 matrix is illustrated below:

$$D = \begin{pmatrix} 2 & 2 & 1 & 2 & 0 & 0 \\ 2 & 3 & 3 & 3 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 2 & 2 & 2 & 3 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 2 & 1 & 2 \end{pmatrix} \approx Q_2 \Sigma_2 P_2^T$$

$$\approx \begin{pmatrix} -0.41 & 0.17 \\ -0.65 & 0.31 \\ -0.23 & 0.13 \\ -0.56 & -0.20 \\ -0.10 & -0.46 \\ -0.19 & -0.78 \end{pmatrix} \begin{pmatrix} 8.4 & 0 \\ 0 & 3.3 \end{pmatrix} \begin{pmatrix} -0.41 & -0.49 & -0.44 & -0.61 & -0.10 & -0.12 \\ 0.21 & 0.31 & 0.26 & -0.37 & -0.44 & -0.68 \end{pmatrix}$$

$$= \begin{pmatrix} 1.55 & 1.87 & \underline{1.67} & 1.91 & 0.10 & 0.04 \\ 2.46 & 2.98 & 2.66 & 2.95 & 0.10 & -0.03 \\ 0.89 & 1.08 & 0.96 & 1.04 & 0.01 & -0.04 \\ 1.81 & 2.11 & 1.91 & 3.14 & 0.77 & 1.03 \\ 0.02 & -0.05 & -0.02 & 1.06 & 0.74 & 1.11 \\ 0.10 & -0.02 & 0.04 & 1.89 & 1.28 & 1.92 \end{pmatrix}$$

Note that the rank-2 matrix is a good approximation of the original matrix. The entry with the largest error is underlined in the final approximated matrix. Interestingly, this entry is also *inconsistent* with the structure of the remaining matrix in the *original* data (why?). Truncated *SVD* often tries to correct inconsistent entries, and this property is sometimes leveraged for noise reduction in error-prone data sets.

2.4.3.3 Latent Semantic Analysis

Latent semantic analysis (*LSA*) is an application of the *SVD* method to the text domain. In this case, the data matrix D is an $n \times d$ document-term matrix containing normalized

word frequencies in the n documents, where d is the size of the lexicon. No mean centering is used, but the results are approximately the same as *PCA* because of the sparsity of D. The sparsity of D implies that most of the entries in D are 0, and the mean values of each column are much smaller than the nonzero values. In such scenarios, it can be shown that the covariance matrix is approximately proportional to $D^T D$. The sparsity of the data set also results in a low intrinsic dimensionality. Therefore, in the text domain, the reduction in dimensionality from *LSA* is rather drastic. For example, it is not uncommon to be able to represent a corpus drawn on a lexicon of 100,000 dimensions in fewer than 300 dimensions.

LSA is a classical example of how the "loss" of information from discarding some dimensions can actually result in an *improvement* in the quality of the data representation. The text domain suffers from two main problems corresponding to *synonymy* and *polysemy*. Synonymy refers to the fact that two words may have the same meaning. For example, the words "*comical*" and "*hilarious*" mean approximately the same thing. Polysemy refers to the fact that the same word may mean two different things. For example, the word "*jaguar*" could refer to a car or a cat. Typically, the significance of a word can only be understood in the context of other words in the document. This is a problem for similarity-based applications because the computation of similarity with the use of word frequencies may not be completely accurate. For example, two documents containing the words "*comical*" and "*hilarious,*" respectively, may not be deemed sufficiently similar in the original representation space. The two aforementioned issues are a direct result of synonymy and polysemy effects. The truncated representation after *LSA* typically removes the noise effects of synonymy and polysemy because the (high-energy) singular vectors represent the directions of correlation in the data, and the appropriate context of the word is implicitly represented along these directions. The variations because of individual differences in usage are implicitly encoded in the low-energy directions, which are truncated anyway. It has been observed that significant *qualitative* improvements [184, 416] for text applications may be achieved with the use of *LSA*. The improvement[4] is generally greater in terms of synonymy effects than polysemy. This noise-removing behavior of *SVD* has also been demonstrated in general multidimensional data sets [25].

2.4.3.4 Applications of PCA and SVD

Although *PCA* and *SVD* are primarily used for data reduction and compression, they have many other applications in data mining. Some examples are as follows:

1. *Noise reduction:* While removal of the smaller eigenvectors/singular vectors in *PCA* and *SVD* can lead to information loss, it can also lead to *improvement* in the quality of data representation in surprisingly many cases. The main reason is that the variations along the small eigenvectors are often the result of noise, and their removal is generally beneficial. An example is the application of *LSA* in the text domain where the removal of the smaller components leads to the enhancement of the semantic characteristics of text. *SVD* is also used for deblurring noisy images. These text- and image-specific results have also been shown to be true in arbitrary data domains [25]. Therefore, the data reduction is not just space efficient but actually provides *qualitative* benefits in many cases.

[4]Concepts that are not present predominantly in the collection will be ignored by truncation. Therefore, alternative meanings reflecting infrequent concepts in the collection will be ignored. While this has a robust effect on the *average*, it may not always be the correct or complete disambiguation of polysemous words.

2. *Data imputation: SVD* and *PCA* can be used for data imputation applications [23], such as collaborative filtering, because the *reduced* matrices Q_k, Σ_k, and P_k can be estimated for small values of k even from incomplete data matrices. Therefore, the entire matrix can be approximately reconstructed as $Q_k \Sigma_k P_k^T$. This application is discussed in Sect. 18.5 of Chap. 18.

3. *Linear equations:* Many data mining applications are optimization problems in which the solution is recast into a system of linear equations. For any linear system $A\overline{y} = 0$, any right singular vector of A with 0 singular value will satisfy the system of equations (see Exercise 14). Therefore, any linear combination of the 0 singular vectors will provide a solution.

4. *Matrix inversion: SVD* can be used for the inversion of a square $d \times d$ matrix D. Let the decomposition of D be given by $Q\Sigma P^T$. Then, the inverse of D is $D^{-1} = P\Sigma^{-1}Q^T$. Note that Σ^{-1} can be trivially computed from Σ by inverting its diagonal entries. The approach can also be generalized to the determination of the *Moore–Penrose pseudoinverse* D^+ of a rank-k matrix D by inverting only the nonzero diagonal entries of Σ. The approach can even be generalized to non-square matrices by performing the additional operation of transposing Σ. Such matrix inversion operations are required in many data mining applications such as least-squares regression (cf. Sect. 11.5 of Chap. 11) and social network analysis (cf. Chap. 19).

5. *Matrix algebra:* Many network mining applications require the application of algebraic operations such as the computation of the powers of a matrix. This is common in random-walk methods (cf. Chap. 19), where the kth powers of the symmetric adjacency matrix of an undirected network may need to be computed. Such symmetric adjacency matrices can be decomposed into the form $Q\Delta Q^T$. The kth power of this decomposition can be efficiently computed as $D^k = Q\Delta^k Q^T$. In fact, any polynomial function of the matrix can be computed efficiently.

SVD and *PCA* are extraordinarily useful because matrix and linear algebra operations are ubiquitous in data mining. *SVD* and *PCA* facilitate such matrix operations by providing convenient decompositions and basis representations. *SVD* has rightly been referred to [481] as "absolutely a high point of linear algebra."

2.4.4 Dimensionality Reduction with Type Transformation

In these methods, dimensionality reduction is coupled with type transformation. In most cases, the data is *transformed* from a more complex type to a less complex type, such as multidimensional data. Thus, these methods serve the dual purpose of data reduction and type portability. This section will study two such transformation methods:

1. *Time series to multidimensional:* A number of methods, such as the discrete Fourier transform and discrete wavelet transform are used. While these methods can also be viewed as a rotation of an axis system defined by the various time stamps of the contextual attribute, the data are no longer dependency oriented after the rotation. Therefore, the resulting data set can be processed in a similar way to multidimensional data. We will study the Haar wavelet transform because of its intuitive simplicity.

2. *Weighted graphs to multidimensional:* Multidimensional scaling and spectral methods are used to embed weighted graphs in multidimensional spaces, so that the similarity or distance values on the edges are captured by a multidimensional embedding.

Table 2.2: An example of wavelet coefficient computation

Granularity (order k)	**Averages** (Φ values)	**DWT coefficients** (ψ values)
$k = 4$	(8, 6, 2, 3, 4, 6, 6, 5)	–
$k = 3$	(7, 2.5, 5, 5.5)	(1, −0.5, −1, 0.5)
$k = 2$	(4.75, 5.25)	(2.25, −0.25)
$k = 1$	(5)	(−0.25)

This section will discuss each of these techniques.

2.4.4.1 Haar Wavelet Transform

Wavelets are a well-known technique that can be used for multigranularity decomposition and summarization of time-series data into the multidimensional representation. The *Haar* wavelet is a particularly popular form of wavelet decomposition because of its intuitive nature and ease of implementation. To understand the intuition behind wavelet decomposition, an example of sensor temperatures will be used.

Suppose that a sensor measured the temperatures over the course of 12 h from the morning until the evening. Assume that the sensor samples temperatures at the rate of 1 sample/s. Thus, over the course of a single day, a sensor will collect $12 \times 60 \times 60 = 43,200$ readings. Clearly, this will not scale well over many days and many sensors. An important observation is that many adjacent sensor readings will be very similar, causing this representation to be very wasteful. So, how can we represent this data approximately in a small amount of space? How can we determine the key regions where "variations" in readings occur, and store these variations instead of repeating values?

Suppose we only stored the average over the entire day. This provides some idea of the temperature but not much else about the variation over the day. Now, if the difference in average temperature between the first half and second half of the day is also stored, we can derive the averages for both the first and second half of the day from these two values. This principle can be applied recursively because the first half of the day can be divided into the first quarter of the day and the second quarter of the day. Thus, with four stored values, we can perfectly reconstruct the averages in four quarters of the day. This process can be applied recursively right down to the level of granularity of the sensor readings. These "difference values" are used to derive wavelet coefficients. Of course, we did not yet achieve any data *reduction* because the number of such coefficients can be shown to be exactly equal to the length of the original time series.

It is important to understand that large difference values tell us more about the *variations* in the temperature values than the small ones, and they are therefore more important to store. Therefore, larger coefficient values are stored after a normalization for the level of granularity. This normalization, which is discussed later, has a bias towards storing coefficients representing longer time scales because trends over longer periods of time are more informative for (global) series reconstruction.

More formally, the wavelet technique creates a decomposition of the time series into a set of coefficient-weighted wavelet basis vectors. Each of the coefficients represents the rough variation of the time series between the two halves of a particular time range. The

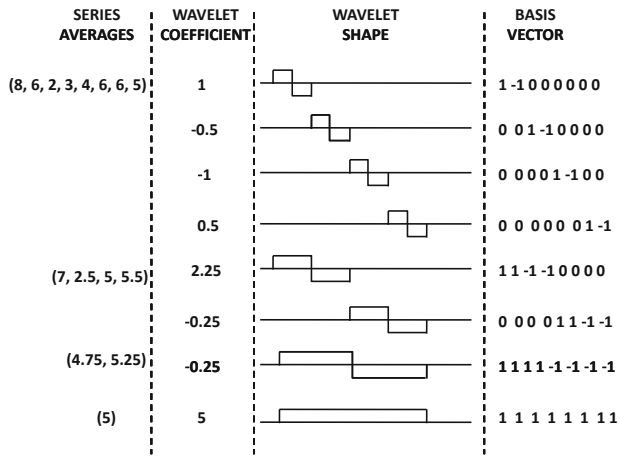


Figure 2.5: Illustration of the wavelet decomposition

wavelet basis vector is a time series that represents the temporal range of this variation in the form of a simple step function. The wavelet coefficients are of different *orders*, depending on the length of the time-series segment analyzed, which also represents the granularity of analysis. The higher-order coefficients represent the broad trends in the series because they correspond to larger ranges. The more localized trends are captured by the lower-order coefficients. Before providing a more notational description, a simple recursive description of wavelet decomposition of a time series segment S is provided below in two steps:

1. Report half the average difference of the behavioral attribute values between the first and second temporal halves of S as a wavelet coefficient.

2. Recursively apply this approach to first and second temporal halves of S.

At the end of the process, a reduction process is performed, where larger (normalized) coefficients are retained. This normalization step will be described in detail later.

A more formal and notation-intensive description will be provided at this point. For ease in discussion, assume that the length q of the series is a power of 2. For each value of $k \geq 1$, the Haar wavelet decomposition defines 2^{k-1} coefficients of order k. Each of these 2^{k-1} coefficients corresponds to a contiguous portion of the time series of length $q/2^{k-1}$. The ith of these 2^{k-1} coefficients corresponds to the segment in the series starting from position $(i-1) \cdot q/2^{k-1} + 1$ to the position $i \cdot q/2^{k-1}$. Let us denote this coefficient by ψ_k^i and the corresponding time-series segment by S_k^i. At the same time, let us define the average value of the first half of the S_k^i by a_k^i and that of the second half by b_k^i. Then, the value of ψ_k^i is given by $(a_k^i - b_k^i)/2$. More formally, if Φ_k^i denote the average value of the S_k^i, then the value of ψ_k^i can be defined recursively as follows:

$$\psi_k^i = (\Phi_{k+1}^{2 \cdot i-1} - \Phi_{k+1}^{2 \cdot i})/2 \tag{2.14}$$

The set of Haar coefficients is defined by all the coefficients of order 1 to $\log_2(q)$. In addition, the global average Φ_1^1 is required for the purpose of perfect reconstruction. The total number of coefficients is exactly equal to the length of the original series, and the dimensionality reduction is obtained by discarding the smaller (normalized) coefficients. This will be discussed later.

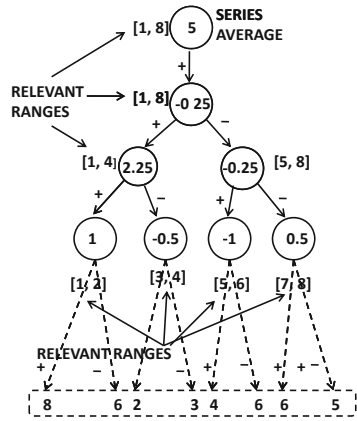


Figure 2.6: The error tree from the wavelet decomposition

The coefficients of different orders provide an understanding of the major trends in the data at a particular level of granularity. For example, the coefficient ψ_k^i is half the quantity by which the first half of the segment S_k^i is larger than the second half of the same segment. Because larger values of k correspond to geometrically reducing segment sizes, one can obtain an understanding of the basic trends at different levels of granularity. This definition of the Haar wavelet makes it very easy to compute by a sequence of averaging and differencing operations. Table 2.2 shows the computation of the wavelet coefficients for the sequence $(8, 6, 2, 3, 4, 6, 6, 5)$. This decomposition is illustrated in graphical form in Fig. 2.5. Note that each value in the original series can be represented as a sum of $\log_2(8) = 3$ wavelet coefficients with a positive or negative sign attached in front. In general, the entire decomposition may be represented as a tree of depth 3, which represents the hierarchical decomposition of the entire series. This is also referred to as the *error tree*. In Fig. 2.6, the error tree for the wavelet decomposition in Table 2.2 is illustrated. The nodes in the tree contain the values of the wavelet coefficients, except for a special *super-root* node that contains the series average.

The number of wavelet coefficients in this series is 8, which is also the length of the original series. The original series has been replicated just below the error tree in Fig. 2.6, and can be reconstructed by adding or subtracting the values in the nodes along the path leading to that value. Each coefficient in a node should be added, if we use the left branch below it to reach to the series values. Otherwise, it should be subtracted. This natural decomposition means that an entire contiguous range along the series can be reconstructed by using only the portion of the error tree which is relevant to it.

As in all dimensionality reduction methods, smaller coefficients are ignored. We will explain the process of discarding coefficients with the help of the notion of the *basis vectors* associated with each coefficient:

The wavelet representation is a decomposition of the original time series of length q into the weighted sum of a set of q "simpler" time series (or wavelets) that are orthogonal to one another. These "simpler" time series are the basis vectors, and the wavelet coefficients represent the weights of the different basis vectors in the decomposition.

Figure 2.5 shows these "simpler" time series along with their corresponding coefficients. The number of wavelet coefficients (and basis vectors) is equal to the length of the series q.

The length of the time series representing each basis vector is also q. Each basis vector has a $+1$ or -1 value in the contiguous time-series segment from which a particular coefficient was derived by a differencing operation. Otherwise, the value is 0 because the wavelet is not related to variations in that region of the time series. The first half of the nonzero segment of the basis vector is $+1$, and the second half is -1. This gives it the shape of a *wavelet* when it is plotted as a time series, and also reflects the differencing operation in the relevant time-series segment. Multiplying a basis vector with the coefficient has the effect of creating a weighted time series in which the difference between the first half and second half reflects the average difference between the corresponding segments in the original time series. Therefore, by adding up all these weighted wavelets over different levels of granularity in the error tree, it is possible to reconstruct the original series. The list of basis vectors in Fig. 2.5 are the rows of the following matrix:

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Note that the dot product of any pair of basis vectors is 0, and therefore these series are orthogonal to one another. The most detailed coefficients have only one $+1$ and one -1, whereas the most coarse coefficient has four $+1$ and -1 entries. In addition, the vector (11111111) is needed to represent the series average.

For a time series T, let $\overline{W_1} \ldots \overline{W_q}$ be the corresponding basis vectors. Then, if $a_1 \ldots a_q$ are the wavelet coefficients for the basis vectors $\overline{W_1} \ldots \overline{W_q}$, the time series T can be represented as follows:

$$T = \sum_{i=1}^{q} a_i \overline{W_i} = \sum_{i=1}^{q} (a_i ||\overline{W_i}||) \frac{\overline{W_i}}{||\overline{W_i}||} \tag{2.15}$$

The coefficients represented in Fig. 2.5 are unnormalized because the underlying basis vectors do not have unit norm. While a_i is the unnormalized value from Fig. 2.5, the values $a_i||\overline{W_i}||$ represent normalized coefficients. The values of $||\overline{W_i}||$ are different for coefficients of different orders, and are equal to $\sqrt{2}$, $\sqrt{4}$, or $\sqrt{8}$ in this particular example. For example, in Fig. 2.5, the broadest level unnormalized coefficient is -0.25, whereas the corresponding normalized value is $-0.25\sqrt{8}$. After normalization, the basis vectors $\overline{W_1} \ldots \overline{W_q}$ are orthonormal, and, therefore, the sum of the squares of the corresponding (normalized) coefficients is equal to the retained energy in the approximated time series. Because the normalized coefficients provide a new coordinate representation after axis rotation, Euclidean distances between time series are preserved in this new representation if coefficients are not dropped. It can be shown that by retaining the coefficients with the largest normalized values, the error loss from the wavelet representation is minimized.

The previous discussion focused on the approximation of a single time series. In practice, one might want to convert a database of N time series into N multidimensional vectors. When a database of multiple time series is available, then two strategies can be used:

1. The coefficient for the same basis vector is selected for each series to create a meaningful multidimensional database of low dimensionality. Therefore, the basis vectors

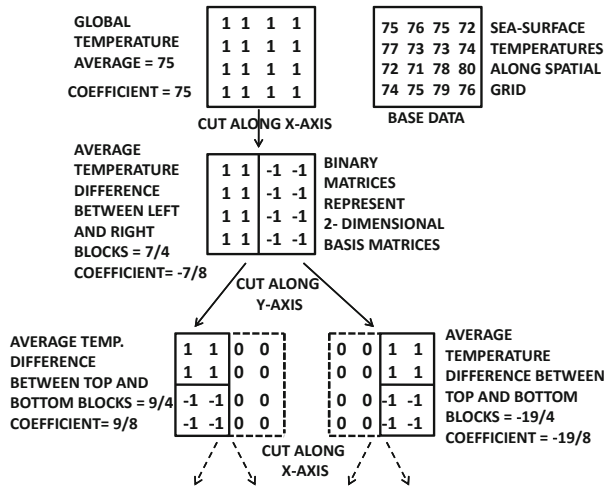


Figure 2.7: Illustration of the top levels of the wavelet decomposition for spatial data in a grid containing sea-surface temperatures

that have the largest *average* normalized coefficient across the N different series are selected.

2. The full dimensionality of the wavelet coefficient representation is retained. However, for each time series, the largest normalized coefficients (in magnitude) are selected individually. The remaining values are set to 0. This results in a sparse database of high dimensionality, in which many values are 0. A method such as *SVD* can be applied as a second step to further reduce the dimensionality. The second step of this approach has the disadvantage of losing interpretability of the features of the wavelet transform. Recall that the Haar wavelet is one of the few dimensionality reduction transforms where the coefficients do have some interpretability in terms of specific trends across particular time-series segments.

The wavelet decomposition method provides a natural method for dimensionality reduction (and data-type transformation) by retaining only a small number of coefficients.

Wavelet Decomposition with Multiple Contextual Attributes

Time-series data contain a single contextual attribute, corresponding to the time value. This helps in simplification of the wavelet decomposition. However, in some cases such as spatial data, there may be two contextual attributes corresponding to the X-coordinate and the Y-coordinate. For example, sea-surface temperatures are measured at spatial locations that are described with the use of two coordinates. How can wavelet decomposition be performed in such cases? In this section, a brief overview of the extension of wavelets to multiple contextual attributes is provided.

Assume that the spatial data is represented in the form of a 2-dimensional grid of size $q \times q$. Recall that in the 1-dimensional case, differencing operations were applied over contiguous segments of the time series by successive division of the time series in hierarchical fashion. The corresponding basis vectors have $+1$ and -1 at the relevant positions. The 2-dimensional case is completely analogous where contiguous *areas* of the spatial grid are used

by successive divisions. These divisions are alternately performed along the different axes. The corresponding basis vectors are 2-dimensional *matrices* of size $q \times q$ that regulate how the differencing operations are performed.

An example of the strategy for 2-dimensional decomposition is illustrated in Fig. 2.7. Only the top two levels of the decomposition are illustrated in the figure. Here, a 4×4 grid of spatial temperatures is used as an example. The first division along the X-axis divides the spatial area into two blocks of size 4×2 each. The corresponding two-dimensional binary basis matrix is illustrated into the same figure. The next phase divides each of these 4×2 blocks into blocks of size 2×2 during the hierarchical decomposition process. As in the case of 1-dimensional time series, the wavelet coefficient is half the difference in the average temperatures between the two halves of the relevant block being decomposed. The alternating process of division along the X-axis and the Y-axis can be carried on to the individual data entries. This creates a hierarchical wavelet error tree, which has many similar properties to that created in the 1-dimensional case. The overall principles of this decomposition are almost identical to the 1-dimensional case, with the major difference in terms of how the cuts along different dimensions are performed by alternating at different levels. The approach can be extended to the case of $k > 2$ contextual attributes with the use of a round-robin rotation in the axes that are selected at different levels of the tree for the differencing operation.

2.4.4.2 Multidimensional Scaling

Graphs are a powerful mechanism for representing relationships between objects. In some data mining scenarios, the data type of an object may be very complex and heterogeneous such as a time series annotated with text and other numeric attributes. However, a crisp notion of distance between several pairs of data objects may be available based on application-specific goals. How can one visualize the inherent similarity between these objects? How can one visualize the "nearness" of two individuals connected in a social network? A natural way of doing so is the concept of multidimensional scaling (*MDS*). Although *MDS* was originally proposed in the context of spatial visualization of graph-structured distances, it has much broader applicability for embedding data objects of arbitrary types in multidimensional space. Such an approach also enables the use of multidimensional data mining algorithms on the embedded data.

For a graph with n nodes, let $\delta_{ij} = \delta_{ji}$ denote the specified distance between nodes i and j. It is assumed that all $\binom{n}{2}$ pairwise distances between nodes are specified. It is desired to map the n nodes to n different k-dimensional vectors denoted by $\overline{X_1} \ldots \overline{X_n}$, so that the distances in multidimensional space closely correspond to the $\binom{n}{2}$ distance values in the distance graph. In *MDS*, the k coordinates of each of these n points are treated as variables that need to be optimized, so that they can *fit* the current set of pairwise distances. Metric *MDS*, also referred to as classical *MDS*, attempts to solve the following optimization (minimization) problem:

$$O = \sum_{i,j:i<j} (||\overline{X_i} - \overline{X_j}|| - \delta_{ij})^2 \tag{2.16}$$

Here $|| \cdot ||$ represents Euclidean norm. In other words, each node is represented by a multidimensional data point, such that the Euclidean distances between these points reflect the graph distances as closely as possible. In other forms of *nonmetric MDS*, this objective function might be different. This optimization problem therefore has $n \cdot k$ variables, and it scales with the size of the data n and the desired dimensionality k of the embedding. The

Table 2.3: Scaled eigenvectors of various similarity matrices yield embeddings with different properties

Method	Relevant similarity matrix
PCA	Dot product matrix DD^T after mean centering D
SVD	Dot product matrix DD^T
Spectral embedding (Symmetric Version)	Sparsified/normalized similarity matrix $\Lambda^{-1/2}W\Lambda^{-1/2}$ (cf. Sect. 19.3.4 of Chap. 19)
MDS/ISOMAP	Similarity matrix derived from distance matrix Δ with cosine law $S = -\frac{1}{2}(I - \frac{U}{n})\Delta(I - \frac{U}{n})$
Kernel PCA	Centered kernel matrix $S = (I - \frac{U}{n})K(I - \frac{U}{n})$ (cf. Sect. 10.6.4.1 of Chap. 10)

objective function O of Eq. 2.16 is usually divided by $\sum_{i,j:i<j} \delta_{ij}^2$ to yield a value in $(0, 1)$. The square root of this value is referred to as *Kruskal stress*.

The basic assumption in classical *MDS* is that the distance matrix $\Delta = [\delta_{ij}^2]_{n\times n}$ is generated by computing pairwise Euclidean distances in some hypothetical data matrix D for which the entries and dimensionality are unknown. The matrix D can never be recovered completely in classical *MDS* because Euclidean distances are invariant to mean translation and axis rotation. The appropriate conventions for the data mean and axis orientation will be discussed later. While the optimization of Eq. 2.16 requires numerical techniques, a direct solution to classical *MDS* can be obtained by eigen decomposition *under the assumption that the specified distance matrix is Euclidean*:

1. Any pairwise (squared) distance matrix $\Delta = [\delta_{ij}^2]_{n\times n}$ can be converted into a symmetric dot-product matrix $S_{n\times n}$ with the help of the *cosine law* in Euclidean space. In particular, if $\overline{X_i}$ and $\overline{X_j}$ are the embedded representations of the ith and jth nodes, the dot product between $\overline{X_i}$ and $\overline{X_j}$ can be related to the distances as follows:

$$\overline{X_i} \cdot \overline{X_j} = -\frac{1}{2}\left[||\overline{X_i} - \overline{X_j}||^2 - (||\overline{X_i}||^2 + ||\overline{X_j}||^2)\right] \quad \forall i, j \in \{1 \ldots n\} \tag{2.17}$$

For a *mean-centered* embedding, the value of $||\overline{X_i}||^2 + ||\overline{X_j}||^2$ can be expressed (see Exercise 9) in terms of the entries of the distance matrix Δ as follows:

$$||\overline{X_i}||^2 + ||\overline{X_j}||^2 = \frac{\sum_{p=1}^n ||\overline{X_i} - \overline{X_p}||^2}{n} + \frac{\sum_{q=1}^n ||\overline{X_j} - \overline{X_q}||^2}{n} - \frac{\sum_{p=1}^n \sum_{q=1}^n ||\overline{X_p} - \overline{X_q}||^2}{n^2} \tag{2.18}$$

A mean-centering assumption is necessary because the Euclidean distance is mean invariant, whereas the dot product is not. By substituting Eq. 2.18 in Eq. 2.17, it is possible to express the dot product $\overline{X_i} \cdot \overline{X_j}$ fully in terms of the entries of the distance matrix Δ. Because this condition is true for all possible values of i and j, we can conveniently express it in $n \times n$ matrix form. Let U be the $n \times n$ matrix of all 1s, and let I be the identity matrix. Then, our argument above shows that the dot-product matrix S is equal to $-\frac{1}{2}(I - \frac{U}{n})\Delta(I - \frac{U}{n})$. Under the Euclidean assumption, the matrix S is always positive semidefinite because it is equal to the $n \times n$ dot-product matrix DD^T of the *unobserved* data matrix D, which has unknown dimensionality. Therefore, it is desired to determine a high-quality factorization of S into the form $D_k D_k^T$, where D_k is an $n \times k$ matrix of dimensionality k.

2. Such a factorization can be obtained with eigen decomposition. Let $S \approx Q_k \Sigma_k^2 Q_k^T = (Q_k \Sigma_k)(Q_k \Sigma_k)^T$ represent the approximate diagonalization of S, where Q_k is an $n \times k$ matrix containing the largest k eigenvectors of S, and Σ_k^2 is a $k \times k$ diagonal matrix containing the eigenvalues. The embedded representation is given by $D_k = Q_k \Sigma_k$. Note that *SVD* also derives the optimal embedding as the scaled eigenvectors of the dot-product matrix of the original data. Therefore, the squared error of representation is minimized by this approach. This can also be shown to be equivalent to minimizing the Kruskal stress.

The optimal solution is not unique, because we can multiply $Q_k \Sigma_k$ with any $k \times k$ matrix with orthonormal columns, and the pairwise Euclidean distances will not be affected. In other words, any representation of $Q_k \Sigma_k$ in a rotated axis system is optimal as well. *MDS* finds an axis system like *PCA* in which the individual attributes are uncorrelated. In fact, if classical *MDS* is applied to a distance matrix Δ, which is constructed by computing the pairwise Euclidean distances in an actual data set, then it will yield the same embedding as the application of *PCA* on that data set. *MDS* is useful when such a data set is not available to begin with, and only the distance matrix Δ is available.

As in all dimensionality reduction methods, the value of the dimensionality k provides the trade-off between representation size and accuracy. Larger values of the dimensionality k will lead to lower stress. A larger number of data points typically requires a larger dimensionality of representation to achieve the same stress. The most crucial element is, however, the inherent structure of the distance matrix. For example, if a $10,000 \times 10,000$ distance matrix contains the pairwise driving distance between 10,000 cities, it can usually be approximated quite well with just a 2-dimensional representation. This is because driving distances are an approximation of Euclidean distances in 2-dimensional space. On the other hand, an arbitrary distance matrix may not be Euclidean and the distances may not even satisfy the triangle inequality. As a result, the matrix S might not be positive semidefinite. In such cases, it is sometimes still possible to use the metric assumption to obtain a high-quality embedding. Specifically, only those positive eigenvalues may be used, whose magnitude exceeds that of the most negative eigenvalue. This approach will work reasonably well if the negative eigenvalues have small magnitude.

MDS is commonly used in nonlinear dimensionality reduction methods such as *ISOMAP* (cf. Sect. 3.2.1.7 of Chap. 3). It is noteworthy that, in conventional *SVD*, the scaled eigenvectors of the $n \times n$ dot-product similarity matrix DD^T yield a low-dimensional embedded representation of D just as the eigenvectors of S yield the embedding in *MDS*. The eigen decomposition of similarity matrices is fundamental to many linear and nonlinear dimensionality reduction methods such as *PCA*, *SVD*, *ISOMAP*, *kernel PCA*, and spectral embedding. The specific properties of each embedding are a result of the choice of the similarity matrix and the scaling used on the resulting eigenvectors. Table 2.3 provides a preliminary comparison of these methods, although some of them are discussed in detail only in later chapters.

2.4.4.3 Spectral Transformation and Embedding of Graphs

Whereas *MDS* methods are designed for preserving global distances, spectral methods are designed for preserving local distances for applications such as clustering. Spectral methods work with *similarity graphs* in which the weights on the edges represent similarity rather than distances. When distance values are available they are converted to similarity values with kernel functions such as the heat kernel discussed earlier in this chapter. The notion

of similarity is natural to many real Web, social, and information networks because of the notion of *homophily*. For example, consider a bibliographic network in which nodes correspond to authors, and the edges correspond to co-authorship relations. The weight of an edge represents the number of publications between authors and therefore represents one possible notion of similarity in author publications. Similarity graphs can also be constructed between arbitrary data types. For example, a set of n time series can be converted into a graph with n nodes, where a node represents each time series. The weight of an edge is equal to the similarity between the two nodes, and only edges with a "sufficient" level of similarity are retained. A discussion of the construction of the similarity graph is provided in Sect. 2.2.2.9. Therefore, if a similarity graph can be transformed to a multidimensional representation that preserves the similarity structure between nodes, it provides a transformation that can port virtually any data type to the easily usable multidimensional representation. The caveat here is that such a transformation can only be used for similarity-based applications such as clustering or nearest neighbor classification because the transformation is designed to preserve the *local* similarity structure. The local similarity structure of a data set is nevertheless fundamental to many data mining applications.

Let $G = (N, A)$ be an undirected graph with node set N and edge set A. It is assumed that the node set contains n nodes. A symmetric $n \times n$ weight matrix $W = [w_{ij}]$ represents the similarities between the different nodes. Unlike *MDS*, which works with a complete graph of *global* distances, this graph is generally a *sparsified* representation of the similarity of each object to its k *nearest objects* (cf. Sect. 2.2.2.9). The similarities to the remaining objects are not distinguished from one another and set to 0. This is because spectral methods preserve only the local similarity structure for applications such as clustering. All entries in this matrix are assumed to be nonnegative, and higher values indicate greater similarity. If an edge does not exist between a pair of nodes, then the corresponding entry is assumed to be 0. It is desired to embed the nodes of this graph into a k-dimensional space so that the similarity structure of the data is preserved.

First, let us discuss the much simpler problem of mapping the nodes onto a 1-dimensional space. The generalization to the k-dimensional case is relatively straightforward. We would like to map the nodes in N into a set of 1-dimensional real values $y_1 \ldots y_n$ on a line, so that the distances between these points reflect the edge connectivity among the nodes. Therefore, it is undesirable for nodes that are connected with high-weight edges, to be mapped onto distant points on this line. Therefore, we would like to determine values of y_i that minimize the following objective function O:

$$O = \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij}(y_i - y_j)^2 \tag{2.19}$$

This objective function penalizes the distances between y_i and y_j with weight proportional to w_{ij}. Therefore, when w_{ij} is very large (more similar nodes), the data points y_i and y_j will be more likely to be closer to one another in the embedded space. The objective function O can be rewritten in terms of the *Laplacian matrix* L of weight matrix W. The Laplacian matrix L is defined as $\Lambda - W$, where Λ is a diagonal matrix satisfying $\Lambda_{ii} = \sum_{j=1}^{n} w_{ij}$. Let the n-dimensional column vector of embedded values be denoted by $\overline{y} = (y_1 \ldots y_n)^T$. It can be shown after some algebraic simplification that the minimization objective function O can be rewritten in terms of the Laplacian matrix:

$$O = 2\overline{y}^T L \overline{y} \tag{2.20}$$

The matrix L is positive semidefinite with nonnegative eigenvalues because the sum-of-squares objective function O is always nonnegative. We need to incorporate a scaling constraint to ensure that the trivial value of $y_i = 0$ for all i, is not selected by the optimization solution. A possible scaling constraint is as follows:

$$\overline{y}^T \Lambda \overline{y} = 1 \tag{2.21}$$

The use of the matrix Λ in the constraint of Eq. 2.21 is essentially a normalization constraint, which is discussed in detail in Sect. 19.3.4 of Chap. 19.

It can be shown that the value of O is optimized by selecting $\overline{y}$ as the smallest eigenvector of the relationship $\Lambda^{-1} L \overline{y} = \lambda \overline{y}$. However, the smallest eigenvalue is always 0, and it corresponds to the trivial solution where the node embedding $\overline{y}$ is proportional to the vector containing only 1s. This trivial eigenvector is non-informative because it corresponds to an embedding in which every node is mapped to the same point. Therefore, it can be discarded, and it is not used in the analysis. The second-smallest eigenvector then provides an optimal solution that is more informative.

This solution can be generalized to finding an optimal k-dimensional embedding by determining successive directions corresponding to eigenvectors with increasing eigenvalues. After discarding the first trivial eigenvector $\overline{e_1}$ with eigenvalue $\lambda_1 = 0$, this results in a set of k eigenvectors $\overline{e_2}, \overline{e_3} \ldots \overline{e_{k+1}}$, with corresponding eigenvalues $\lambda_2 \leq \lambda_3 \leq \ldots \leq \lambda_{k+1}$. Each eigenvector is of length n and contains one coordinate value for each node. The ith value along the jth eigenvector represents the jth coordinate of the ith node. This creates an $n \times k$ matrix, corresponding to the k-dimensional embedding of the n nodes.

What do the small magnitude eigenvectors intuitively represent in the new transformed space? By using the ordering of the nodes along a small magnitude eigenvector to create a cut, the weight of the edges across the cut is likely to be small. Thus, this represents a cluster in the space of nodes. In practice, the k smallest eigenvectors (after ignoring the first) are selected to perform the reduction and create a k-dimensional embedding. This embedding typically contains an excellent representation of the underlying similarity structure of the nodes. The embedding can be used for virtually any similarity-based application, although the most common application of this approach is spectral clustering. Many variations of this approach exist in terms of how the Laplacian L is normalized, and in terms of how the final clusters are generated. The spectral clustering method will be discussed in detail in Sect. 19.3.4 of Chap. 19.

2.5 Summary

Data preparation is an important part of the data mining process because of the sensitivity of the analytical algorithms to the quality of the input data. The data mining process requires the collection of raw data from a variety of sources that may be in a form which is unsuitable for direct application of analytical algorithms. Therefore, numerous methods may need to be applied to extract features from the underlying data. The resulting data may have significant missing values, errors, inconsistencies, and redundancies. A variety of analytical methods and data scrubbing tools exist for imputing the missing entries or correcting inconsistencies in the data.

Another important issue is that of data heterogeneity. The analyst may be faced with a multitude of attributes that are distinct, and therefore the direct application of data mining algorithms may not be easy. Therefore, data type portability is important, wherein some subsets of attributes are converted to a predefined format. The multidimensional

format is often preferred because of its simplicity. Virtually, any data type can be converted to multidimensional representation with the two-step process of constructing a similarity graph, followed by multidimensional embedding.

The data set may be very large, and it may be desirable to reduce its size both in terms of the number of rows and the number of dimensions. The reduction in terms of the number of rows is straightforward with the use of sampling. To reduce the number of columns in the data, either feature subset selection or data transformation may be used. In feature subset selection, only a smaller set of features is retained that is most suitable for analysis. These methods are closely related to analytical methods because the relevance of a feature may be application dependent. Therefore, the feature selection phase need to be tailored to the specific analytical method.

There are two types of feature transformation. In the first type, the axis system may be rotated to align with the correlations of the data and retain the directions with the greatest variance. The second type is applied to complex data types such as graphs and time series. In these methods, the size of the representation is reduced, and the data is also transformed to a multidimensional representation.

2.6 Bibliographic Notes

The problem of feature extraction is an important one for the data mining process but it is highly application specific. For example, the methods for extracting named entities from a document data set [400] are very different from those that extract features from an image data set [424]. An overview of some of the promising technologies for feature extraction in various domains may be found in [245].

After the features have been extracted from different sources, they need to be integrated into a single database. Numerous methods have been described in the conventional database literature for data integration [194, 434]. Subsequently, the data needs to be cleaned and missing entries need to be removed. A new field of probabilistic or uncertain data has emerged [18] that models uncertain and erroneous records in the form of probabilistic databases. This field is, however, still in the research stage and has not entered the mainstream of database applications. Most of the current methods either use tools for missing data analysis [71, 364] or more conventional data cleaning and data scrubbing tools [222, 433, 435].

After the data has been cleaned, its size needs to be reduced either in terms of numerosity or in terms of dimensionality. The most common and simple numerosity reduction method is sampling. Sampling methods can be used for either static data sets or dynamic data sets. Traditional methods for data sampling are discussed in [156]. The method of sampling has also been extended to data streams in the form of reservoir sampling [35, 498]. The work in [35] discusses the extension of reservoir sampling methods to the case where a biased sample needs to be created from the data stream.

Feature selection is an important aspect of the data mining process. The approach is often highly dependent on the particular data mining algorithm being used. For example, a feature selection method that works well for clustering may not work well for classification. Therefore, we have deferred the discussion of feature selection to the relevant chapters on the topic on clustering and classification in this book. Numerous books are available on the topic of feature selection [246, 366].

The two most common dimensionality reduction methods used for multidimensional data are *SVD* [480, 481] and *PCA* [295]. These methods have also been extended to text in

the form of *LSA* [184, 416]. It has been shown in many domains [25, 184, 416] that the use of methods such as *SVD*, *LSA*, and *PCA* unexpectedly improves the quality of the underlying representation after performing the reduction. This improvement is because of reduction in noise effects by discarding the low-variance dimensions. Applications of *SVD* to data imputation are found in [23] and Chap. 18 of this book. Other methods for dimensionality reduction and transformation include Kalman filtering [260], Fastmap [202], and nonlinear methods such as *Laplacian eigenmaps* [90], *MDS* [328], and *ISOMAP* [490].

Many dimensionality reduction methods have also been proposed in recent years that simultaneously perform type transformation together with the reduction process. These include wavelet transformation [475] and graph embedding methods such as *ISOMAP* and *Laplacian eigenmaps* [90, 490]. A tutorial on spectral methods for graph embedding may be found in [371].

2.7 Exercises

1. Consider the time-series $(-3, -1, 1, 3, 5, 7, *)$. Here, a missing entry is denoted by $*$. What is the estimated value of the missing entry using linear interpolation on a window of size 3?

2. Suppose you had a bunch of text documents, and you wanted to determine all the personalities mentioned in these documents. What class of technologies would you use to achieve this goal?

3. Download the *Arrythmia* data set from the *UCI Machine Learning Repository* [213]. Normalize all records to a mean of 0 and a standard deviation of 1. Discretize each numerical attribute into (a) 10 equi-width ranges and (b) 10 equi-depth ranges.

4. Suppose that you had a set of arbitrary objects of different types representing different characteristics of widgets. A domain expert gave you the similarity value between every pair of objects. How would you convert these objects into a multidimensional data set for clustering?

5. Suppose that you had a data set, such that each data point corresponds to sea-surface temperatures over a square mile of resolution 10×10. In other words, each data record contains a 10×10 grid of temperature values with spatial locations. You also have some text associated with each 10×10 grid. How would you convert this data into a multidimensional data set?

6. Suppose that you had a set of discrete biological protein sequences that are annotated with text describing the properties of the protein. How would you create a multidimensional representation from this heterogeneous data set?

7. Download the *Musk* data set from the *UCI Machine Learning Repository* [213]. Apply *PCA* to the data set, and report the eigenvectors and eigenvalues.

8. Repeat the previous exercise using *SVD*.

9. For a mean-centered data set with points $\overline{X_1} \ldots \overline{X_n}$, show that the following is true:

$$||\overline{X_i}||^2 + ||\overline{X_j}||^2 = \frac{\sum_{p=1}^{n} ||\overline{X_i} - \overline{X_p}||^2}{n} + \frac{\sum_{q=1}^{n} ||\overline{X_j} - \overline{X_q}||^2}{n} - \frac{\sum_{p=1}^{n} \sum_{q=1}^{n} ||\overline{X_p} - \overline{X_q}||^2}{n^2} \tag{2.22}$$

10. Consider the time series $1, 1, 3, 3, 3, 3, 1, 1$. Perform wavelet decomposition on the time series. How many coefficients of the series are nonzero?

11. Download the *Intel Research Berkeley data set*. Apply a wavelet transformation to the temperature values in the first sensor.

12. Treat each quantitative variable in the *KDD CUP 1999 Network Intrusion Data Set* from the *UCI Machine Learning Repository* [213] as a time series. Perform the wavelet decomposition of this time series.

13. Create samples of size $n = 1, 10, 100, 1000, 10000$ records from the data set of the previous exercise, and determine the average value e_i of each quantitative column i using the sample. Let μ_i and σ_i be the global mean and standard deviation over the entire data set. Compute the number of standard deviations z_i by which e_i varies from μ_i.

$$z_i = \frac{|e_i - \mu_i|}{\sigma_i}$$

How does z_i vary with n?

14. Show that any right singular vector $\overline{y}$ of A with 0 singular value satisfies $A\overline{y} = 0$.

15. Show that the diagonalization of a square matrix is a specialized variation of *SVD*.

Chapter 3

Similarity and Distances

"*Love is the power to see similarity in the dissimilar.*"—Theodor Adorno

3.1 Introduction

Many data mining applications require the determination of similar or dissimilar objects, patterns, attributes, and events in the data. In other words, a methodical way of quantifying similarity between data objects is required. Virtually all data mining problems, such as clustering, outlier detection, and classification, require the computation of similarity. A formal statement of the problem of similarity or distance quantification is as follows:

Given two objects O_1 and O_2, determine a value of the similarity $Sim(O_1, O_2)$ (or distance $Dist(O_1, O_2)$) between the two objects.

In similarity functions, larger values imply greater similarity, whereas in distance functions, smaller values imply greater similarity. In some domains, such as spatial data, it is more natural to talk about distance functions, whereas in other domains, such as text, it is more natural to talk about similarity functions. Nevertheless, the principles involved in the design of such functions are generally invariant across different data domains. This chapter will, therefore, use either of the terms "distance function" and "similarity function," depending on the domain at hand. Similarity and distance functions are often expressed in closed form (e.g., Euclidean distance), but in some domains, such as time-series data, they are defined algorithmically and cannot be expressed in closed form.

Distance functions are fundamental to the effective design of data mining algorithms, because a poor choice in this respect may be very detrimental to the quality of the results. Sometimes, data analysts use the Euclidean function as a "black box" without much thought about the overall impact of such a choice. It is not uncommon for an inexperienced analyst to invest significant effort in the algorithmic design of a data mining problem, while treating the distance function subroutine as an afterthought. This is a mistake. As this chapter will elucidate, poor choices of the distance function can sometimes be disastrously misleading

C. C. Aggarwal, *Data Mining: The Textbook*, DOI 10.1007/978-3-319-14142-8_3

depending on the application domain. Good distance function design is also crucial for type portability. As discussed in Sect. 2.4.4.3 of Chap. 2, spectral embedding can be used to convert a similarity graph constructed on any data type into multidimensional data.

Distance functions are highly sensitive to the data distribution, dimensionality, and data type. In some data types, such as multidimensional data, it is much simpler to define and compute distance functions than in other types such as time-series data. In some cases, user intentions (or *training feedback* on object pairs) are available to *supervise* the distance function design. Although this chapter will primarily focus on unsupervised methods, we will also briefly touch on the broader principles of using supervised methods.

This chapter is organized as follows. Section 3.2 studies distance functions for multidimensional data. This includes quantitative, categorical, and mixed attribute data. Similarity measures for text, binary, and set data are discussed in Sect. 3.3. Temporal data is discussed in Sect. 3.4. Distance functions for graph data are addressed in Sect. 3.5. A discussion of supervised similarity will be provided in Sect. 3.6. Section 3.7 gives a summary.

3.2 Multidimensional Data

Although multidimensional data are the simplest form of data, there is significant diversity in distance function design across different attribute types such as categorical or quantitative data. This section will therefore study each of these types separately.

3.2.1 Quantitative Data

The most common distance function for quantitative data is the L_p-norm. The L_p-norm between two data points $\overline{X} = (x_1 \ldots x_d)$ and $\overline{Y} = (y_1 \ldots y_d)$ is defined as follows:

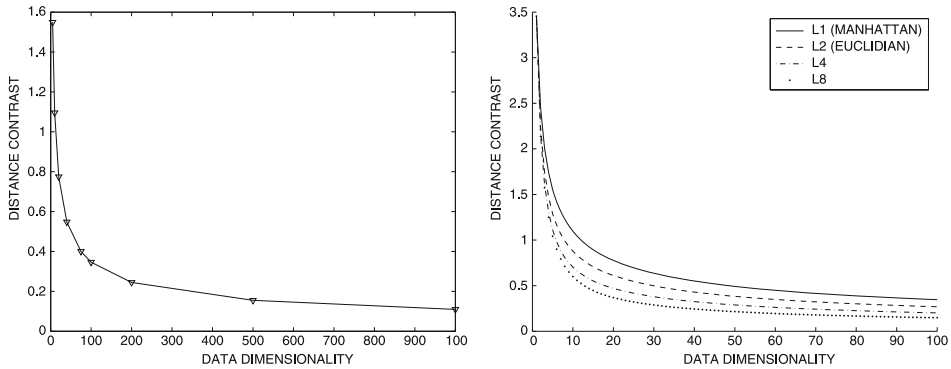
$$Dist(\overline{X}, \overline{Y}) = \left(\sum_{i=1}^{d} |x_i - y_i|^p \right)^{1/p}. \tag{3.1}$$

Two special cases of the L_p-norm are the *Euclidean* ($p = 2$) and the *Manhattan* ($p = 1$) metrics. These special cases derive their intuition from spatial applications where they have clear physical interpretability. The Euclidean distance is the straight-line distance between two data points. The Manhattan distance is the "city block" driving distance in a region in which the streets are arranged as a rectangular grid, such as the Manhattan Island of New York City.

A nice property of the Euclidean distance is that it is rotation-invariant because the straight-line distance between two data points does not change with the orientation of the axis system. This property also means that transformations, such as *PCA*, *SVD*, or the wavelet transformation for time series (discussed in Chap. 2), can be used on the data without affecting[1] the distance. Another interesting special case is that obtained by setting $p = \infty$. The result of this computation is to select the dimension for which the two objects are the most distant from one another and report the absolute value of this distance. All other features are ignored.*a*

The L_p-norm is one of the most popular distance functions used by data mining analysts. One of the reasons for its popularity is the natural intuitive appeal and interpretability of L_1- and L_2-norms in spatial applications. The intuitive interpretability of these distances does not, however, mean that they are the most relevant ones, especially for the high-dimensional case. In fact, these distance functions may not work very well when the data

[1]The distances are affected after dimensions are dropped. However, the transformation itself does not impact distances.



(a) Contrasts with dimensionality (b) Contrasts with norms

Figure 3.1: Reduction in distance contrasts with increasing dimensionality and norms

are high dimensional because of the varying impact of data sparsity, distribution, noise, and feature relevance. This chapter will discuss these broader principles in the context of distance function design.

3.2.1.1 Impact of Domain-Specific Relevance

In some cases, an analyst may know which features are more important than others for a particular application. For example, for a credit-scoring application, an attribute such as salary is much more relevant to the design of the distance function than an attribute such as gender, though both may have some impact. In such cases, the analyst may choose to weight the features differently if domain-specific knowledge about the relative importance of different features is available. This is often a heuristic process based on experience and skill. The generalized L_p-distance is most suitable for this case and is defined in a similar way to the L_p-norm, except that a coefficient a_i is associated with the ith feature. This coefficient is used to weight the corresponding feature component in the L_p-norm:

$$Dist(\overline{X}, \overline{Y}) = \left(\sum_{i=1}^{d} a_i \cdot |x_i - y_i|^p \right)^{1/p}. \tag{3.2}$$

This distance is also referred to as the generalized *Minkowski distance*. In many cases, such domain knowledge is not available. Therefore, the L_p-norm may be used as a default option. Unfortunately, without knowledge about the most relevant features, the L_p-norm is susceptible to some undesirable effects of increasing dimensionality, as discussed subsequently.

3.2.1.2 Impact of High Dimensionality

Many distance-based data mining applications lose their effectiveness as the dimensionality of the data increases. For example, a distance-based clustering algorithm may group unrelated data points because the distance function may poorly reflect the intrinsic semantic distances between data points with increasing dimensionality. As a result, distance-based models of clustering, classification, and outlier detection are often *qualitatively* ineffective. This phenomenon is referred to as the "curse of dimensionality," a term first coined by Richard Bellman.

To better understand the impact of the dimensionality curse on distances, let us examine a unit cube of dimensionality d that is fully located in the nonnegative quadrant, with one corner at the origin $\overline{O}$. What is the Manhattan distance of the corner of this cube (say, at the origin) to a randomly chosen point $\overline{X}$ inside the cube? In this case, because one end point is the origin, and all coordinates are nonnegative, the Manhattan distance will sum up the coordinates of $\overline{X}$ over the different dimensions. Each of these coordinates is uniformly distributed in $[0, 1]$. Therefore, if Y_i represents the uniformly distributed random variable in $[0, 1]$, it follows that the Manhattan distance is as follows:

$$Dist(\overline{O}, \overline{X}) = \sum_{i=1}^{d} (Y_i - 0). \tag{3.3}$$

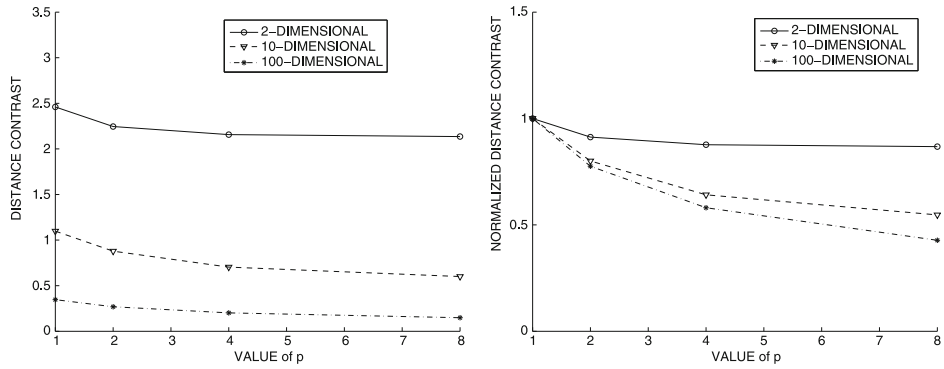
The result is a random variable with a mean of $\mu = d/2$ and a standard deviation of $\sigma = \sqrt{d/12}$. For large values of d, it can be shown by the law of large numbers that the vast majority of randomly chosen points inside the cube will lie in the range $[D_{min}, D_{max}] = [\mu - 3\sigma, \mu + 3\sigma]$. Therefore, most of the points in the cube lie within a distance range of $D_{max} - D_{min} = 6\sigma = \sqrt{3d}$ from the origin. Note that the expected Manhattan distance grows with dimensionality at a rate that is linearly proportional to d. Therefore, the *ratio* of the variation in the distances to the absolute values that is referred to as $Contrast(d)$, is given by:

$$\text{Contrast}(d) = \frac{D_{max} - D_{min}}{\mu} = \sqrt{12/d}. \tag{3.4}$$

This ratio can be interpreted as the distance *contrast* between the different data points, in terms of how different the minimum and maximum distances from the origin might be considered. Because the contrast reduces with $\sqrt{d}$, it means that there is virtually no contrast with increasing dimensionality. Lower contrasts are obviously not desirable because it means that the data mining algorithm will score the distances between all pairs of data points in approximately the same way and will not discriminate well between different pairs of objects with varying levels of semantic relationships. The variation in contrast with increasing dimensionality is shown in Fig. 3.1a. This behavior is, in fact, observed for all L_p-norms at different values of p, though with varying severity. These differences in severity will be explored in a later section. Clearly, with increasing dimensionality, a direct use of the L_p-norm may not be effective.

3.2.1.3 Impact of Locally Irrelevant Features

A more fundamental way of exploring the effects of high dimensionality is by examining the impact of irrelevant features. This is because many features are likely to be irrelevant in a typical high-dimensional data set. Consider, for example, a set of medical records, containing patients with diverse medical conditions and very extensive quantitative measurements about various aspects of an individual's medical history. For a cluster containing diabetic patients, certain attributes such as the blood glucose level are more important for the distance computation. On the other hand, for a cluster containing epileptic patients, a different set of features will be more important. The additive effects of the natural variations in the many attribute values may be quite significant. A distance metric such as the Euclidean metric may unnecessarily contribute a high value from the more noisy components because of its square-sum approach. The key point to understand here is that the precise features that are relevant to the distance computation may sometimes be sensitive to the particular pair of objects that are being compared. This problem cannot be solved by global feature



(a) Contrast (b) Contrast/Manhattan Contrast

Figure 3.2: Impact of p on contrast

subset selection during preprocessing, because the relevance of features is *locally* determined by the pair of objects that are being considered. Globally, all features may be relevant.

When many features are irrelevant, the additive noise effects of the irrelevant features can sometimes be reflected in the concentration of the distances. In any case, such irrelevant features will almost always result in errors in distance computation. Because high-dimensional data sets are often likely to contain diverse features, many of which are irrelevant, the additive effect with the use of a sum-of-squares approach, such as the L_2-norm, can be very detrimental.

3.2.1.4 Impact of Different L_p-Norms

Different L_p-norms do not behave in a similar way either in terms of the impact of irrelevant features or the distance contrast. Consider the extreme case when $p = \infty$. This translates to using only the dimension where the two objects are the most *dissimilar*. Very often, this may be the impact of the natural variations in an irrelevant attribute that is not too useful for a similarity-based application. In fact, for a 1000-dimensional application, if two objects have similar values on 999 attributes, such objects should be considered *very* similar. However, a single irrelevant attribute on which the two objects are very different will throw off the distance value in the case of the L_∞ metric. In other words, local similarity properties of the data are de-emphasized by L_∞. Clearly, this is not desirable.

This behavior is generally true for larger values of p, where the irrelevant attributes are emphasized. In fact, it can also be shown that distance contrasts are also poorer for larger values of p for certain data distributions. In Fig. 3.1b, the distance contrasts have been illustrated for different values of p for the L_p-norm over different dimensionalities. The figure is constructed using the same approach as Fig. 3.1a. While all L_p-norms degrade with increasing dimensionality, the degradation is much faster for the plots representing larger values of p. This trend can be understood better from Fig. 3.2 where the value of p is used on the X-axis. In Fig. 3.2a, the contrast is illustrated with different values of p for data of different dimensionalities. Figure 3.2b is derived from Fig. 3.2a, except that the results show the fraction of the Manhattan performance achieved by higher order norms. It is evident that the *rate* of degradation with increasing p is higher when the dimensionality of the data is large. For 2-dimensional data, there is very little degradation. This is the reason that the value of p matters less in lower dimensional applications.

This argument has been used to propose the concept of fractional metrics, for which $p \in (0, 1)$. Such fractional metrics can provide more effective results for the high-dimensional case. As a rule of thumb, the larger the dimensionality, the lower the value of p. However, no exact rule exists on the precise choice of p because dimensionality is not the only factor in determining the proper value of p. The precise choice of p should be selected in an application-specific way, with the use of benchmarking. The bibliographic notes contain discussions on the use of fractional metrics.

3.2.1.5 Match-Based Similarity Computation

Because it is desirable to select locally relevant features for distance computation, a question arises as to how this can be achieved in a meaningful and practical way for data mining applications. A simple approach that is based on the cumulative evidence of matching many attribute values has been shown to be effective in many scenarios. This approach is also relatively easy to implement efficiently.

A broader principle that seems to work well for high-dimensional data is that the impact of the noisy variation along individual attributes needs to be de-emphasized while counting the cumulative match across many dimensions. Of course, such an approach poses challenges for low-dimensional data, because the cumulative impact of matching cannot be counted in a statistically robust way with a small number of dimensions. Therefore, an approach is needed that can automatically adjust to the dimensionality of the data.

With increasing dimensionality, a record is likely to contain both relevant and irrelevant features. A pair of semantically similar objects may contain feature values that are dissimilar (at the level of one standard deviation along that dimension) because of the noisy variations in irrelevant features. Conversely, a pair of objects are unlikely to have similar values across many attributes, just by chance, unless these attributes were relevant. Interestingly, the Euclidean metric (and L_p-norm in general) achieves exactly the opposite effect by using the squared sum of the difference in attribute values. As a result, the "noise" components from the irrelevant attributes dominate the computation and mask the similarity effects of a large number of relevant attributes. The L_∞-norm provides an extreme example of this effect where the dimension with the largest distance value is used. In high-dimensional domains such as text, similarity functions such as the cosine measure (discussed in Sect. 3.3), tend to emphasize the cumulative effect of matches on many attribute values rather than large distances along individual attributes. This general principle can also be used for quantitative data.

One way of de-emphasizing precise levels of dissimilarity is to use *proximity thresholding* in a dimensionality-sensitive way. To perform proximity thresholding, the data are discretized into equidepth buckets. Each dimension is divided into k_d equidepth buckets, containing a fraction $1/k_d$ of the records. The number of buckets, k_d, is dependent on the data dimensionality d.

Let $\overline{X} = (x_1 \ldots x_d)$ and $\overline{Y} = (y_1 \ldots y_d)$ be two d-dimensional records. Then, for dimension i, if both x_i and y_i belong to the same bucket, the two records are said to be in proximity on dimension i. The subset of dimensions on which $\overline{X}$ and $\overline{Y}$ map to the same bucket is referred to as the proximity set, and it is denoted by $\mathcal{S}(\overline{X}, \overline{Y}, k_d)$. Furthermore, for each dimension $i \in \mathcal{S}(\overline{X}, \overline{Y}, k_d)$, let m_i and n_i be the upper and lower bounds of the bucket in dimension i, in which the two records are proximate to one another. Then, the

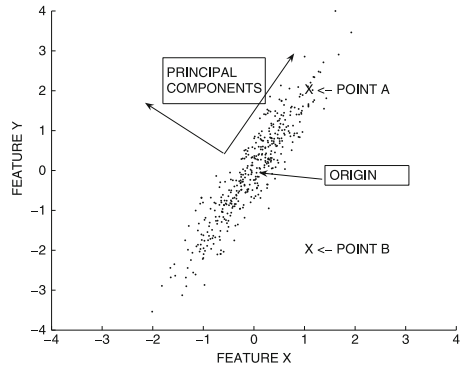


Figure 3.3: Global data distributions impact distance computations

similarity $PSelect(\overline{X}, \overline{Y}, k_d)$ is defined as follows:

$$PSelect(\overline{X}, \overline{Y}, k_d) = \left[\sum_{i \in \mathcal{S}(\overline{X}, \overline{Y}, k_d)} \left(1 - \frac{|x_i - y_i|}{m_i - n_i}\right)^p \right]^{1/p}. \tag{3.5}$$

The value of the aforementioned expression will vary between 0 and $|\mathcal{S}(\overline{X}, \overline{Y}, k_d)|$ because each individual expression in the summation lies between 0 and 1. This is a *similarity* function because larger values imply greater similarity.

The aforementioned similarity function guarantees a nonzero similarity component only for dimensions mapping to the same bucket. The use of equidepth partitions ensures that the probability of two records sharing a bucket for a particular dimension is given by $1/k_d$. Thus, on average, the aforementioned summation is likely to have d/k_d nonzero components. For more similar records, the number of such components will be greater, and each individual component is also likely to contribute more to the similarity value. The degree of dissimilarity on the distant dimensions is ignored by this approach because it is often dominated by noise. It has been shown theoretically [7] that picking $k_d \propto d$ achieves a constant level of contrast in high-dimensional space for certain data distributions. High values of k_d result in more stringent quality bounds for each dimension. These results suggest that in high-dimensional space, it is better to aim for higher quality bounds for each dimension, so that a smaller percentage (not number) of retained dimensions are used in similarity computation. An interesting aspect of this distance function is the nature of its sensitivity to data dimensionality. The choice of k_d with respect to d ensures that for low-dimensional applications, it bears some resemblance to the L_p-norm by using most of the dimensions; whereas for high-dimensional applications, it behaves similar to text domain-like similarity functions by using similarity on matching attributes. The distance function has also been shown to be more effective for a prototypical nearest-neighbor classification application.

3.2.1.6 Impact of Data Distribution

The L_p-norm depends only on the two data points in its argument and is invariant to the global statistics of the remaining data points. Should distances depend on the underlying data distribution of the remaining points in the data set? The answer is yes. To illustrate this point, consider the distribution illustrated in Fig. 3.3 that is centered at the origin. In

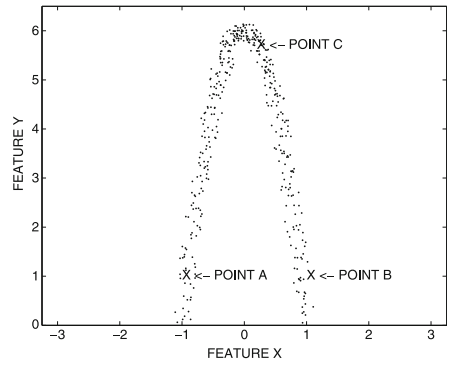


Figure 3.4: Impact of nonlinear distributions on distance computations

addition, two data points A= $(1, 2)$ and B= $(1, -2)$ are marked in the figure. Clearly, A and B are equidistant from the origin according to any L_p-norm. However, a question arises, as to whether A and B should truly be considered equidistant from the origin O. This is because the straight line from O to A is aligned with a *high-variance* direction in the data, and statistically, it is more *likely* for data points to be further away in this direction. On the other hand, many segments of the path from O to B are sparsely populated, and the corresponding direction is a *low-variance* direction. Statistically, it is much less likely for B to be so far away from O along this direction. Therefore, the distance from O to A *ought* to be less than that of O to B.

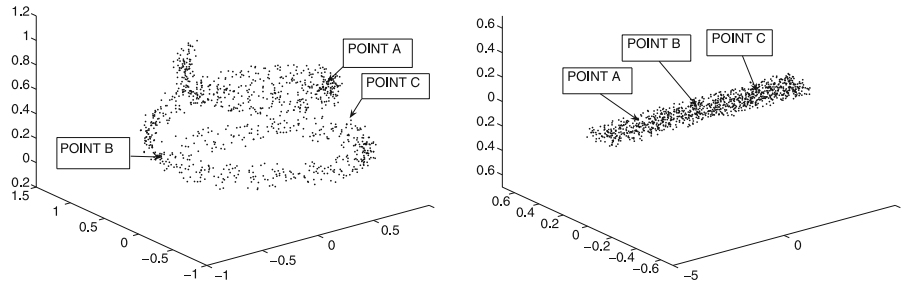
The *Mahalanobis distance* is based on this general principle. Let Σ be its $d \times d$ covariance matrix of the data set. In this case, the (i, j)th entry of the covariance matrix is equal to the covariance between the dimensions i and j. Then, the Mahalanobis distance $Maha(\overline{X}, \overline{Y})$ between two d-dimensional data points $\overline{X}$ and $\overline{Y}$ is as follows:

$$Maha(\overline{X}, \overline{Y}) = \sqrt{(\overline{X} - \overline{Y})\Sigma^{-1}(\overline{X} - \overline{Y})^T}.$$

A different way of understanding the Mahalanobis distance is in terms of principal component analysis (*PCA*). The Mahalanobis distance is similar to the Euclidean distance, except that it normalizes the data on the basis of the interattribute correlations. For example, if the axis system were to be rotated to the principal directions of the data (shown in Fig. 3.3), then the data would have no (second order) interattribute correlations. The Mahalanobis distance is equivalent to the Euclidean distance in such a transformed (axes-rotated) data set *after* dividing each of the transformed coordinate values by the standard deviation of the data along that direction. As a result, the data point B will have a larger distance from the origin than data point A in Fig. 3.3.

3.2.1.7 Nonlinear Distributions: ISOMAP

We now examine the case in which the data contain nonlinear distributions of arbitrary shape. For example, consider the global distribution illustrated in Fig. 3.4. Among the three data points A, B, and C, which pair are the closest to one another? At first sight, it would seem that data points A and B are the closest on the basis of Euclidean distance. However, the global data distribution tells us otherwise. One way of understanding distances is as the shortest length of the path from one data point to another, when using only point-to-point jumps from data points to one of their k-nearest neighbors based on a standard metric



(a) A and C seem close (original data)

(b) A and C are actually far away (*ISOMAP* embedding)

Figure 3.5: Impact of *ISOMAP* embedding on distances

such as the Euclidean measure. The intuitive rationale for this is that only *short* point-to-point jumps can accurately measure minor changes in the generative process for that point. Therefore, the overall sum of the point-to-point jumps reflects the aggregate change (distance) from one point to another (distant) point more accurately than a straight-line distance between the points. Such distances are referred to as *geodesic distances*. In the case of Fig. 3.4, the only way to walk from A to B with short point-to-point jumps is to walk along the entire elliptical shape of the data distribution while passing C along the way. Therefore, A and B are actually the *farthest* pair of data points (from A, B, and C) on this basis! The implicit assumption is that nonlinear distributions are *locally* Euclidean but are *globally* far from Euclidean.

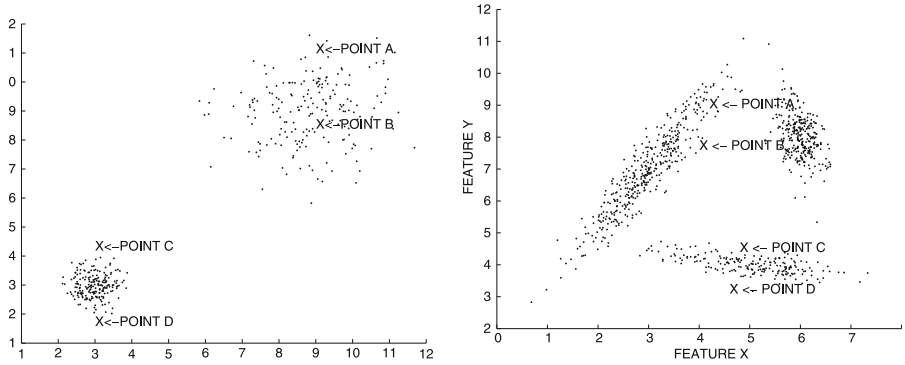
Such distances can be computed by using an approach that is derived from a nonlinear dimensionality reduction and embedding method, known as *ISOMAP*. The approach consists of two steps:

1. Compute the k-nearest neighbors of each point. Construct a weighted graph G with nodes representing data points, and edge weights (costs) representing distances of these k-nearest neighbors.

2. For any pair of points $\overline{X}$ and $\overline{Y}$, report $Dist(\overline{X}, \overline{Y})$ as the shortest path between the corresponding nodes in G.

These two steps are already able to compute the distances without explicitly performing dimensionality reduction. However, an additional step of embedding the data into a multidimensional space makes *repeated* distance computations between many pairs of points much faster, while losing some accuracy. Such an embedding also allows the use of algorithms that work naturally on numeric multidimensional data with predefined distance metrics.

This is achieved by using the all-pairs shortest-path problem to construct the full set of distances between any pair of nodes in G. Subsequently, *multidimensional scaling (MDS)* (cf. Sect. 2.4.4.2 of Chap. 2) is applied to embed the data into a lower dimensional space. The overall effect of the approach is to "straighten out" the nonlinear shape of Fig. 3.4 and embed it into a space where the data are aligned along a flat strip. In fact, a 1-dimensional representation can approximate the data after this transformation. Furthermore, in this new space, a distance function such as the Euclidean metric will work very well as long as metric *MDS* was used in the final phase. A 3-dimensional example is illustrated in Fig. 3.5a, in which the data are arranged along a spiral. In this figure, data points A and C seem much



(a) local density variation (b) local orientation variation

Figure 3.6: Impact of local distributions on distance computations

closer to each other than data point B. However, in the *ISOMAP* embedding of Fig. 3.5b, the data point B is much closer to each of A and C. This example shows the drastic effect of data distributions on distance computation.

In general, high-dimensional data are aligned along nonlinear low-dimensional shapes, which are also referred to as *manifolds*. These manifolds can be "flattened out" to a new representation where metric distances can be used effectively. Thus, this is a data transformation method that facilitates the use of standard metrics. The major computational challenge is in performing the dimensionality reduction. However, after the one-time preprocessing cost has been paid for, repeated distance computations can be implemented efficiently.

Nonlinear embeddings can also be achieved with extensions of *PCA*. *PCA* can be extended to discovering nonlinear embeddings with the use of a method known as the *kernel trick*. Refer to Sect. 10.6.4.1 of Chap. 10 for a brief description of kernel *PCA*.

3.2.1.8 Impact of Local Data Distribution

The discussion so far addresses the impact of global distributions on the distance computations. However, the distribution of the data varies significantly with locality. This variation may be of two types. For example, the absolute density of the data may vary significantly with data locality, or the shape of clusters may vary with locality. The first type of variation is illustrated in Fig. 3.6a, which has two clusters containing the same number of points, but one of them is denser than the other. Even though the absolute distance between (A, B) is identical to that between (C, D), the distance between C and D should be considered greater on the basis of the *local* data distribution. In other words, C and D are much farther away in the *context* of what their local distributions look like. This problem is often encountered in many distance-based methods such as outlier detection. It has been shown that methods that adjust for the local variations in the distances typically perform much better than those that do not adjust for local variations. One of the most well-known methods for outlier detection, known as Local Outlier Factor (*LOF*), is based on this principle.

A second example is illustrated in Fig. 3.6b, which illustrates the impact of varying local orientation of the clusters. Here, the distance between (A, B) is identical to that between (C, D) using the Euclidean metric. However, the local clusters in each region show very different orientation. The high-variance axis of the cluster of data points relevant to (A, B)

is aligned along the path from A to B. This is not true for (C, D). As a result, the intrinsic distance between C and D is much greater than that between A and B. For example, if the *local* Mahalanobis distance is computed using the relevant cluster covariance statistics, then the distance between C and D will evaluate to a larger value than that between A and B.

Shared Nearest-Neighbor Similarity: The first problem can be at least partially alleviated with the use of a shared nearest-neighbor similarity. In this approach, the k-nearest neighbors of each data point are computed in a preprocessing phase. The shared nearest-neighbor similarity is equal to the number of common neighbors between the two data points. This metric is locally sensitive because it depends on the number of common *neighbors*, and not on the absolute values of the distances. In dense regions, the k-nearest neighbor distances will be small, and therefore data points need to be closer together to have a larger number of shared nearest neighbors. Shared nearest-neighbor methods can be used to define a *similarity* graph on the underlying data points in which pairs of data points with at least one shared neighbor have an edge between them. Similarity graph-based methods are almost always locality sensitive because of their local focus on the k-nearest neighbor distribution.

Generic Methods: In generic local distance computation methods, the idea is to divide the space into a set of local regions. The distances are then adjusted in each region using the local statistics of this region. Therefore, the broad approach is as follows:

1. Partition the data into a set of local regions.
2. For any pair of objects, determine the most relevant region for the pair, and compute the pairwise distances using the local statistics of that region. For example, the local Mahalanobis distance may be used in each local region.

A variety of clustering methods are used for partitioning the data into local regions. In cases where each of the objects in the pair belongs to a different region, either the global distribution may be used, or the average may be computed using both local regions. Another problem is that the first step of the algorithm (partitioning process) itself requires a notion of distances for clustering. This makes the solution circular, and calls for an iterative solution. Although a detailed discussion of these methods is beyond the scope of this book, the bibliographic notes at the end of this chapter provide a number of pointers.

3.2.1.9 Computational Considerations

A major consideration in the design of distance functions is the computational complexity. This is because distance function computation is often embedded as a subroutine that is used repeatedly in the application at hand. If the subroutine is not efficiently implementable, the applicability becomes more restricted. For example, methods such as *ISOMAP* are computationally expensive and hard to implement for very large data sets because these methods scale with at least the square of the data size. However, they do have the merit that a one-time transformation can create a representation that can be used efficiently by data mining algorithms. Distance functions are executed repeatedly, whereas the preprocessing is performed only once. Therefore, it is definitely advantageous to use a preprocessing-intensive approach as long as it speeds up later computations. For many applications, sophisticated methods such as *ISOMAP* may be too expensive even for one-time analysis. For such cases, one of the earlier methods discussed in this chapter may need to be used. Among the methods discussed in this section, carefully chosen L_p-norms and match-based techniques are the fastest methods for large-scale applications.

3.2.2 Categorical Data

Distance functions are naturally computed as functions of value differences along dimensions in numeric data, which is *ordered*. However, no ordering exists among the discrete values of categorical data. How can distances be computed? One possibility is to transform the categorical data to numeric data with the use of the binarization approach discussed in Sect. 2.2.2.2 of Chap. 2. Because the binary vector is likely to be sparse (many zero values), similarity functions can be adapted from other sparse domains such as text. For the case of categorical data, it is more common to work with similarity functions rather than distance functions because discrete values can be matched more naturally.

Consider two records $\overline{X} = (x_1 \ldots x_d)$ and $\overline{Y} = (y_1 \ldots y_d)$. The simplest possible similarity between the records $\overline{X}$ and $\overline{Y}$ is the sum of the similarities on the individual attribute values. In other words, if $S(x_i, y_i)$ is the similarity between the attributes values x_i and y_i, then the overall similarity is defined as follows:

$$Sim(\overline{X}, \overline{Y}) = \sum_{i=1}^{d} S(x_i, y_i).$$

Therefore, the choice of $S(x_i, y_i)$ defines the overall similarity function.

The simplest possible choice is to set $S(x_i, y_i)$ to 1 when $x_i = y_i$ and 0 otherwise. This is also referred to as the *overlap* measure. The major drawback of this measure is that it does not account for the relative frequencies among the different attributes. For example, consider a categorical attribute in which the attribute value is "Normal" for 99 % of the records, and either "Cancer" or "Diabetes" for the remaining records. Clearly, if two records have a "Normal" value for this variable, this does not provide statistically significant information about the similarity, because the majority of pairs are likely to show that pattern just by chance. However, if the two records have a matching "Cancer" or "Diabetes" value for this variable, it provides significant statistical evidence of similarity. This argument is similar to that made earlier about the importance of the global data distribution. Similarities or differences that are unusual are statistically more significant than those that are common.

In the context of categorical data, the *aggregate statistical properties* of the data set should be used in computing similarity. This is similar to how the Mahalanobis distance was used to compute similarity more accurately with the use of global statistics. The idea is that matches on unusual values of a categorical attribute should be weighted more heavily than values that appear frequently. This also forms the underlying principle of many common normalization techniques that are used in domains such as text. An example, which is discussed in the next section, is the use of *inverse document frequency (IDF)* in the information retrieval domain. An analogous measure for categorical data will be introduced here.

The *inverse occurrence frequency* is a generalization of the simple matching measure. This measure weights the similarity between the matching attributes of two records by an inverse function of the frequency of the matched value. Thus, when $x_i = y_i$, the similarity $S(x_i, y_i)$ is equal to the inverse weighted frequency, and 0 otherwise. Let $p_k(x)$ be the fraction of records in which the kth attribute takes on the value of x in the data set. In other words, when $x_i = y_i$, the value of $S(x_i, y_i)$ is $1/p_k(x_i)^2$ and 0 otherwise.

$$S(x_i, y_i) = \begin{cases} 1/p_k(x_i)^2 & \text{if } x_i = y_i \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

A related measure is the *Goodall* measure. As in the case of the inverse occurrence frequency, a higher similarity value is assigned to a match when the value is infrequent. In a simple variant of this measure [104], the similarity on the kth attribute is defined as $1 - p_k(x_i)^2$, when $x_i = y_i$, and 0 otherwise.

$$S(x_i, y_i) = \begin{cases} 1 - p_k(x_i)^2 & \text{if } x_i = y_i \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

The bibliographic notes contain pointers to various similarity measures for categorical data.

3.2.3 Mixed Quantitative and Categorical Data

It is fairly straightforward to generalize the approach to mixed data by adding the weights of the numeric and quantitative components. The main challenge is in deciding how to assign the weights of the quantitative and categorical components. For example, consider two records $\overline{X} = (\overline{X_n}, \overline{X_c})$ and $\overline{Y} = (\overline{Y_n}, \overline{Y_c})$ where $\overline{X_n}$, $\overline{Y_n}$ are the subsets of numerical attributes and $\overline{X_c}$, $\overline{Y_c}$ are the subsets of categorical attributes. Then, the overall similarity between $\overline{X}$ and $\overline{Y}$ is defined as follows:

$$Sim(\overline{X}, \overline{Y}) = \lambda \cdot NumSim(\overline{X_n}, \overline{Y_n}) + (1 - \lambda) \cdot CatSim(\overline{X_c}, \overline{Y_c}). \tag{3.8}$$

The parameter λ regulates the relative importance of the categorical and numerical attributes. The choice of λ is a difficult one. In the absence of domain knowledge about the relative importance of attributes, a natural choice is to use a value of λ that is equal to the fraction of numerical attributes in the data. Furthermore, the proximity in numerical data is often computed with the use of distance functions rather than similarity functions. However, distance values can be converted to similarity values as well. For a distance value of $dist$, a common approach is to use a kernel mapping that yields [104] the similarity value of $1/(1 + dist)$.

Further normalization is required to meaningfully compare the similarity value components on the numerical and categorical attributes that may be on completely different scales. One way of achieving this goal is to determine the standard deviations in the similarity values over the two domains with the use of sample pairs of records. Each component of the similarity value (numerical or categorical) is divided by its standard deviation. Therefore, if σ_c and σ_n are the standard deviations of the similarity values in the categorical and numerical components, then Eq. 3.8 needs to be modified as follows:

$$Sim(\overline{X}, \overline{Y}) = \lambda \cdot NumSim(\overline{X_n}, \overline{Y_n})/\sigma_n + (1 - \lambda) \cdot CatSim(\overline{X_c}, \overline{Y_c})/\sigma_c. \tag{3.9}$$

By performing this normalization, the value of λ becomes more meaningful, as a true *relative weight* between the two components. By default, this weight can be set to be proportional to the number of attributes in each component unless specific domain knowledge is available about the relative importance of attributes.

3.3 Text Similarity Measures

Strictly speaking, text can be considered quantitative multidimensional data when it is treated as a bag of words. The frequency of each word can be treated as a quantitative attribute, and the base lexicon can be treated as the full set of attributes. However, the

structure of text is *sparse* in which most attributes take on 0 values. Furthermore, all word frequencies are nonnegative. This special structure of text has important implications for similarity computation and other mining algorithms. Measures such as the L_p-norm do not adjust well to the varying length of the different documents in the collection. For example, the L_2-distance between two long documents will almost always be larger than that between two short documents even if the two long documents have many words in common, and the short documents are completely disjoint. How can one normalize for such irregularities? One way of doing so is by using the cosine measure. The cosine measure computes the *angle* between the two documents, which is insensitive to the absolute length of the document. Let $\overline{X} = (x_1 \ldots x_d)$ and $\overline{Y} = (y_1 \ldots y_d)$ be two documents on a lexicon of size d. Then, the cosine measure $\cos(\overline{X}, \overline{Y})$ between $\overline{X}$ and $\overline{Y}$ can be defined as follows:

$$\cos(\overline{X}, \overline{Y}) = \frac{\sum_{i=1}^{d} x_i \cdot y_i}{\sqrt{\sum_{i=1}^{d} x_i^2} \cdot \sqrt{\sum_{i=1}^{d} y_i^2}}. \tag{3.10}$$

The aforementioned measure simply uses the raw frequencies between attributes. However, as in other data types, it is possible to use global statistical measures to improve the similarity computation. For example, if two documents match on an uncommon word, it is more indicative of similarity than the case where two documents match on a word that occurs very commonly. The *inverse document frequency* id_i, which is a decreasing function of the number of documents n_i in which the ith word occurs, is commonly used for normalization:

$$id_i = \log(n/n_i). \tag{3.11}$$

Here, the number of documents in the collection is denoted by n. Another common adjustment is to ensure that the excessive presence of single word does not throw off the similarity measure. A damping function $f(\cdot)$, such as the square root or the logarithm, is optionally applied to the frequencies before similarity computation.

$$f(x_i) = \sqrt{x_i}$$
$$f(x_i) = \log(x_i)$$

In many cases, the damping function is not used, which is equivalent to setting $f(x_i)$ to x_i. Therefore, the *normalized* frequency $h(x_i)$ for the ith word may be defined as follows:

$$h(x_i) = f(x_i) \cdot id_i. \tag{3.12}$$

Then, the cosine measure is defined as in Eq. 3.10, except that the normalized frequencies of the words are used:

$$\cos(\overline{X}, \overline{Y}) = \frac{\sum_{i=1}^{d} h(x_i) \cdot h(y_i)}{\sqrt{\sum_{i=1}^{d} h(x_i)^2} \cdot \sqrt{\sum_{i=1}^{d} h(y_i)^2}}. \tag{3.13}$$

Another measure that is less commonly used for text is the *Jaccard coefficient* $J(\overline{X}, \overline{Y})$:

$$J(\overline{X}, \overline{Y}) = \frac{\sum_{i=1}^{d} h(x_i) \cdot h(y_i)}{\sum_{i=1}^{d} h(x_i)^2 + \sum_{i=1}^{d} h(y_i)^2 - \sum_{i=1}^{d} h(x_i) \cdot h(y_i)}. \tag{3.14}$$

The Jaccard coefficient is rarely used for the text domain, but it is used commonly for sparse binary data sets.

3.3.1 Binary and Set Data

Binary multidimensional data are a representation of set-based data, where a value of 1 indicates the presence of an element in a set. Binary data occur commonly in market-basket domains in which transactions contain information corresponding to whether or not an item is present in a transaction. It can be considered a special case of text data in which word frequencies are either 0 or 1. If S_X and S_Y are two sets with binary representations $\overline{X}$ and $\overline{Y}$, then it can be shown that applying Eq. 3.14 to the raw binary representation of the two sets is equivalent to:

$$J(\overline{X}, \overline{Y}) = \frac{\sum_{i=1}^{d} x_i \cdot y_i}{\sum_{i=1}^{d} x_i^2 + \sum_{i=1}^{d} y_i^2 - \sum_{i=1}^{d} x_i \cdot y_i} = \frac{|S_X \cap S_Y|}{|S_X \cup S_Y|}. \tag{3.15}$$

This is a particularly intuitive measure because it carefully accounts for the number of common and disjoint elements in the two sets.

3.4 Temporal Similarity Measures

Temporal data contain a single contextual attribute representing time and one or more behavioral attributes that measure the properties varying along a particular time period. Temporal data may be represented as continuous time series, or as discrete sequences, depending on the application domain. The latter representation may be viewed as the discrete version of the former. It should be pointed out that discrete sequence data are not always temporal because the contextual attribute may represent placement. This is typically the case in biological sequence data. Discrete sequences are also sometimes referred to as *strings*. Many of the similarity measures used for time series and discrete sequences can be reused across either domain, though some of the measures are more suited to one of the domains. Therefore, this section will address both data types, and each similarity measure will be discussed in a subsection on either continuous series or discrete series, based on its most common use. For some measures, the usage is common across both data types.

3.4.1 Time-Series Similarity Measures

The design of time-series similarity measures is highly application specific. For example, the simplest possible similarity measure between two time series of equal length is the Euclidean metric. Although such a metric may work well in many scenarios, it does not account for several distortion factors that are common in many applications. Some of these factors are as follows:

1. *Behavioral attribute scaling and translation:* In many applications, the different time series may not be drawn on the same scales. For example, the time series representing various stocks prices may show similar patterns of movements, but the absolute values may be very different both in terms of the mean and the standard deviation. For example, the share prices of several different hypothetical stock tickers are illustrated in Fig. 3.7. All three series show similar patterns but with different scaling and some random variations. Clearly, they show similar patterns but cannot be meaningfully compared if the absolute values of the series are used.

2. *Temporal (contextual) attribute translation:* In some applications, such as real-time analysis of financial markets, the different time series may represent the same periods

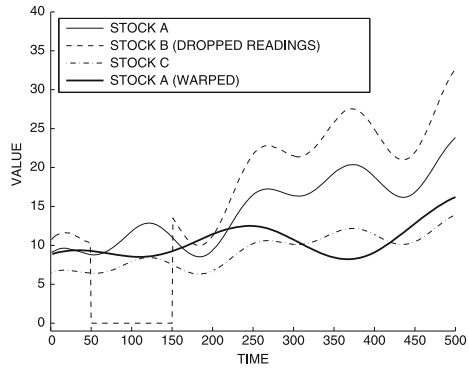


Figure 3.7: Impact of scaling, translation, and noise

in time. In other applications, such as the analysis of the time series obtained from medical measurements, the absolute time stamp of when the reading was taken is not important. In such cases, the temporal attribute value needs to be shifted in at least one of the time series to allow more effective matching.

3. *Temporal (contextual) attribute scaling:* In this case, the series may need to be stretched or compressed along the temporal axis to allow more effective matching. This is referred to as *time warping*. An additional complication is that different temporal segments of the series may need to be warped differently to allow for better matching. In Fig. 3.7, the simplest case of warping is shown where the entire set of values for stock A has been stretched. In general, the time warping can be more complex where different windows in the same series may be stretched or compressed differently. This is referred to as *dynamic* time warping (DTW).

4. *Noncontiguity in matching:* Long time series may have noisy segments that do not match very well with one another. For example, one of the series in Fig. 3.7 has a window of dropped readings because of data collection limitations. This is common in sensor data. The distance function may need to be robust to such noise.

Some of these issues can be addressed by attribute normalization during preprocessing.

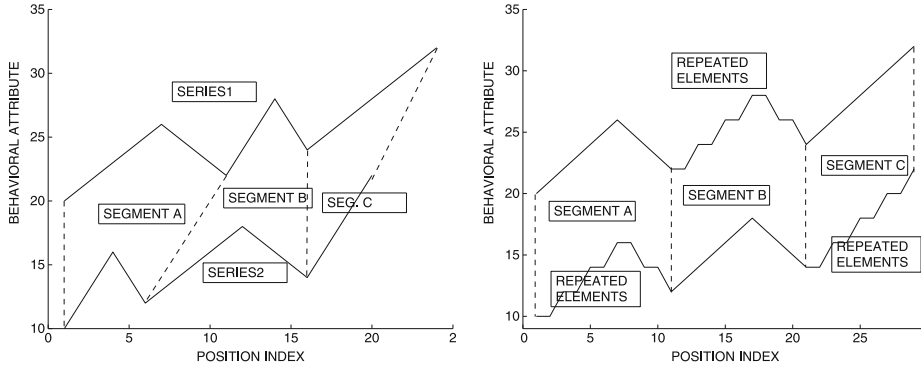
3.4.1.1 Impact of Behavioral Attribute Normalization

The translation and scaling issues are often easier to address for the behavioral attributes as compared to contextual attributes, because they can be addressed by normalization during preprocessing:

1. *Behavioral attribute translation:* The behavioral attribute is mean centered during preprocessing.

2. *Behavioral attribute scaling:* The standard deviation of the behavioral attribute is scaled to 1 unit.

It is important to remember that these normalization issues may not be relevant to every application. Some applications may require only translation, only scaling, or neither of the two. Other applications may require both. In fact, in some cases, the wrong choice of



(a) Original series (b) Warped series

Figure 3.8: Illustration of dynamic time warping by repeating elements

normalization may have detrimental effects on the interpretability of the results. Therefore, an analyst needs to judiciously select a normalization approach depending on application-specific needs.

3.4.1.2 L_p-Norm

The L_p-norm may be defined for two series $\overline{X} = (x_1 \ldots x_n)$ and $\overline{Y} = (y_1 \ldots y_n)$. This measure treats a time series as a multidimensional data point in which each time stamp is a dimension.

$$Dist(\overline{X}, \overline{Y}) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p} \tag{3.16}$$

The L_p-norm can also be applied to wavelet transformations of the time series. In the special case where $p = 2$, accurate distance computations are obtained with the wavelet representation, if most of the larger wavelet coefficients are retained in the representation. In fact, it can be shown that if no wavelet coefficients are removed, then the distances are identical between the two representations. This is because wavelet transformations can be viewed as a rotation of an axis system in which each dimension represents a time stamp. Euclidean metrics are invariant to axis rotation. The major problem with L_p-norms is that they are designed for time series of equal length and cannot address distortions on the temporal (contextual) attributes.

3.4.1.3 Dynamic Time Warping Distance

DTW stretches the series along the time axis in a varying (or *dynamic*) way over different portions to enable more effective matching. An example of warping is illustrated in Fig. 3.8a, where the two series have very similar shape in segments A, B, and C, but specific segments in each series need to be stretched appropriately to enable better matching. The DTW measure has been adapted from the field of speech recognition, where time warping was deemed necessary to match different speaking speeds. DTW can be used either for time-series or sequence data, because it addresses only the issue of contextual attribute scaling, and it is unrelated to the nature of the behavioral attribute. The following description is a generic one, which can be used either for time-series or sequence data.

The L_p-metric can only be defined between two time series of equal length. However, DTW, by its very nature, allows the measurement of distances between two series of *different* lengths. In the L_p distance, a one-to-one mapping exists between the time stamps of the two time series. However, in DTW, a many-to-one mapping is allowed to account for the time warping. This many-to-one mapping can be thought of in terms of repeating some of the elements in carefully chosen segments of either of the two time series. This can be used to artificially create two series of the same length that have a one-to-one mapping between them. The distances can be measured on the resulting warped series using any distance measure such as the L_p-norm. For example, in Fig. 3.8b, some elements in a few segments of either series are repeated to create a one-to-one mapping between the two series. Note that the two series now look much more similar than the two series in Fig. 3.8a. Of course, this repeating can be done in many different ways, and the goal is to perform it in an *optimal* way to minimize the DTW distance. The optimal choice of warping is determined using dynamic programming.

To understand how DTW generalizes a one-to-one distance metric such as the L_p-norm, consider the L_1 (Manhattan) metric $M(\overline{X_i}, \overline{Y_i})$, computed on the first i elements of two time series $\overline{X} = (x_1 \ldots x_n)$ and $\overline{Y} = (y_1 \ldots y_n)$ of equal length. The value of $M(\overline{X_i}, \overline{Y_i})$ can be written *recursively* as follows:

$$M(\overline{X_i}, \overline{Y_i}) = |x_i - y_i| + M(\overline{X_{i-1}}, \overline{Y_{i-1}}). \tag{3.17}$$

Note that the indices of *both* series are reduced by 1 in the right-hand side because of the one-to-one matching. In DTW, both indices need not reduce by 1 unit because a many-to-one mapping is allowed. Rather, any one or both indices may reduce by 1, depending on the *best match* between the two time series (or sequences). The index that did *not* reduce by 1 corresponds to the repeated element. The choice of index reduction is naturally defined, recursively, as an optimization over the various options.

Let $DTW(i, j)$ be the optimal distance between the first i and first j elements of two time series $\overline{X} = (x_1 \ldots x_m)$ and $\overline{Y} = (y_1 \ldots y_n)$, respectively. Note that the two time series are of lengths m and n, which may not be the same. Then, the value of $DTW(i, j)$ is defined recursively as follows:

$$DTW(i,j) = distance(x_i, y_j) + \min \begin{cases} DTW(i, j-1) & \text{repeat } x_i \\ DTW(i-1, j) & \text{repeat } y_j \\ DTW(i-1, j-1) & \text{repeat neither} \end{cases}. \tag{3.18}$$

The value of $distance(x_i, y_j)$ may be defined in a variety of ways, depending on the application domain. For example, for continuous time series, it may be defined as $|x_i - y_i|^p$, or by a distance that accounts for (behavioral attribute) scaling and translation. For discrete sequences, it may be defined using a categorical measure. The DTW approach is primarily focused on warping the *contextual* attribute, and has little to do with the nature of the behavioral attribute or distance function. Because of this fact, time warping can easily be extended to multiple behavioral attributes by simply using the distances along multiple attributes in the recursion.

Equation 3.18 yields a natural iterative approach. The approach starts by initializing $DTW(0, 0)$ to 0, $DTW(0, j)$ to ∞ for $j \in \{1 \ldots n\}$, and $DTW(i, 0)$ to ∞ for $i \in \{1 \ldots m\}$. The algorithm computes $DTW(i, j)$ by repeatedly executing Eq. 3.18 with increasing index values of i and j. This can be achieved by a simple nested loop in which the indices i and j increase from 1 to m and 1 to n, respectively:

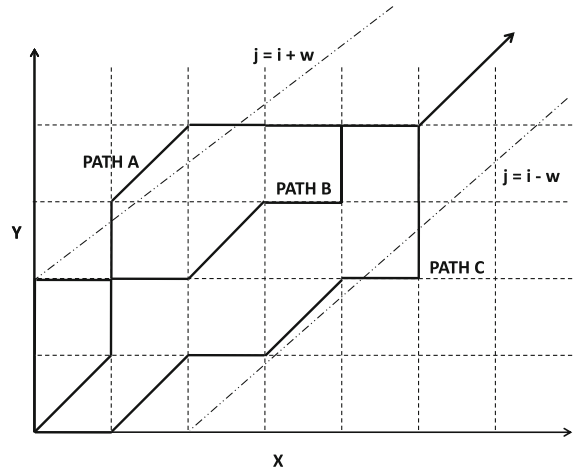


Figure 3.9: Illustration of warping paths

for $i = 1$ to m
  for $j = 1$ to n
    compute $DTW(i, j)$ using Eq. 3.18

The aforementioned code snippet is a nonrecursive and iterative approach. It is also possible to implement a recursive computer program by directly using Eq. 3.18. Therefore, the approach requires the computation of all values of $DTW(i, j)$ for every $i \in [1, m]$ and every $j \in [1, n]$. This is a $m \times n$ grid of values, and therefore the approach may require $O(m \cdot n)$ iterations, where m and n are lengths of the series.

The optimal warping can be understood as an *optimal path* through different values of i and j in the $m \times n$ grid of values, as illustrated in Fig. 3.9. Three possible paths, denoted by A, B, and C, are shown in the figure. These paths only move to the right (increasing i and repeating y_j), upward (increasing j and repeating x_i), or both (repeating neither).

A number of practical constraints are often added to the DTW computation. One commonly used constraint is the *window constraint* that imposes a minimum level w of positional alignment between matched elements. The window constraint requires that $DTW(i, j)$ be computed only when $|i - j| \leq w$. Otherwise, the value may be set to ∞ by default. For example, the paths B and C in Fig. 3.9 no longer need to be computed. This saves the computation of many values in the dynamic programming recursion. Correspondingly, the computations in the inner variable j of the nested loop above can be saved by constraining the index j, so that it is never more than w units apart from the outer loop variable i. Therefore, the inner loop index j is varied from $\max\{0, i - w\}$ to $\min\{n, i + w\}$.

The DTW distance can be extended to multiple behavioral attributes easily, if it is assumed that the different behavioral attributes have the same time warping. In this case, the recursion is unchanged, and the only difference is that $distance(\overline{x_i}, \overline{y_j})$ is computed using a vector-based distance measure. We have used a bar on $\overline{x_i}$ and $\overline{y_j}$ to denote that these are vectors of multiple behavioral attributes. This multivariate extension is discussed in Sect. 16.3.4.1 of Chap. 16 for measuring distances between 2-dimensional trajectories.

3.4.1.4 Window-Based Methods

The example in Fig. 3.7 illustrates a case where dropped readings may cause a gap in the matching. Window-based schemes attempt to decompose the two series into windows and then "stitch" together the similarity measure. The intuition here is that if two series have many contiguous matching segments, they should be considered similar. For long time series, a global match becomes increasingly unlikely. The only reasonable choice is the use of windows for measurement of segment-wise similarity.

Consider two time series $\overline{X}$ and $\overline{Y}$, and let $\overline{X_1} \ldots \overline{X_r}$ and $\overline{Y_1} \ldots \overline{Y_r}$ be temporally ordered and nonoverlapping windows extracted from the respective series. Note that some windows from the base series may not be included in these segments at all. These correspond to the noise segments that are dropped. Then, the overall similarity between $\overline{X}$ and $\overline{Y}$ can be computed as follows:

$$Sim(\overline{X}, \overline{Y}) = \sum_{i=1}^{r} Match(\overline{X_i}, \overline{Y_i}). \tag{3.19}$$

A variety of measures discussed in this section may be used to instantiate the value of $Match(\overline{X_i}, \overline{Y_i})$. It is tricky to determine the proper value of $Match(\overline{X_i}, \overline{Y_i})$ because a contiguous match along a long window is more unusual than many short segments of the same length. The proper choice of $Match(\overline{X_i}, \overline{Y_i})$ may depend on the application at hand. Another problem is that the optimal decomposition of the series into windows may be a difficult task. These methods are not discussed in detail here, but the interested reader is referred to the bibliographic notes for pointers to relevant methods.

3.4.2 Discrete Sequence Similarity Measures

Discrete sequence similarity measures are based on the same general principles as time-series similarity measures. As in the case of time-series data, discrete sequence data may or may not have a one-to-one mapping between the positions. When a one-to-one mapping does exist, many of the multidimensional categorical distance measures can be adapted to this domain, just as the L_p-norm can be adapted to continuous time series. However, the application domains of discrete sequence data are most often such that a one-to-one mapping does not exist. Aside from the DTW approach, a number of other dynamic programming methods are commonly used.

3.4.2.1 Edit Distance

The edit distance defines the distance between two strings as the *least* amount of "effort" (or *cost*) required to transform one sequence into another by using a series of transformation operations, referred to as "edits." The edit distance is also referred to as the *Levenshtein* distance. The edit operations include the use of symbol insertions, deletions, and replacements with specific costs. In many models, replacements are assumed to have higher cost than insertions or deletions, though insertions and deletions are usually assumed to have the same cost. Consider the sequences *abababab* and *bababababa*, which are drawn on the alphabet $\{a, b\}$. The first string can be transformed to the second in several ways. For example, if every alphabet in the first string was replaced by the other alphabet, it would result in the second string. The cost of doing so is that of ten replacements. However, a more cost-efficient way of achieving the same goal is to delete the leftmost element of the string, and insert the symbol "a" as the rightmost element. The cost of this sequence of operations is only one insertion and one deletion. The edit distance is defined as the optimal cost to

transform one string to another with a sequence of insertions, deletions, and replacements. The computation of the optimal cost requires a dynamic programming recursion.

For two sequences $\overline{X} = (x_1 \ldots x_m)$ and $\overline{Y} = (y_1 \ldots y_n)$, let the edits be performed on sequence $\overline{X}$ to transform to $\overline{Y}$. Note that this distance function is asymmetric because of the directionality to the edit. For example, $Edit(\overline{X}, \overline{Y})$ may not be the same as $Edit(\overline{Y}, \overline{X})$ if the insertion and deletion costs are not identical. In practice, however, the insertion and deletion costs are assumed to be the same.

Let I_{ij} be a binary indicator that is 0 when the ith symbol of $\overline{X}$ and jth symbols of $\overline{Y}$ are the same. Otherwise, the value of this indicator is 1. Then, consider the first i symbols of $\overline{X}$ and the first j symbols of $\overline{Y}$. Assume that these segments are represented by $\overline{X_i}$ and $\overline{Y_j}$, respectively. Let $Edit(i, j)$ represent the optimal matching cost between these segments. The goal is to determine what operation to perform on the last element of $\overline{X_i}$ so that it either matches an element in $\overline{Y_j}$, or it is deleted. Three possibilities arise:

1. The last element of $\overline{X_i}$ is deleted, and the cost of this is $[Edit(i-1, j)+\text{Deletion Cost}]$. The last element of the truncated segment $\overline{X_{i-1}}$ may or may not match the last element of $\overline{Y_j}$ at this point.

2. An element is inserted at the end of $\overline{X_i}$ to match the last element of Y_j, and the cost of this is $[Edit(i, j-1) + \text{Insertion Cost}]$. The indices of the edit term $Edit(i, j-1)$ reflect the fact that the matched elements of both series can now be removed.

3. The last element of $\overline{X_i}$ is flipped to that of $\overline{Y_j}$ *if it is different*, and the cost of this is $[Edit(i-1, j-1) + I_{ij} \cdot (\text{Replacement Cost})]$. In cases where the last elements are the same, the additional replacement cost is not incurred, but progress is nevertheless made in matching. This is because the matched elements (x_i, y_j) of both series need not be considered further, and residual matching cost is $Edit(i-1, j-1)$.

Clearly, it is desirable to pick the minimum of these costs for the optimal matching. Therefore, the optimal matching is defined by the following recursion:

$$Edit(i, j) = \min \begin{cases} Edit(i-1, j) + \text{Deletion Cost} \\ Edit(i, j-1) + \text{Insertion Cost} \\ Edit(i-1, j-1) + I_{ij} \cdot (\text{Replacement Cost}) \end{cases} . \tag{3.20}$$

Furthermore, the bottom of the recursion also needs to be set up. The value of $Edit(i, 0)$ is equal to the cost of i deletions for any value of i, and that of $Edit(0, j)$ is equal to the cost of j insertions for any value of j. This nicely sets up the dynamic programming approach. It is possible to write the corresponding computer program either as a nonrecursive nested loop (as in DTW) or as a recursive computer program that directly uses the aforementioned cases.

The aforementioned discussion assumes general insertion, deletion, and replacement costs. In practice, however, the insertion and deletion costs are usually assumed to be the same. In such a case, the edit function is symmetric because it does not matter which of the two strings is edited to the other. For any sequence of edits from one string to the other, a reverse sequence of edits, with the same cost, will exist from the other string to the first.

The edit distance can be extended to numeric data by changing the primitive operations of *insert*, *delete*, and *replace* to transformation rules that are designed for time series. Such transformation rules can include making basic changes to the shape of the time series in

window segments. This is more complex because it requires one to design the base set of allowed time-series shape transformations and their costs. Such an approach has not found much popularity for time-series distance computation.

3.4.2.2 Longest Common Subsequence

A *subsequence* of a sequence is a set of symbols drawn from the sequence in the same order as the original sequence. A subsequence is different from a *substring* in that the values of the subsequence need not be contiguous, whereas the values in the substring need to be contiguous. Consider the sequences $agbfcgdhei$ and $afbgchdiei$. In this case, ei is a substring of both sequences and also a subsequence. However, $abcde$ and fgi are subsequences of both strings but not substrings. Clearly, subsequences of longer length are indicative of a greater level of matching between the strings. Unlike the edit distance, the longest common subsequence ($LCSS$) is a similarity function because higher values indicate greater similarity. The number of possible subsequences is exponentially related to the length of a string. However, the LCSS can be computed in polynomial time with a dynamic programming approach.

For two sequences $\overline{X} = (x_1 \ldots x_m)$ and $\overline{Y} = (y_1 \ldots y_n)$, consider the first i symbols of $\overline{X}$ and the first j symbols of $\overline{Y}$. Assume that these segments are represented by $\overline{X_i}$ and $\overline{Y_j}$, respectively. Let $LCSS(i, j)$ represent the optimal LCSS values between these segments. The goal here is to either match the last element of $\overline{X_i}$ and $\overline{Y_j}$, or delete the last element in one of the two sequences. Two possibilities arise:

1. The last element of $\overline{X_i}$ matches $\overline{Y_j}$, in which case, it cannot hurt to instantiate the matching on the last element and then delete the last element of both sequences. The similarity value $LCSS(i, j)$ can be expressed recursively as this is $LCSS(i-1, j-1)+1$.

2. The last element does not match. In such a case, the last element of at least one of the two strings needs to be deleted under the assumption that it cannot occur in the matching. In this case, the value of $LCSS(i, j)$ is either $LCSS(i, j-1)$ or $LCSS(i-1, j)$, depending on which string is selected for deletion.

Therefore, the optimal matching can be expressed by enumerating these cases:

$$LCSS(i,j) = \max \begin{cases} LCSS(i-1,j-1)+1 & \text{only if } x_i = y_j \\ LCSS(i-1,j) & \text{otherwise (no match on } x_i) \\ LCSS(i,j-1) & \text{otherwise (no match on } y_j) \end{cases}. \tag{3.21}$$

Furthermore, the boundary conditions need to be set up. The values of $LCSS(i, 0)$ and $LCSS(0, j)$ are always equal to 0 for any value of i and j. As in the case of the DTW and edit-distance computations, a nested loop can be set up to compute the final value. A recursive computer program can also be implemented that uses the aforementioned recursive relationship. Although the $LCSS$ approach is defined for a discrete sequence, it can also be applied to a continuous time series after discretizing the time-series values into a sequence of categorical values. Alternatively, one can discretize the time-series *movement* between two contiguous time stamps. The particular choice of discretization depends on the goals of the application at hand.

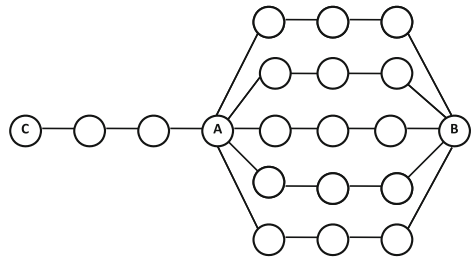


Figure 3.10: Shortest path versus homophily

3.5 Graph Similarity Measures

The similarity in graphs can be measured in different ways, depending on whether the similarity is being measured between two graphs, or between two nodes in a single graph. For simplicity, undirected networks are assumed, though the measures can be easily generalized to directed networks.

3.5.1 Similarity between Two Nodes in a Single Graph

Let $G = (N, A)$ be an undirected network with node set N and edge set A. In some domains, costs are associated with nodes, whereas in others, weights are associated with nodes. For example, in domains such as bibliographic networks, the edges are naturally weighted, and in road networks, the edges naturally have costs. Typically, *distance* functions work with costs, whereas *similarity* functions work with weights. Therefore, it may be assumed that either the cost c_{ij}, or the weight w_{ij} of the edge (i, j) is specified. It is often possible to convert costs into weights (and vice versa) using simple heuristic kernel functions that are chosen in an application-specific way. An example is the heat kernel $K(x) = e^{-x^2/t^2}$.

It is desired to measure the similarity between any pair of nodes i and j. The principle of similarity between two nodes in a single graph is based on the concept of *homophily* in real networks. The principle of homophily is that nodes are typically more similar in a network when they are connected to one another with edges. This is common in many domains such as the Web and social networks. Therefore, nodes that are connected via *short paths* and *many paths* should be considered more similar. The latter criterion is closely related to the concept of connectivity between nodes. The first criterion is relatively easy to implement with the use of the shortest-path algorithm in networks.

3.5.1.1 Structural Distance-Based Measure

The goal here is to measure the distances from any source node s to any other node in the network. Let $SP(s, j)$ be the shortest-path distance from source node s to any node j. The value of $SP(s, j)$ is initialized to 0 for $j = s$ and ∞ otherwise. Then, the distance computation of s to all other nodes in the network may be summarized in a single step that is performed exactly once for each node in the network in a certain order:

- Among all nodes not examined so far, select the node i with the smallest value of $SP(s, i)$ and update the distance labels of each of its neighbors j as follows:

$$SP(s, j) = \min\{SP(s, j), SP(s, i) + c_{ij}\}. \tag{3.22}$$

This is the essence of the well-known Dijkstra algorithm. This approach is linear in the number of edges in the network, because it examines each node and its incident edges exactly once. The approach provides the distances from a single node to all other nodes in a single pass. The final value of $SP(s, j)$ provides a quantification of the structural distance between node s and node j. Structural distance-based measures do not leverage the multiplicity in paths between a pair of nodes because they focus only on the raw structural distances.

3.5.1.2 Random Walk-Based Similarity

The structural measure of the previous section does not work well when pairs of nodes have varying numbers of paths between them. For example, in Fig. 3.10, the shortest-path length between nodes A and B is 4, whereas that between A and C is 3. Yet, node B should be considered more similar to A because the two nodes are more tightly connected with a *multiplicity* of paths. The idea of random walk-based similarity is based on this principle.

In random walk-based similarity, the approach is as follows: Imagine a random walk that starts at source node s, and proceeds to an adjacent node with weighted probability proportional to w_{ij}. Furthermore, at any given node, it is allowed to "jump back" to the source node s with a probability referred to as the *restart probability*. This will result in a probability distribution that is heavily biased toward the source node s. Nodes that are more similar to s will have higher probability of visits. Such an approach will adjust very well to the scenario illustrated in Fig. 3.10 because the walk will visit B more frequently.

The intuition here is the following: If you were lost in a road network and drove randomly, while taking turns randomly, which location are you more *likely* to reach? You are more likely to reach a location that is close by *and* can be reached in multiple ways. The random-walk measure therefore provides a result that is different from that of the shortest-path measure because it also accounts for multiplicity in paths during similarity computation.

This similarity computation is closely related to concept of *PageRank*, which is used to rank pages on the Web by search engines. The corresponding modification for measuring similarity between nodes is also referred to as *personalized PageRank*, and a symmetric variant is referred to as *SimRank*. This chapter will not discuss the details of *PageRank* and *SimRank* computation, because it requires more background on the notion of ranking. Refer to Sect. 18.4 of Chap. 18, which provides a more complete discussion.

3.5.2 Similarity Between Two Graphs

In many applications, multiple graphs are available, and it is sometimes necessary to determine the distances between multiple graphs. A complicating factor in similarity computation is that many nodes may have the same label, which makes them indistinguishable. Such cases arise often in domains such as chemical compound analysis. Chemical compounds can be represented as graphs where nodes are elements, and bonds are edges. Because an element may be repeated in a molecule, the labels on the nodes are not distinct. Determining a similarity measure on graphs is extremely challenging in this scenario, because even the very special case of determining whether the two graphs are identical is hard. The latter problem is referred to as the *graph isomorphism* problem, and is known to the NP-hard [221]. Numerous measures, such as the graph-edit distance and substructure-based similarity, have been proposed to address this very difficult case. The core idea in each of these methods is as follows:

1. *Maximum common subgraph distance:* When two graphs contain a large subgraph in common, they are generally considered more similar. The maximum common subgraph problem and the related distance functions are addressed in Sect. 17.2 of Chap. 17.

2. *Substructure-based similarity:* Although it is difficult to match two large graphs, it is much easier to match smaller substructures. The core idea is to count the frequently occurring substructures between the two graphs and report it as a similarity measure. This can be considered the graph analog of subsequence-based similarity in strings. Substructure-based similarity measures are discussed in detail in Sect. 17.3 of Chap. 17.

3. *Graph-edit distance:* This distance measure is analogous to the string-edit distance and is defined as the number of edits required to transform one graph to the other. Because graph matching is a hard problem, this measure is difficult to implement for large graphs. The graph-edit distance is discussed in detail in Sect. 17.2.3.2 of Chap. 17.

4. *Graph kernels:* Numerous kernel functions have been defined to measure similarity between graphs, such as the shortest-path kernel and the random-walk kernel. This topic is discussed in detail in Sect. 17.3.3 of Chap. 17.

These methods are quite complex and require a greater background in the area of graphs. Therefore, the discussion of these measures is deferred to Chap. 17 of this book.

3.6 Supervised Similarity Functions

The previous sections discussed similarity measures that do not require any understanding of user intentions. In practice, the relevance of a feature or the choice of distance function heavily depends on the domain at hand. For example, for an image data set, should the color feature or the texture feature be weighted more heavily? These aspects cannot be modeled by a distance function without taking the user intentions into account. Unsupervised measures, such as the L_p-norm, treat all features equally, and have little intrinsic understanding of the end user's *semantic* notion of similarity. The only way to incorporate this information into the similarity function is to use explicit feedback about the similarity and dissimilarity of objects. For example, the feedback can be expressed as the following sets of object pairs:

$$\mathcal{S} = \{(O_i, O_j) : O_i \text{ is similar to } O_j\}$$
$$\mathcal{D} = \{(O_i, O_j) : O_i \text{ is dissimilar to } O_j\}.$$

How can this information be leveraged to improve the computation of similarity? Many specialized methods have been designed for supervised similarity computation. A common approach is to assume a specific closed form of the similarity function for which the parameters need to be learned. An example is the weighted L_p-norm in Sect. 3.2.1.1, where the parameters represented by Θ correspond to the feature weights $(a_1 \ldots a_d)$. Therefore, the first step is to create a distance function $f(O_i, O_j, \Theta)$, where Θ is a set of unknown weights. Assume that higher values of the function indicate greater dissimilarity. Therefore, this is a distance function, rather than a similarity function. Then, it is desirable to determine the

parameters Θ, so that the following conditions are satisfied as closely as possible:

$$f(O_i, O_j, \Theta) = \begin{cases} 0 & \text{if } (O_i, O_j) \in \mathcal{S} \\ 1 & \text{if } (O_i, O_j) \in \mathcal{D} \end{cases}. \tag{3.23}$$

This can be expressed as a least squares optimization problem over Θ, with the following error E:

$$E = \sum_{(O_i, O_j) \in \mathcal{S}} (f(O_i, O_j, \Theta) - 0)^2 + \sum_{(O_i, O_j) \in \mathcal{D}} (f(O_i, O_j, \Theta) - 1)^2. \tag{3.24}$$

This objective function can be optimized with respect to Θ with the use of any off-the-shelf optimization solver. If desired, the additional constraint $\Theta \geq 0$ can be added where appropriate. For example, when Θ represents the feature weights $(a_1 \ldots a_d)$ in the Minkowski metric, it is natural to make the assumption of nonnegativity of the coefficients. Such a constrained optimization problem can be easily solved using many nonlinear optimization methods. The use of a closed form such as $f(O_i, O_j, \Theta)$ ensures that the function $f(O_i, O_j, \Theta)$ can be computed efficiently after the one-time cost of computing the parameters Θ.

Where possible, user feedback should be used to improve the quality of the distance function. The problem of learning distance functions can be modeled more generally as that of classification. The classification problem will be studied in detail in Chaps. 10 and 11. Supervised distance function design with the use of Fisher's method is also discussed in detail in the section on instance-based learning in Chap. 10.

3.7 Summary

The problem of distance function design is a crucial one in the context of data mining applications. This is because many data mining algorithms use the distance function as a key subroutine, and the design of the function directly impacts the quality of the results. Distance functions are highly sensitive to the type of the data, the dimensionality of the data, and the global and local nature of the data distribution.

The L_p-norm is the most common distance function used for multidimensional data. This distance function does not seem to work well with increasing dimensionality. Higher values of p work particularly poorly with increasing dimensionality. In some cases, it has been shown that fractional metrics are particularly effective when p is chosen in the range $(0, 1)$. Numerous proximity-based measures have also been shown to work effectively with increasing dimensionality.

The data distribution also has an impact on the distance function design. The simplest possible distance function that uses global distributions is the Mahalanobis metric. This metric is a generalization of the Euclidean measure, and stretches the distance values along the principal components according to their variance. A more sophisticated approach, referred to as *ISOMAP*, uses nonlinear embeddings to account for the impact of nonlinear data distributions. Local normalization can often provide more effective measures when the distribution of the data is heterogeneous.

Other data types such as categorical data, text, temporal, and graph data present further challenges. The determination of time-series and discrete-sequence similarity measures is closely related because the latter can be considered the categorical version of the former. The main problem is that two similar time series may exhibit different scaling of their behavioral and contextual attributes. This needs to be accounted for with the use of different normalization functions for the behavioral attribute, and the use of warping functions for the

contextual attribute. For the case of discrete sequence data, many distance and similarity functions, such as the edit distance and the LCSS, are commonly used.

Because distance functions are often intended to model user notions of similarity, feedback should be used, where possible, to improve the distance function design. This feedback can be used within the context of a parameterized model to learn the optimal parameters that are consistent with the user-provided feedback.

3.8 Bibliographic Notes

The problem of similarity computation has been studied extensively by data mining researchers and practitioners in recent years. The issues with high-dimensional data were explored in [17, 88, 266]. In the work of [88], the impact of the distance concentration effects on high-dimensional computation was analyzed. The work in [266] showed the relative advantages of picking distance functions that are locality sensitive. The work also showed the advantages of the Manhattan metric over the Euclidean metric. Fractional metrics were proposed in [17] and generally provide more accurate results than the Manhattan and Euclidean metric. The *ISOMAP* method discussed in this chapter was proposed in [490]. Numerous local methods are also possible for distance function computation. An example of an effective local method is the instance-based method proposed in [543].

Similarity in categorical data was explored extensively in [104]. In this work, a number of similarity measures were analyzed, and how they apply to the outlier detection problem was tested. The Goodall measure is introduced in [232]. The work in [122] uses information theoretic measures for computation of similarity. Most of the measures discussed in this chapter do not distinguish between mismatches on an attribute. However, a number of methods proposed in [74, 363, 473] distinguish between mismatches on an attribute value. The premise is that infrequent attribute values are statistically expected to be more different than frequent attribute values. Thus, in these methods, $S(x_i, y_i)$ is not always set to 0 (or the same value) when x_i and y_i are different. A local similarity measure is presented in [182]. Text similarity measures have been studied extensively in the information retrieval literature [441].

The area of time-series similarity measures is a rich one, and a significant number of algorithms have been designed in this context. An excellent tutorial on the topic may be found in [241]. The use of wavelets for similarity computation in time series is discussed in [130]. While DTW has been used extensively in the context of speech recognition, its use in data mining applications was first proposed by [87]. Subsequently, it has been used extensively [526] for similarity-based applications in data mining. The major challenge in data mining applications is its computationally intensive nature. Numerous methods [307] have been proposed in the time series data mining literature to speed up DTW. A fast method for computing a lower bound on DTW was proposed in [308], and how this can be used for exact indexing was shown. A window-based approach for computing similarity in sequences with noise, scaling, and translation was proposed in [53]. Methods for similarity search in multivariate time series and sequences were proposed in [499, 500]. The edit distance has been used extensively in biological data for computing similarity between sequences [244]. The use of transformation rules for time-series similarity has been studied in [283, 432]. Such rules can be used to create edit distance-like measures for continuous time series. Methods for the string-edit distance are proposed in [438]. It has been shown in [141], how the L_p-norm may be combined with the edit distance. Algorithms for the LCSS problem may be found in [77, 92, 270, 280]. A survey of these algorithms is available

in [92]. A variety of other measures for time series and sequence similarity are discussed in [32].

Numerous methods are available for similarity search in graphs. A variety of efficient shortest-path algorithms for finding distances between nodes may be found in [62]. The page rank algorithm is discussed in the Web mining book [357]. The NP-hardness of the graph isomorphism problem, and other closely related problems to the edit distance are discussed in [221]. The relationship between the maximum common subgraph problem and the graph-edit distance problem has been studied in [119, 120]. The problem of substructure similarity search, and the use of substructures for similarity search have been addressed in [520, 521]. A notion of mutation distance has been proposed in [522] to measure the distances between graphs. A method that uses the frequent substructures of a graph for similarity computation in clustering is proposed in [42]. A survey on graph-matching techniques may be found in [26].

User supervision has been studied extensively in the context of distance function learning. One of the earliest methods that parameterizes the weights of the L_p-norm was proposed in [15]. The problem of distance function learning has been formally related to that of classification and has been studied recently in great detail. A survey that covers the important topics in distance function learning is provided in [33].

3.9 Exercises

1. Compute the L_p-norm between $(1, 2)$ and $(3, 4)$ for $p = 1, 2, \infty$.

2. Show that the Mahalanobis distance between two data points is equivalent to the Euclidean distance on a transformed data set, where the transformation is performed by representing the data along the principal components, and dividing by the standard deviation of each component.

3. Download the *Ionosphere data set* from the *UCI Machine Learning Repository* [213], and compute the L_p distance between all pairs of data points, for $p = 1, 2$, and ∞. Compute the contrast measure on the data set for the different norms. Repeat the exercise after sampling the first r dimensions, where r varies from 1 to the full dimensionality of the data.

4. Compute the match-based similarity, cosine similarity, and the Jaccard coefficient, between the two sets $\{A, B, C\}$ and $\{A, C, D, E\}$.

5. Let $\overline{X}$ and $\overline{Y}$ be two data points. Show that the cosine angle between the vectors $\overline{X}$ and $\overline{Y}$ is given by:

$$cosine(\overline{X}, \overline{Y}) = \frac{||\overline{X}||^2 + ||\overline{Y}||^2 - ||\overline{X} - \overline{Y}||^2}{2||\overline{X}||||\overline{Y}||}. \tag{3.25}$$

6. Download the *KDD Cup Network Intrusion Data Set* for the *UCI Machine Learning Repository* [213]. Create a data set containing only the categorical attributes. Compute the nearest neighbor for each data point using the (a) match measure, and (b) inverse occurrence frequency measure. Compute the number of cases where there is a match on the class label.

7. Repeat Exercise 6 using only the quantitative attributes of the data set, and using the L_p-norm for values of $p = 1, 2, \infty$.

8. Repeat Exercise 6 using all attributes in the data set. Use the mixed-attribute function, and different combinations of the categorical and quantitative distance functions of Exercises 6 and 7.

9. Write a computer program to compute the edit distance.

10. Write a computer program to compute the $LCSS$ distance.

11. Write a computer program to compute the DTW distance.

12. Assume that $Edit(\overline{X}, \overline{Y})$ represents the cost of transforming the string $\overline{X}$ to $\overline{Y}$. Show that $Edit(\overline{X}, \overline{Y})$ and $Edit(\overline{Y}, \overline{X})$ are the same, as long as the insertion and deletion costs are the same.

13. Compute the edit distance, and $LCSS$ similarity between: (a) *ababcabc* and *babcbc* and (b) *cbacbacba* and *acbacbacb*. For the edit distance, assume equal cost of insertion, deletion, or replacement.

14. Show that $Edit(i, j)$, $LCSS(i, j)$, and $DTW(i, j)$ are all monotonic functions in i and j.

15. Compute the cosine measure using the raw frequencies between the following two sentences:

(a) "The sly fox jumped over the lazy dog."

(b) "The dog jumped at the intruder."

16. Suppose that insertion and deletion costs are 1, and replacement costs are 2 units for the edit distance. Show that the optimal edit distance between two strings can be computed only with insertion and deletion operations. Under the aforementioned cost assumptions, show that the optimal edit distance can be expressed as a function of the optimal $LCSS$ distance and the lengths of the two strings.

Chapter 4

Association Pattern Mining

"The pattern of the prodigal is: rebellion, ruin, repentance, reconciliation, restoration."—Edwin Louis Cole

4.1 Introduction

The classical problem of association pattern mining is defined in the context of supermarket data containing sets of items bought by customers, which are referred to as *transactions*. The goal is to determine *associations* between groups of items bought by customers, which can intuitively be viewed as k-way correlations between items. The most popular model for association pattern mining uses the frequencies of sets of items as the quantification of the level of association. The discovered sets of items are referred to as *large itemsets*, *frequent itemsets*, or *frequent patterns*. The association pattern mining problem has a wide variety of applications:

1. *Supermarket data:* The supermarket application was the original motivating scenario in which the association pattern mining problem was proposed. This is also the reason that the term *itemset* is used to refer to a frequent pattern in the context of supermarket *items* bought by a customer. The determination of frequent itemsets provides useful insights about target marketing and shelf placement of the items.

2. *Text mining:* Because text data is often represented in the bag-of-words model, frequent pattern mining can help in identifying co-occurring terms and keywords. Such co-occurring terms have numerous text-mining applications.

3. *Generalization to dependency-oriented data types:* The original frequent pattern mining model has been generalized to many dependency-oriented data types, such as time-series data, sequential data, spatial data, and graph data, with a few modifications. Such models are useful in applications such as Web log analysis, software bug detection, and spatiotemporal event detection.

C. C. Aggarwal, *Data Mining: The Textbook*, DOI 10.1007/978-3-319-14142-8_4

4. *Other major data mining problems:* Frequent pattern mining can be used as a subroutine to provide effective solutions to many data mining problems such as clustering, classification, and outlier analysis.

Because the frequent pattern mining problem was originally proposed in the context of market basket data, a significant amount of terminology used to describe both the data (e.g., *transactions*) and the output (e.g., *itemsets*) is borrowed from the supermarket analogy. From an application-neutral perspective, a frequent pattern may be defined as a frequent subset, defined on the universe of all possible sets. Nevertheless, because the market basket terminology has been used popularly, this chapter will be consistent with it.

Frequent itemsets can be used to generate *association rules* of the form $X \Rightarrow Y$, where X and Y are sets of items. A famous example of an association rule, which has now become part[1] of the data mining folklore, is $\{Beer\} \Rightarrow \{Diapers\}$. This rule suggests that buying beer makes it more likely that diapers will also be bought. Thus, there is a certain directionality to the implication that is quantified as a conditional probability. Association rules are particularly useful for a variety of target market applications. For example, if a supermarket owner discovers that $\{Eggs, Milk\} \Rightarrow \{Yogurt\}$ is an association rule, he or she can promote yogurt to customers who often buy eggs and milk. Alternatively, the supermarket owner may place yogurt on shelves that are located in proximity to eggs and milk.

The frequency-based model for association pattern mining is very popular because of its simplicity. However, the raw frequency of a pattern is not quite the same as the statistical significance of the underlying correlations. Therefore, numerous models for frequent pattern mining have been proposed that are based on statistical significance. This chapter will also explore some of these alternative models, which are also referred to as *interesting patterns*.

This chapter is organized as follows. Section 4.2 introduces the basic model for association pattern mining. The generation of association rules from frequent itemsets is discussed in Sect. 4.3. A variety of algorithms for frequent pattern mining are discussed in Sect. 4.4. This includes the *Apriori* algorithm, a number of enumeration tree algorithms, and a suffix-based recursive approach. Methods for finding interesting frequent patterns are discussed in Sect. 4.5. Meta-algorithms for frequent pattern mining are discussed in Sect. 4.6. Section 4.7 discusses the conclusions and summary.

4.2 The Frequent Pattern Mining Model

The problem of association pattern mining is naturally defined on unordered set-wise data. It is assumed that the database $\mathcal{T}$ contains a set of n transactions, denoted by $T_1 \ldots T_n$. Each transaction T_i is drawn on the universe of items U and can also be represented as a multidimensional record of dimensionality, $d = |U|$, containing only binary attributes. Each binary attribute in this record represents a particular item. The value of an attribute in this record is 1 if that item is present in the transaction, and 0 otherwise. In practical settings, the universe of items U is very large compared to the typical number of items in each transaction T_i. For example, a supermarket database may have tens of thousands of items, and a single transaction will typically contain less than 50 items. This property is often leveraged in the design of frequent pattern mining algorithms.

An *itemset* is a set of items. A k-itemset is an itemset that contains exactly k items. In other words, a k-itemset is a set of items of cardinality k. The fraction of transactions

[1]This rule was derived in some early publications on supermarket data. No assertion is made here about the likelihood of such a rule appearing in an arbitrary supermarket data set.

Table 4.1: Example of a snapshot of a market basket data set

tid	Set of items	Binary representation
1	$\{Bread, Butter, Milk\}$	110010
2	$\{Eggs, Milk, Yogurt\}$	000111
3	$\{Bread, Cheese, Eggs, Milk\}$	101110
4	$\{Eggs, Milk, Yogurt\}$	000111
5	$\{Cheese, Milk, Yogurt\}$	001011

in $T_1 \ldots T_n$ in which an itemset occurs as a subset provides a crisp quantification of its frequency. This frequency is also known as the *support*.

Definition 4.2.1 (Support) *The support of an itemset I is defined as the fraction of the transactions in the database $\mathcal{T} = \{T_1 \ldots T_n\}$ that contain I as a subset.*

The support of an itemset I is denoted by $sup(I)$. Clearly, items that are correlated will frequently occur together in transactions. Such itemsets will have high support. Therefore, the frequent pattern mining problem is that of determining itemsets that have the requisite level of *minimum support*.

Definition 4.2.2 (Frequent Itemset Mining) *Given a set of transactions $\mathcal{T} = \{T_1 \ldots T_n\}$, where each transaction T_i is a subset of items from U, determine all itemsets I that occur as a subset of at least a predefined fraction minsup of the transactions in $\mathcal{T}$.*

The predefined fraction *minsup* is referred to as the minimum support. While the default convention in this book is to assume that *minsup* refers to a fractional relative value, it is also sometimes specified as an absolute integer value in terms of the raw number of transactions. This chapter will always assume the convention of a relative value, unless specified otherwise. Frequent patterns are also referred to as frequent itemsets, or large itemsets. This book will use these terms interchangeably.

The unique identifier of a transaction is referred to as a *transaction identifier*, or *tid* for short. The frequent itemset mining problem may also be stated more generally in set-wise form.

Definition 4.2.3 (Frequent Itemset Mining: Set-wise Definition) *Given a set of sets $\mathcal{T} = \{T_1 \ldots T_n\}$, where each element of the set T_i is drawn on the universe of elements U, determine all sets I that occur as a subset of at least a predefined fraction minsup of the sets in $\mathcal{T}$.*

As discussed in Chap. 1, binary multidimensional data and set data are equivalent. This equivalence is because each multidimensional attribute can represent a set element (or item). A value of 1 for a multidimensional attribute corresponds to inclusion in the set (or transaction). Therefore, a transaction data set (or set of sets) can also be represented as a multidimensional binary database whose dimensionality is equal to the number of items.

Consider the transactions illustrated in Table 4.1. Each transaction is associated with a unique transaction identifier in the leftmost column, and contains a baskets of items that were bought together at the same time. The right column in Table 4.1 contains the binary multidimensional representation of the corresponding basket. The attributes of this binary representation are arranged in the order $\{Bread, Butter, Cheese, Eggs, Milk, Yogurt\}$. In

this database of 5 transactions, the *support* of $\{Bread, Milk\}$ is $2/5 = 0.4$ because both items in this basket occur in 2 out of a total of 5 transactions. Similarly, the support of $\{Cheese, Yogurt\}$ is 0.2 because it appears in only the last transaction. Therefore, if the minimum support is set to 0.3, then the itemset $\{Bread, Milk\}$ will be reported but not the itemset $\{Cheese, Yogurt\}$.

The number of frequent itemsets is generally very sensitive to the minimum support level. Consider the case where a minimum support level of 0.3 is used. Each of the items *Bread*, *Milk*, *Eggs*, *Cheese*, and *Yogurt* occur in more than 2 transactions, and can therefore be considered frequent items at a minimum support level of 0.3. These items are frequent 1-itemsets. In fact, the only item that is not frequent at a support level of 0.3 is *Butter*. Furthermore, the frequent 2-itemsets at a minimum support level of 0.3 are $\{Bread, Milk\}$, $\{Eggs, Milk\}$, $\{Cheese, Milk\}$, $\{Eggs, Yogurt\}$, and $\{Milk, Yogurt\}$. The only 3-itemset reported at a support level of 0.3 is $\{Eggs, Milk, Yogurt\}$. On the other hand, if the minimum support level is set to 0.2, it corresponds to an absolute support value of only 1. In such a case, every subset of every transaction will be reported. Therefore, the use of lower minimum support levels yields a larger number of frequent patterns. On the other hand, if the support level is too high, then no frequent patterns will be found. Therefore, an appropriate choice of the support level is crucial for discovering a set of frequent patterns with meaningful size.

When an itemset I is contained in a transaction, all its subsets will also be contained in the transaction. Therefore, the support of any subset J of I will always be at least equal to that of I. This property is referred to as the *support monotonicity property*.

Property 4.2.1 (Support Monotonicity Property) *The support of every subset J of I is at least equal to that of the support of itemset I.*

$$sup(J) \geq sup(I) \quad \forall J \subseteq I \tag{4.1}$$

The monotonicity property of support implies that every subset of a frequent itemset will also be frequent. This is referred to as the *downward closure property*.

Property 4.2.2 (Downward Closure Property) *Every subset of a frequent itemset is also frequent.*

The downward closure property of frequent patterns is algorithmically very convenient because it provides an important constraint on the inherent structure of frequent patterns. This constraint is often leveraged by frequent pattern mining algorithms to prune the search process and achieve greater efficiency. Furthermore, the downward closure property can be used to create concise representations of frequent patterns, wherein only the *maximal* frequent subsets are retained.

Definition 4.2.4 (Maximal Frequent Itemsets) *A frequent itemset is maximal at a given minimum support level minsup, if it is frequent, and no superset of it is frequent.*

In the example of Table 4.1, the itemset $\{Eggs, Milk, Yogurt\}$ is a maximal frequent itemset at a minimum support level of 0.3. However, the itemset $\{Eggs, Milk\}$ is not maximal because it has a superset that is also frequent. Furthermore, the set of *maximal* frequent patterns at a minimum support level of 0.3 is $\{Bread, Milk\}$, $\{Cheese, Milk\}$, and $\{Eggs, Milk, Yogurt\}$. Thus, there are only 3 maximal frequent itemsets, whereas the number of frequent itemsets in the entire transaction database is 11. All frequent itemsets can be derived from the maximal patterns by enumerating the subsets of the maximal frequent

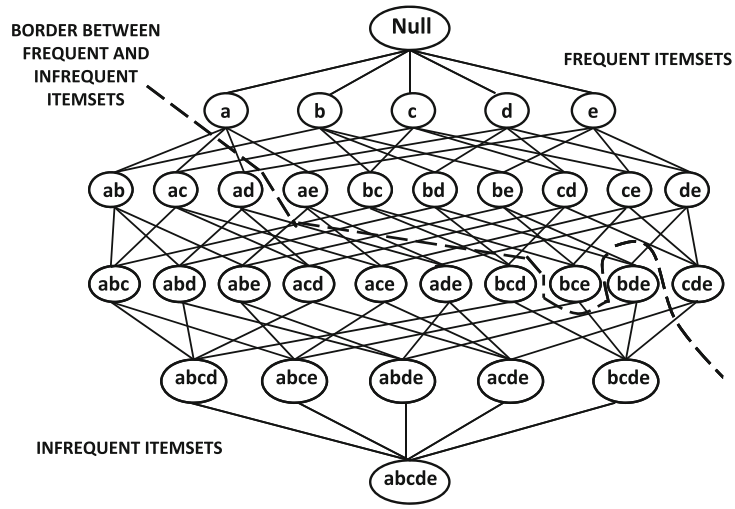


Figure 4.1: The itemset lattice

patterns. Therefore, the maximal patterns can be considered condensed representations of the frequent patterns. However, this condensed representation does not retain information about the support values of the subsets. For example, the support of $\{Eggs, Milk, Yogurt\}$ is 0.4, but it does not provide any information about the support of $\{Eggs, Milk\}$, which is 0.6. A different condensed representation, referred to as *closed frequent itemsets*, is able to retain support information as well. The notion of closed frequent itemsets will be studied in detail in Chap. 5.

An interesting property of itemsets is that they can be conceptually arranged in the form of a *lattice of itemsets*. This lattice contains one node for each of the $2^{|U|}$ sets drawn from the universe of items U. An edge exists between a pair of nodes, if the corresponding sets differ by exactly one item. An example of an itemset lattice of size $2^5 = 32$ on a universe of 5 items is illustrated in Fig. 4.1. The lattice represents the search space of frequent patterns. All frequent pattern mining algorithms, implicitly or explicitly, traverse this search space to determine the frequent patterns.

The lattice is separated into frequent and infrequent itemsets by a *border*, which is illustrated by a dashed line in Fig. 4.1. All itemsets above this border are frequent, whereas those below the border are infrequent. Note that all maximal frequent itemsets are adjacent to this border of itemsets. Furthermore, any valid border representing a true division between frequent and infrequent itemsets will always respect the downward closure property.

4.3 Association Rule Generation Framework

Frequent itemsets can be used to generate *association rules*, with the use of a measure known as the *confidence*. The confidence of a rule $X \Rightarrow Y$ is the conditional probability that a transaction contains the set of items Y, given that it contains the set X. This probability is estimated by dividing the support of itemset $X \cup Y$ with that of itemset X.

Definition 4.3.1 (Confidence) *Let X and Y be two sets of items. The confidence $conf(X \cup Y)$ of the rule $X \cup Y$ is the conditional probability of $X \cup Y$ occurring in a*

transaction, given that the transaction contains X. Therefore, the confidence $conf(X \Rightarrow Y)$ is defined as follows:

$$conf(X \Rightarrow Y) = \frac{sup(X \cup Y)}{sup(X)}. \tag{4.2}$$

The itemsets X and Y are said to be the *antecedent* and the *consequent* of the rule, respectively. In the case of Table 4.1, the support of $\{Eggs, Milk\}$ is 0.6, whereas the support of $\{Eggs, Milk, Yogurt\}$ is 0.4. Therefore, the confidence of the rule $\{Eggs, Milk\} \Rightarrow \{Yogurt\}$ is $(0.4/0.6) = 2/3$.

As in the case of support, a minimum confidence threshold *minconf* can be used to generate the most relevant association rules. Association rules are defined using both *support* and *confidence* criteria.

Definition 4.3.2 (Association Rules) *Let X and Y be two sets of items. Then, the rule $X \Rightarrow Y$ is said to be an association rule at a minimum support of minsup and minimum confidence of minconf, if it satisfies both the following criteria:*

1. *The support of the itemset $X \cup Y$ is at least minsup.*
2. *The confidence of the rule $X \Rightarrow Y$ is at least minconf.*

The first criterion ensures that a sufficient number of transactions are relevant to the rule; therefore, it has the required critical mass for it to be considered relevant to the application at hand. The second criterion ensures that the rule has sufficient strength in terms of conditional probabilities. Thus, the two measures quantify different aspects of the association rule.

The overall framework for association rule generation uses two phases. These phases correspond to the two criteria in Definition 4.3.2, representing the support and confidence constraints.

1. In the first phase, all the frequent itemsets are generated at the minimum support of *minsup*.
2. In the second phase, the association rules are generated from the frequent itemsets at the minimum confidence level of *minconf*.

The first phase is more computationally intensive and is, therefore, the more interesting part of the process. The second phase is relatively straightforward. Therefore, the discussion of the first phase will be deferred to the remaining portion of this chapter, and a quick discussion of the (more straightforward) second phase is provided here.

Assume that a set of frequent itemsets $\mathcal{F}$ is provided. For each itemset $I \in \mathcal{F}$, a simple way of generating the rules would be to partition the set I into all possible combinations of sets X and $Y = I - X$, such that $I = X \cup Y$. The confidence of each rule $X \Rightarrow Y$ can then be determined, and it can be retained if it satisfies the minimum confidence requirement. Association rules also satisfy a confidence monotonicity property.

Property 4.3.1 (Confidence Monotonicity) *Let X_1, X_2, and I be itemsets such that $X_1 \subset X_2 \subset I$. Then the confidence of $X_2 \Rightarrow I - X_2$ is at least that of $X_1 \Rightarrow I - X_1$.*

$$conf(X_2 \Rightarrow I - X_2) \geq conf(X_1 \Rightarrow I - X_1) \tag{4.3}$$

This property follows directly from definition of confidence and the property of support monotonicity. Consider the rules $\{Bread\} \Rightarrow \{Butter, Milk\}$ and $\{Bread, Butter\} \Rightarrow \{Milk\}$. The second rule is redundant with respect to the first because it will have the same support, but a confidence that is no less than the first. Because of confidence monotonicity, it is possible to report only the non-redundant rules. This issue is discussed in detail in the next chapter.

4.4 Frequent Itemset Mining Algorithms

In this section, a number of popular algorithms for frequent itemset generation will be discussed. Because there are a large number of frequent itemset mining algorithms, the focus of the chapter will be to discuss specific algorithms in detail to introduce the reader to the key tricks in algorithmic design. These tricks are often reusable across different algorithms because the same *enumeration tree framework* is used by virtually all frequent pattern mining algorithms.

4.4.1 Brute Force Algorithms

For a universe of items U, there are a total of $2^{|U|} - 1$ distinct subsets, excluding the empty set. All 2^5 subsets for a universe of 5 items are illustrated in Fig. 4.1. Therefore, one possibility would be to generate all these *candidate* itemsets, and count their support against the transaction database $\mathcal{T}$. In the frequent itemset mining literature, the term *candidate itemsets* is commonly used to refer to itemsets that might *possibly* be frequent (or *candidates* for being frequent). These candidates need to be verified against the transaction database by *support counting*. To count the support of an itemset, we would need to check whether a given itemset I is a subset of each transaction $T_i \in \mathcal{T}$. Such an exhaustive approach is likely to be impractical, when the universe of items U is large. Consider the case where $d = |U| = 1000$. In that case, there are a total of $2^{1000} > 10^{300}$ candidates. To put this number in perspective, if the fastest computer available today were somehow able to process one candidate in one elementary machine cycle, then the time required to process all candidates would be hundreds of orders of magnitude greater than the age of the universe. Therefore, this is not a practical solution.

Of course, one can make the brute-force approach faster by observing that no $(k+1)$-patterns are frequent if no k-patterns are frequent. This observation follows directly from the downward closure property. Therefore, one can enumerate and count the support of all the patterns with increasing length. In other words, one can enumerate and count the support of all patterns containing one item, two items, and so on, until for a certain length l, none of the candidates of length l turn out to be frequent. For sparse transaction databases, the value of l is typically very small compared to $|U|$. At this point, one can terminate. This is a significant improvement over the previous approach because it requires the enumeration of $\sum_{i=1}^{l} \binom{|U|}{i} \ll 2^{|U|}$ candidates. Because the longest frequent itemset is of *much* smaller length than $|U|$ in sparse transaction databases, this approach is orders of magnitude faster. However, the resulting computational complexity is still not satisfactory for large values of U. For example, when $|U| = 1000$ and $l = 10$, the value of $\sum_{i=1}^{10} \binom{|U|}{i}$ is of the order of 10^{23}. This value is still quite large and outside reasonable computational capabilities available today.

One observation is that even a very minor and rather blunt application of the downward closure property made the algorithm hundreds of orders of magnitude faster. Many of the fast algorithms for itemset generation use the downward closure property in a more refined way, both to generate the candidates and to prune them before counting. Algorithms for

frequent pattern mining search the lattice of possibilities (or candidates) for frequent patterns (see Fig. 4.1) and use the transaction database to count the support of candidates in this lattice. Better efficiencies can be achieved in a frequent pattern mining algorithm by using one or more of the following approaches:

1. Reducing the size of the explored search space (lattice of Fig. 4.1) by pruning candidate *itemsets* (lattice nodes) using tricks, such as the *downward closure* property.
2. Counting the support of each candidate more efficiently by pruning *transactions* that are known to be irrelevant for counting a candidate itemset.
3. Using compact data structures to represent either candidates or transaction databases that support efficient counting.

The first algorithm that used an effective pruning of the search space with the use of the downward closure property was the *Apriori* algorithm.

4.4.2 The Apriori Algorithm

The *Apriori* algorithm uses the downward closure property in order to prune the candidate search space. The downward closure property imposes a clear structure on the set of frequent patterns. In particular, information about the *infrequency* of itemsets can be leveraged to generate the superset candidates more carefully. Thus, if an itemset is infrequent, there is little point in counting the support of its superset candidates. This is useful for avoiding wasteful counting of support levels of itemsets that are known not to be frequent. The *Apriori* algorithm generates candidates with smaller length k first and counts their supports before generating candidates of length $(k+1)$. The resulting frequent k-itemsets are used to restrict the number of $(k+1)$-candidates with the downward closure property. Candidate generation and support counting of patterns with increasing length is interleaved in *Apriori*. Because the counting of candidate supports is the most expensive part of the frequent pattern generation process, it is extremely important to keep the number of candidates low.

For ease in description of the algorithm, it will be assumed that the items in U have a lexicographic ordering, and therefore an itemset $\{a, b, c, d\}$ can be treated as a (lexicographically ordered) string *abcd* of items. This can be used to impose an ordering among itemsets (patterns), which is the same as the order in which the corresponding strings would appear in a dictionary.

The *Apriori* algorithm starts by counting the supports of the individual items to generate the frequent 1-itemsets. The 1-itemsets are combined to create candidate 2-itemsets, whose support is counted. The frequent 2-itemsets are retained. In general, the frequent itemsets of length k are used to generate the candidates of length $(k+1)$ for increasing values of k. Algorithms that count the support of candidates with increasing length are referred to as *level-wise* algorithms. Let $\mathcal{F}_k$ denote the set of frequent k-itemsets, and $\mathcal{C}_k$ denote the set of candidate k-itemsets. The core of the approach is to iteratively generate the $(k+1)$-candidates $\mathcal{C}_{k+1}$ from frequent k-itemsets in $\mathcal{F}_k$ already found by the algorithm. The frequencies of these $(k+1)$-candidates are counted with respect to the transaction database. While generating the $(k+1)$-candidates, the search space may be pruned by checking whether all k-subsets of $\mathcal{C}_{k+1}$ are included in $\mathcal{F}_k$. So, how does one generate the relevant $(k+1)$-candidates in $\mathcal{C}_{k+1}$ from frequent k-patterns in $\mathcal{F}_k$?

If a pair of itemsets X and Y in $\mathcal{F}_k$ have $(k-1)$ items in common, then a join between them using the $(k-1)$ common items will create a candidate itemset of size $(k+1)$. For example, the two 3-itemsets $\{a, b, c\}$ (or *abc* for short) and $\{a, b, d\}$ (or *abd* for short), when

Algorithm *Apriori*(Transactions: $\mathcal{T}$, Minimum Support: *minsup*)
begin
  $k = 1$;
  $\mathcal{F}_1 = \{$ All Frequent 1-itemsets $\}$;
  while $\mathcal{F}_k$ is not empty **do begin**
    Generate $\mathcal{C}_{k+1}$ by joining itemset-pairs in $\mathcal{F}_k$;
    Prune itemsets from $\mathcal{C}_{k+1}$ that violate downward closure;
    Determine $\mathcal{F}_{k+1}$ by support counting on $(\mathcal{C}_{k+1}, \mathcal{T})$ and retaining
      itemsets from $\mathcal{C}_{k+1}$ with support at least *minsup*;
    $k = k + 1$;
  end;
  return$(\cup_{i=1}^{k} \mathcal{F}_i)$;
end

Figure 4.2: The *Apriori* algorithm

joined together on the two common items a and b, will yield the candidate 4-itemset *abcd*. Of course, it is possible to join other frequent patterns to create the same candidate. One might also join *abc* and *bcd* to achieve the same result. Suppose that all four of the 3-subsets of *abcd* are present in the set of frequent 3-itemsets. One can create the candidate 4-itemset in $\binom{4}{2} = 6$ different ways. To avoid redundancy in candidate generation, the convention is to impose a lexicographic ordering on the items and use the first $(k-1)$ items of the itemset for the join. Thus, in this case, the only way to generate *abcd* would be to join using the first two items a and b. Therefore, the itemsets *abc* and *abd* would need to be joined to create *abcd*. Note that, if either of *abc* and *abd* are *not* frequent, then *abcd* will *not* be generated as a candidate using this join approach. Furthermore, in such a case, it is assured that *abcd* will not be frequent because of the downward closure property of frequent itemsets. Thus, the downward closure property ensures that the candidate set generated using this approach does not miss any itemset that is truly frequent. As we will see later, this *non-repetitive* and *exhaustive* way of generating candidates can be interpreted in the context of a conceptual hierarchy of the patterns known as the *enumeration tree*. Another point to note is that the joins can usually be performed very efficiently. This efficiency is because, if the set $\mathcal{F}_k$ is sorted in lexicographic (dictionary) order, all itemsets with a common set of items in the first $k-1$ positions will appear contiguously, allowing them to be located easily.

A *level-wise pruning trick* can be used to further reduce the size of the $(k+1)$-candidate set. All the k-subsets (i.e., subsets of cardinality k) of an itemset $I \in \mathcal{C}_{k+1}$ need to be present in $\mathcal{F}_k$ because of the downward closure property. Otherwise, it is guaranteed that the itemset I is not frequent. Therefore, it is checked whether all k-subsets of each itemset $I \in \mathcal{C}_{k+1}$ are present in $\mathcal{F}_k$. If this is not the case, then such itemsets I are removed from $\mathcal{C}_{k+1}$.

After the candidate itemsets $\mathcal{C}_{k+1}$ of size $(k+1)$ have been generated, their support can be determined by counting the number of occurrences of each candidate in the transaction database $\mathcal{T}$. Only the candidate itemsets that have the required minimum support are retained to create the set of $(k+1)$-frequent itemsets $\mathcal{F}_{k+1} \subseteq \mathcal{C}_{k+1}$. In the event that the set $\mathcal{F}_{k+1}$ is empty, the algorithm terminates. At termination, the union $\cup_{i=1}^{k} \mathcal{F}_i$ of the frequent patterns of different sizes is reported as the final output of the algorithm.

The overall algorithm is illustrated in Fig. 4.2. The heart of the algorithm is an iterative loop that generates $(k+1)$-candidates from frequent k-patterns for successively higher values of k and counts them. The three main operations of the algorithm are candidate

generation, pruning, and support counting. Of these, the support counting process is the most expensive one because it depends on the size of the transaction database $\mathcal{T}$. The level-wise approach ensures that the algorithm is relatively efficient at least from a disk-access cost perspective. This is because each set of candidates in $\mathcal{C}_{k+1}$ can be counted in a single pass over the data without the need for random disk accesses. The number of passes over the data is, therefore, equal to the cardinality of the longest frequent itemset in the data. Nevertheless, the counting procedure is still quite expensive especially if one were to use the naive approach of checking whether each itemset is a subset of a transaction. Therefore, efficient support counting procedures are necessary.

4.4.2.1 Efficient Support Counting

To perform support counting, *Apriori* needs to *efficiently* examined whether each candidate itemset is present in a transaction. This is achieved with the use of a data structure known as the *hash tree*. The hash tree is used to carefully organize the candidate patterns in $\mathcal{C}_{k+1}$ for more efficient counting. Assume that the items in the transactions and the candidate itemsets are sorted lexicographically. A hash tree is a tree with a fixed degree of the internal nodes. Each internal node is associated with a random hash function that maps to the index of the different children of that node in the tree. A leaf node of the hash tree contains a list of lexicographically sorted itemsets, whereas an interior node contains a hash table. Every itemset in $\mathcal{C}_{k+1}$ is contained in exactly one leaf node of the hash tree. The hash functions in the interior nodes are used to decide which candidate itemset belongs to which leaf node with the use of a methodology described below.

It may be assumed that all interior nodes use the same hash function $f(\cdot)$ that maps to $[0 \ldots h-1]$. The value of h is also the branching degree of the hash tree. A candidate itemset in $\mathcal{C}_{k+1}$ is mapped to a leaf node of the tree by defining a path from the root to the leaf node with the use of these hash functions at the internal nodes. Assume that the root of the hash tree is level 1, and all successive levels below it increase by 1. As before, assume that the items in the candidates and transactions are arranged in lexicographically sorted order. At an interior node in level i, a hash function is applied to the ith item of a candidate itemset $I \in \mathcal{C}_{k+1}$ to decide which branch of the hash tree to follow for the candidate itemset. The tree is constructed recursively in top-down fashion, and a minimum threshold is imposed on the number of candidates in the leaf node to decide where to terminate the hash tree extension. The candidate itemsets in the leaf node are stored in sorted order.

To perform the counting, all possible candidate k-itemsets in $\mathcal{C}_{k+1}$ that are subsets of a transaction $T_j \in \mathcal{T}$ are discovered in a single exploration of the hash tree. To achieve this goal, all possible paths in the hash tree, whose leaves *might* contain subset itemsets of the transaction T_j, are discovered using a recursive traversal. The selection of the relevant leaf nodes is performed by recursive traversal as follows. At the root node, all branches are followed such that any of the items in the transaction T_j hash to one of the branches. At a given interior node, if the ith item of the transaction T_j was last hashed (at the parent node), then all items following it in the transaction are hashed to determine the possible children to follow. Thus, by following all these paths, the relevant leaf nodes in the tree are determined. The candidates in the leaf node are stored in sorted order and can be compared efficiently to the transaction T_j to determine whether they are relevant. This process is repeated for each transaction to determine the final support count of each itemset in $\mathcal{C}_{k+1}$.

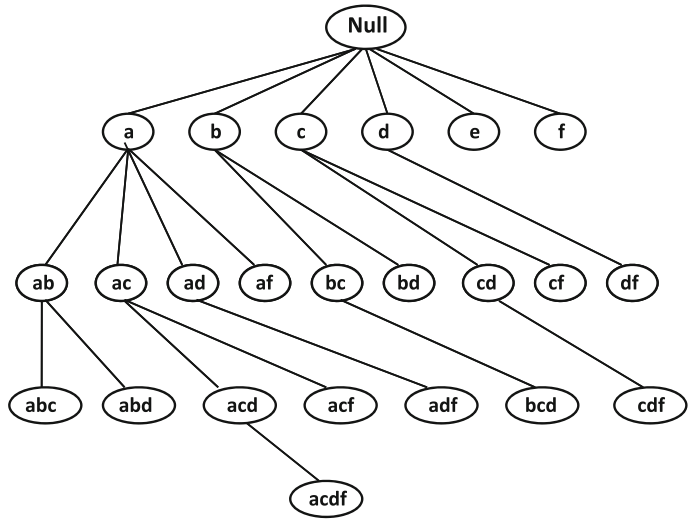


Figure 4.3: The lexicographic or enumeration tree of frequent itemsets

4.4.3 Enumeration-Tree Algorithms

These algorithms are based on set enumeration concepts, in which the different candidate itemsets are generated in a tree-like structure known as the *enumeration tree*, which is a subgraph of the lattice of itemsets introduced in Fig. 4.1. This tree-like structure is also referred to as a lexicographic tree because it is dependent on an upfront lexicographic ordering among the items. The candidate patterns are generated by growing this lexicographic tree. This tree can be grown in a wide variety of different strategies to achieve different trade-offs between storage, disk access costs, and computational efficiency. Because most of the discussion in this section will use this structure as a base for algorithmic development, this concept will be discussed in detail here. The main characteristic of the enumeration tree (or lexicographic tree) is that it provides an abstract hierarchical representation of the itemsets. This representation is leveraged by frequent pattern mining algorithms for systematic exploration of the candidate patterns in a non-repetitive way. The final output of these algorithms can also be viewed as an enumeration tree structure that is defined only on the frequent itemsets. The enumeration tree is defined on the frequent itemsets in the following way:

1. A node exists in the tree corresponding to each frequent itemset. The root of the tree corresponds to the *null* itemset.

2. Let $I = \{i_1, \ldots i_k\}$ be a frequent itemset, where $i_1, i_2 \ldots i_k$ are listed in lexicographic order. The parent of the node I is the itemset $\{i_1, \ldots i_{k-1}\}$. Thus, the child of a node can only be extended with items occurring lexicographically *after* all items occurring in that node. The enumeration tree can also be viewed as a *prefix* tree on the lexicographically ordered string representation of the itemsets.

This definition of an ancestral relationship naturally creates a tree structure on the nodes, which is rooted at the *null* node. An example of the frequent portion of the enumeration tree is illustrated in Fig. 4.3. An item that is used to extend a node to its (frequent) child in the enumeration tree is referred to as a *frequent tree extension*, or simply a tree extension. In the example of Fig. 4.3, the frequent tree extensions of node a are b, c, d, and f, because

these items extend node a to the frequent itemsets ab, ac, ad, and af, respectively. The lattice provides many paths to extend the *null* itemset to a node, whereas an enumeration tree provides only one path. For example, itemset ab can be extended either in the order $a \rightarrow ab$, or in the order $b \rightarrow ab$ in the lattice. However, only the former is possible in the enumeration tree after the lexicographic ordering has been fixed. Thus, the lexicographic ordering imposes a strictly hierarchical structure on the itemsets. This hierarchical structure enables *systematic* and *non-redundant* exploration of the itemset search space by algorithms that generate candidates by extending frequent itemsets with one item at a time. The enumeration tree can be constructed in many ways with different lexicographic orderings of items. The impact of this ordering will be discussed later.

Most of the enumeration tree algorithms work by growing this enumeration tree of frequent itemsets with a predefined strategy. First, the root node of the tree is extended by finding the frequent 1-items. Then, these nodes may be extended to create *candidates*. These are checked against the transaction database to determine the ones that are frequent. The enumeration tree framework provides an order and structure to the frequent itemset discovery, which can be leveraged to improve the counting and pruning process of candidates. In the following discussion, the terms "node" and "itemset" will be used interchangeably. Therefore, the notation P will be used to denote both an itemset, and its corresponding node in the enumeration tree.

So, how can candidates nodes be generated in a systematic way from the frequent nodes in the enumeration tree that have already been discovered? For an item i to be considered a candidate for extending a frequent node P to $P \cup \{i\}$, it must also be a frequent extension of the parent Q of P. This is because of the downward closure property, and it can be used to systematically define the candidate extensions of a node P after the frequent extensions of its parent Q have been determined. Let $F(Q)$ represent the frequent lexicographic tree extensions of node Q. Let $i \in F(Q)$ be the frequent extension item that extends frequent node Q to frequent node $P = Q \cup \{i\}$. Let $C(P)$ denote the subset of items from $F(Q)$ occurring lexicographically *after* the item i used to extend node Q to node P. The set $C(P)$ defines the *candidate extension items* of node P, which are defined as items that can be appended at the end of P to create candidate itemsets. This provides a systematic methodology to generate candidate children of node P. As we will see in Sect. 4.4.3.1, the resulting candidates are identical to those generated by *Apriori* joins. Note that the relationship $F(P) \subseteq C(P) \subset F(Q)$ is always true. The value of $F(P)$ in Fig. 4.3, when $P = ab$, is $\{c, d\}$. The value of $C(P)$ for $P = ab$ is $\{c, d, f\}$ because these are frequent extensions of parent itemset $Q = \{a\}$ of P occurring lexicographically after the item b. Note that the set of *candidate* extensions $C(ab)$ also contains the (infrequent) item f that the set of *frequent* extensions $F(ab)$ does not. Such infrequent item extensions correspond to *failed* candidate tests in all enumeration tree algorithms. Note that the infrequent itemset abf is not included in the *frequent* itemset tree of Fig. 4.3. It is also possible to create an enumeration tree structure on the *candidate* itemsets, which contains an additional layer of infrequent candidate extensions of the nodes in Fig. 4.3. Such a tree would contain abf.

Enumeration tree algorithms iteratively grow the enumeration tree $\mathcal{ET}$ of frequent patterns. A very generic description of this iterative step, which is executed repeatedly to extend the enumeration tree $\mathcal{ET}$, is as follows:

Select one or more nodes $\mathcal{P}$ in $\mathcal{ET}$;
Determine candidate extensions $C(P)$ for each such node $P \in \mathcal{P}$;
Count support of generated candidates;
Add frequent candidates to $\mathcal{ET}$ (tree growth);

Algorithm *GenericEnumerationTree*(Transactions: $\mathcal{T}$,
Minimum Support: *minsup*)
begin
Initialize enumeration tree $\mathcal{ET}$ to single *Null* node;
while any node in $\mathcal{ET}$ has not been examined **do begin**
Select one of more unexamined nodes $\mathcal{P}$ from $\mathcal{ET}$ for examination;
Generate candidates extensions $C(P)$ of each node $P \in \mathcal{P}$;
Determine frequent extensions $F(P) \subseteq C(P)$ for each $P \in \mathcal{P}$ with support counting;
Extend each node $P \in \mathcal{P}$ in $\mathcal{ET}$ with its frequent extensions in $F(P)$;
end
return enumeration tree $\mathcal{ET}$;
end

Figure 4.4: Generic enumeration-tree growth with unspecified growth strategy and counting method

This approach is continued until none of the nodes can be extended any further. At this point, the algorithm terminates. A more detailed description is provided in Fig. 4.4. Interestingly, almost all frequent pattern mining algorithms can be viewed as variations and extensions of this simple enumeration-tree framework. Within this broader framework, a wide variability exists both in terms of the growth strategy of the tree and the specific data structures used for support counting. Therefore, the description of Fig. 4.4 is very generic because none of these aspects are specified. The different choices of growth strategy and counting methodology provide different trade-offs between efficiency, space-requirements, and disk access costs. For example, in breadth-first strategies, the node set $\mathcal{P}$ selected in an iteration of Fig. 4.4 corresponds to all nodes at one level of the tree. This approach may be more relevant for disk-resident databases because all nodes at a single level of the tree can be extended during one counting pass on the transaction database. Depth-first strategies select a single node at the deepest level to create $\mathcal{P}$. These strategies may have better ability to explore the tree deeply and discover long frequent patterns early. The early discovery of longer patterns is especially useful for computational efficiency in maximal pattern mining and for better memory management in certain classes of *projection-based* algorithms.

Because the counting approach is the most expensive part, the different techniques attempt to use growth strategies that optimize the work done during counting. Furthermore, it is crucial for the counting data structures to be efficient. This section will explore some of the common algorithms, data structures, and pruning strategies that leverage the enumeration-tree structure in the counting process. Interestingly, the enumeration-tree framework is so general that even the *Apriori* algorithm can be interpreted within this framework, although the concept of an enumeration tree was not used when *Apriori* was proposed.

4.4.3.1 Enumeration-Tree-Based Interpretation of Apriori

The *Apriori* algorithm can be viewed as the level-wise construction of the enumeration tree in breadth-first manner. The *Apriori* join for generating candidate $(k+1)$-itemsets is performed in a non-redundant way by using only the *first* $(k-1)$ items from two frequent k-itemsets. This is equivalent to joining all pairs of *immediate siblings* at the kth level of the enumeration tree. For example, the children of *ab* in Fig. 4.3 may be obtained by joining

ab with all its frequent siblings (other children of node a) that occur lexicographically later than it. In other words, the join operation of node P with its lexicographically later frequent siblings produces the candidates corresponding to the extension of P with each of its candidate tree-extensions $C(P)$. In fact, the candidate extensions $C(P)$ for all nodes P at a given level of the tree can be *exhaustively* and *non-repetitively* generated by using joins between all pairs of frequent *siblings* at that level. The *Apriori* pruning trick then discards some of the enumeration tree nodes because they are guaranteed not to be frequent. A single pass over the transaction database is used to count the support of these candidate extensions, and generate the *frequent* extensions $F(P) \subseteq C(P)$ for each node P in the level being extended. The approach terminates when the tree cannot be grown further in a particular pass over the database. Thus, the join operation of *Apriori* has a direct interpretation in terms of the enumeration tree, and the *Apriori* algorithm implicitly extends the enumeration tree in a level-wise fashion with the use of joins.

4.4.3.2 TreeProjection and DepthProject

TreeProjection is a family of methods that uses recursive *projections* of the transactions down the enumeration tree structure. The goal of these recursive projections is to reuse the counting work that has already been done at a given node of the enumeration tree at its descendent nodes. This reduces the overall counting effort by orders of magnitude. *TreeProjection* is a general framework that shows how to use database projection in the context of a variety of different strategies for construction of the enumeration tree, such as breadth-first, depth-first, or a combination of the two. The *DepthProject* approach is a specific instantiation of this framework with the depth-first strategy. Different strategies have different trade-offs between the memory requirements and disk-access costs.

The main observation in projection-based methods is that if a transaction does not contain the itemset corresponding to an enumeration-tree node, then this transaction will not be relevant for counting at any descendent (superset itemset) of that node. Therefore, when counting is done at an enumeration-tree node, the information about irrelevant transactions should somehow be preserved for counting at its descendent nodes. This is achieved with the notion of *projected databases*. Each projected transaction database is specific to an enumeration-tree node. Transactions that do not contain the itemset P are not included in the projected databases at node P and its descendants. This results in a significant reduction in the number of projected transactions. Furthermore, only the candidate extension items of P, denoted by $C(P)$, are relevant for counting at any of the subtrees rooted at node P. Therefore, the projected database at node P can be expressed only in terms of the items in $C(P)$. The size of $C(P)$ is much smaller than the universe of items, and therefore the projected database contains a smaller number of items per transaction with increasing size of P. We denote the projected database at node P by $\mathcal{T}(P)$. For example, consider the node $P = ab$ in Fig. 4.3, in which the candidate items for extending ab are $C(P) = \{c, d, f\}$. Then, the transaction $abcfg$ maps to the projected transaction cf in $\mathcal{T}(P)$. On the other hand, the transaction $acfg$ is not even present in $\mathcal{T}(P)$ because $P = ab$ is not a subset of $acfg$. The special case $\mathcal{T}(Null) = \mathcal{T}$ corresponds to the top level of the enumeration tree and is equal to the full transaction database. In fact, the subproblem at node P with transaction database $\mathcal{T}(P)$ is structurally identical to the top-level problem, except that it is a much smaller problem focused on determining frequent patterns with a prefix of P. Therefore, the frequent node P in the enumeration tree can be extended further by counting the support of individual items in $C(P)$ using the relatively small database $\mathcal{T}(P)$. This

Algorithm *ProjectedEnumerationTree*(Transactions: $\mathcal{T}$,
Minimum Support: *minsup*)
begin
Initialize enumeration tree $\mathcal{ET}$ to a single $(Null, \mathcal{T})$ root node;
while any node in $\mathcal{ET}$ has not been examined **do begin**
Select an unexamined node $(P, \mathcal{T}(P))$ from $\mathcal{ET}$ for examination;
Generate candidates item extensions $C(P)$ of node $(P, \mathcal{T}(P))$;
Determine frequent item extensions $F(P) \subseteq C(P)$ by support counting
of individual items in smaller projected database $\mathcal{T}(P)$;
Remove infrequent items in $\mathcal{T}(P)$;
for each frequent item extension $i \in F(P)$ **do begin**
Generate $\mathcal{T}(P \cup \{i\})$ from $\mathcal{T}(P)$;
Add $(P \cup \{i\}, \mathcal{T}(P \cup \{i\}))$ as child of P in $\mathcal{ET}$;
end
end
return enumeration tree $\mathcal{ET}$;
end

Figure 4.5: Generic enumeration-tree growth with unspecified growth strategy and database projections

results in a simplified and efficient counting process of candidate 1-item *extensions* rather than *itemsets*.

The enumeration tree can be grown with a variety of strategies such as the breadth-first or depth-first strategies. At each node, the counting is performed with the use of the projected database rather than the entire transaction database, and a further reduced and projected transaction database is propagated to the children of P. At each level of the hierarchical projection down the enumeration tree, the number of items and the number of transactions in the projected database are reduced. The basic idea is that $\mathcal{T}(P)$ contains the minimal portion of the transaction database that is *relevant* for counting the subtree rooted at P, based on the removal of irrelevant transactions and items by the counting process that has already been performed at higher levels of the tree. By *recursively* projecting the transaction database down the enumeration tree, this counting work is reused. We refer to this approach as *projection-based reuse* of counting effort.

The generic enumeration-tree algorithm with hierarchical projections is illustrated in Fig. 4.5. This generic algorithm does not assume any specific exploration strategy, and is quite similar to the generic enumeration-tree pseudocode shown in Fig. 4.4. There are two differences between the pseudocodes.

1. For simplicity of notation, we have shown the exploration of a single node P at one time in Fig. 4.5, rather than a group of nodes $\mathcal{P}$ (as in Fig. 4.4). However, the pseudocode shown in Fig. 4.5 can easily be rewritten for a group of nodes $\mathcal{P}$. Therefore, this is not a significant difference.

2. The key difference is that the projected database $\mathcal{T}(P)$ is used to count support at node P. Each node in the enumeration tree is now represented by the itemset and projected database pair $(P, \mathcal{T}(P))$. This is a very important difference because $\mathcal{T}(P)$ is much smaller than the original database. Therefore, a significant amount of information gained by counting the supports of ancestors of node P, is preserved in $\mathcal{T}(P)$. Furthermore, one only needs to count the support of single item *extensions* of node P in $\mathcal{T}(P)$ (rather than entire itemsets) in order to grow the subtree at P further.

The enumeration tree can be constructed in many different ways depending on the lexicographic ordering of items. How should the items be ordered? The structure of the enumeration tree has a built-in bias towards creating unbalanced trees in which the lexicographically smaller items have more descendants. For example, in Fig. 4.3, node a has many more descendants than node f. Therefore, ordering the items from least support to greatest support ensures that the computationally heavier branches of the enumeration tree have fewer relevant transactions. This is helpful in maximizing the selectivity of projections and ensuring better efficiency.

The strategy used for selection of the node P defines the order in which the nodes of the enumeration tree are materialized. This strategy has a direct impact on memory management because projected databases, which are no longer required for future computation, can be deleted. In depth-first strategies, the lexicographically smallest unexamined node P is selected for extension. In this case, one only needs to maintain projected databases along the current path of the enumeration tree being explored. In breadth-first strategies, an entire group of nodes $\mathcal{P}$ corresponding to all patterns of a particular size are grown first. In such cases, the projected databases need to be simultaneously maintained along the full breadth of the enumeration tree $\mathcal{ET}$ at the two current levels involved in the growth process. Although it may be possible to perform the projection on such a large number of nodes for smaller transaction databases, some modifications to the basic framework of Fig. 4.5 are needed for the general case of larger databases.

In particular, breadth-first variations of the *TreeProjection* framework perform hierarchical projections on the fly during counting from their ancestor nodes. The depth-first variations of *TreeProjection*, such as *DepthProject*, achieve full projection-based reuse because the projected transactions can be consistently maintained at each materialized node along the relatively small path of the enumeration tree from the root to the current node. The breadth-first variations do have the merit that they can optimize disk-access costs for arbitrarily large databases at the expense of losing some of the power of projection-based reuse. As will be discussed later, all (full) projection-based reuse methods face memory-management challenges with increasing database size. These additional memory requirements can be viewed as the price for persistently storing the relevant work done in earlier iterations in the indirect form of projected databases. There is usually a different trade-off between disk-access costs and memory/computational requirements in various strategies, which is exploited by the *TreeProjection* framework. The bibliographic notes contain pointers to specific details of these optimized variations of *TreeProjection*.

Optimized counting at deeper level nodes: The projection-based approach enables specialized counting techniques at deeper level nodes near the leaves of the enumeration tree. These specialized counting methods can provide the counts of *all* the itemsets in a lower-level subtree in the time required to *scan* the projected database. Because such nodes are more numerous, this can lead to large computational improvements.

What is the point at which such counting methods can be used? When the number of frequent extensions $F(P)$ of a node P falls below a threshold t such that 2^t fits in memory, an approach known as *bucketing* can be used. To obtain the best computational results, the value of t used should be such that 2^t is much smaller than the number of transactions in the projected database. This can occur only when there are many repeated transactions in the projected database.

A two-phase approach is used. In the first phase, the count of each distinct transaction in the projected database is determined. This can be accomplished easily by maintaining $2^{|F(P)|}$ buckets or counters, scanning the transactions one by one, and adding counts to the buckets. This phase can be completed in a simple scan of the small (projected) database

of transactions. Of course, this process only provides transaction counts and not itemset counts.

In the second phase, the transaction frequency counts can be further aggregated in a systematic way to create itemset frequency counts. Conceptually, the process of aggregating projected transaction counts is similar to arranging all the $2^{|F(P)|}$ possibilities in the form of a lattice, as illustrated in Fig. 4.1. The counts of the lattice nodes, which are computed in the first phase, are aggregated up the lattice structure by adding the count of immediate supersets to their subsets. For small values of $|F(P)|$, such as 10, this phase is not the limiting computational factor, and the overall time is dominated by that required to scan the projected database in the first phase. An efficient implementation of the second phase is discussed in detail below.

Consider a string composed of 0, 1, and $*$ that refers to an itemset in which the positions with 0 and 1 are fixed to those values (corresponding to presence or absence of items), whereas a position with a $*$ is a "don't care." Thus, all transactions can be expressed in terms of 0 and 1 in their binary representation. On the other hand, all itemsets can be expressed in terms of 1 and $*$ because itemsets are traditionally defined with respect to presence of items and ambiguity with respect to absence. Consider, for example, the case when $|F(P)| = 4$, and there are four items, numbered $\{1, 2, 3, 4\}$. An itemset containing items 2 and 4 is denoted by $*1*1$. We start with the information on $2^4 = 16$ bitstrings that are composed 0 and 1. These represent all possible distinct transactions. The algorithm aggregates the counts in $|F(P)|$ iterations. The count for a string with a "*" in a particular position may be obtained by adding the counts for the strings with a 0 and 1 in those positions. For example, the count for the string *1*1 may be expressed as the sum of the counts of the strings 01*1 and 11*1. The positions may be processed in any order, although the simplest approach is to aggregate them from the least significant to the most significant.

A simple pseudocode to perform the aggregation is described below. In this pseudocode, the initial value of $bucket[i]$ is equal to the count of the transaction corresponding to the bitstring representation of integer i. The final value of $bucket[i]$ is one in which the transaction count has been converted to an itemset count by successive aggregation. In other words, the 0s in the bitstring are replaced by "don't cares."

```
for i := 1 to k do begin
   for j := 1 to 2^k do begin
      if the ith bit of bitstring representation
          of j is 0 then bucket[j] = bucket[j] + bucket[j + 2^(i-1)];
   endfor
endfor
```

An example of bucketing for $|F(P)| = 4$ is illustrated in Fig. 4.6. The bucketing trick is performed commonly at lower nodes of the tree because the value of $|F(P)|$ falls drastically at the lower levels. Because the nodes at the lower levels dominate the total number of nodes in the enumeration-tree structure, the impact of bucketing can be very significant.

Optimizations for maximal pattern mining: The *DepthProject* method, which is a depth-first variant of the approach, is particularly adaptable for maximal pattern discovery. In this case, the enumeration tree is explored in depth-first order to maximize the advantages of pruning the search space of regions containing only non-maximal patterns. The order of construction of the enumeration tree is important in the particular case of maximal frequent

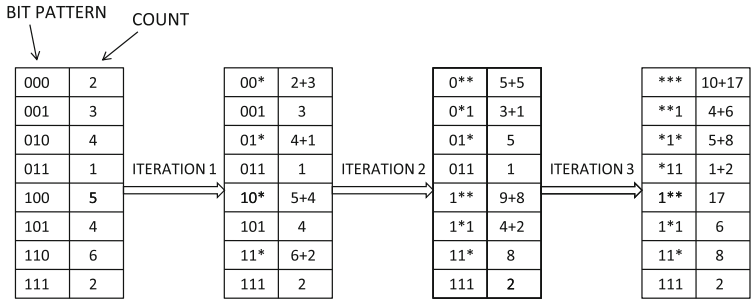


Figure 4.6: Performing the second phase of bucketing

pattern mining because certain kinds of non-maximal search-space pruning are optimized with the depth-first order. The notion of *lookaheads* is one such optimization.

Let $C(P)$ be the set of candidate item extensions of node P. Before support counting, it is tested whether $P \cup C(P)$ is a subset of a frequent pattern that has already been found. If such is indeed the case, then the pattern $P \cup C(P)$ is a non-maximal frequent pattern, and the entire subtree (of the enumeration tree) rooted at P can be pruned. This kind of pruning is referred to as *superset-based pruning*. When P cannot be pruned, the supports of its candidate extensions need to be determined. During this support counting, the support of $P \cup C(P)$ is counted along with the individual item extensions of P. If $P \cup C(P)$ is found to be frequent, then it eliminates any further work of counting the support of (non-maximal) nodes in the subtree rooted at node P.

While lookaheads can also be used with breadth-first algorithms, they are more effective with a depth-first strategy. In depth-first methods, longer patterns tend to be found first, and are, therefore, already available in the frequent set for superset-based pruning. For example, consider a frequent pattern of length 20 with 2^{20} subsets. In a depth-first strategy, it can be shown that the pattern of length 20 will be discovered after exploring only 19 of its immediate prefixes. On the other hand, a breadth-first method may remain trapped by discovery of shorter patterns. Therefore, the longer patterns become available very early in depth-first methods such as *DepthProject* to prune large portions of the enumeration tree with superset-based pruning.

4.4.3.3 Vertical Counting Methods

The *Partition* [446] and *Monet* [273] methods pioneered the concept of *vertical database representations* of the transaction database $\mathcal{T}$. In the *vertical representation*, each item is associated with a list of its transaction identifiers (*tids*). It can also be thought of as using the transpose of the binary transaction data matrix representing the transactions so that columns are transformed to rows. These rows are used as the new "records." Each item, thus, has a *tid* list of identifiers of transactions containing it. For example, the vertical representation of the database of Table 4.1 is illustrated in Table 4.2. Note that the binary matrix in Table 4.2 is the transpose of that in Table 4.1.

The intersection of two item *tid* lists yields a new *tid* list whose length is equal to the support of that 2-itemset. Further intersection of the resulting *tid* list with that of another item yields the support of 3-itemsets. For example, the intersection of the *tid* lists of *Milk* and *Yogurt* yields $\{2, 4, 5\}$ with length 3. Further intersection of the *tid* list of $\{Milk, Yogurt\}$ with that of *Eggs* yields the *tid* list $\{2, 4\}$ of length 2. This means that the support of

Table 4.2: Vertical representation of market basket data set

Item	Set of tids	Binary representation
Bread	$\{1, 3\}$	10100
Butter	$\{1\}$	10000
Cheese	$\{3, 5\}$	00101
Eggs	$\{2, 3, 4\}$	01110
Milk	$\{1, 2, 3, 4, 5\}$	11111
Yogurt	$\{2, 4, 5\}$	01011

$\{Milk, Yogurt\}$ is $3/5 = 0.6$ and that of $\{Milk, Eggs, Yogurt\}$ is $2/5 = 0.4$. Note that one can also intersect the smaller *tid* lists of $\{Milk, Yogurt\}$ and $\{Milk, Eggs\}$ to achieve the same result. For a pair of k-itemsets that join to create a $(k+1)$-itemset, it is possible to intersect the *tid* lists of the k-itemset pair to obtain the *tid*-list of the resulting $(k+1)$-itemset. Intersecting *tid* lists of k-itemsets is preferable to intersecting *tid* lists of 1-itemsets because the *tid* lists of k-itemsets are typically smaller than those of 1-itemsets, which makes intersection faster. Such an approach is referred to as *recursive tid* list intersection. This insightful notion of recursive *tid* list intersection was introduced[2] by the *Monet* [273] and *Partition* [446] algorithms. The *Partition* framework [446] proposed a vertical version of the *Apriori* algorithm with *tid* list intersection. The pseudocode of this vertical version of the *Apriori* algorithm is illustrated in Fig. 4.7. The only difference from the horizontal *Apriori* algorithm is the use of recursive *tid* list intersections for counting. While the vertical *Apriori* algorithm is computationally more efficient than horizontal *Apriori*, it is memory-intensive because of the need to store *tid* lists with each itemset. Memory requirements can be reduced with the use of a *partitioned ensemble* in which the database is divided into smaller chunks which are independently processed. This approach reduces the memory requirements at the expense of running-time overheads in terms of postprocessing, and it is discussed in Sect. 4.6.2. For smaller databases, no partitioning needs to be applied. In such cases, the vertical *Apriori* algorithm of Fig. 4.7 is also referred to as *Partition-1*, and it is the progenitor of all modern vertical pattern mining algorithms.

The vertical database representation can, in fact, be used in almost any enumeration-tree algorithm with a growth strategy that is different from the breadth-first method. As in the case of the vertical *Apriori* algorithm, the *tid* lists can be stored with the itemsets (nodes) during the growth of the tree. If the *tid* list of any node P is known, it can be intersected with the *tid* list of a sibling node to determine the support count (and *tid* list) of the corresponding extension of P. This provides an efficient way of performing the counting. By varying the strategy of growing the tree, the memory overhead of storing the *tid* lists can be reduced but not the number of operations. For example, while both breadth-first and depth-first strategies will require exactly the same *tid* list intersections for a particular pair of nodes, the depth-first strategy will have a smaller memory footprint because the *tid* lists need to be stored only at the nodes on the tree-path being explored and their immediate siblings. Reducing the memory footprint is, nevertheless, important because it increases the size of the database that can be processed entirely in core.

Subsequently, many algorithms, such as *Eclat* and *VIPER*, adopted *Partition*'s recursive *tid* list intersection approach. *Eclat* is a lattice-partitioned memory-optimization of the algo-

[2]Strictly speaking, *Monet* is the name of the vertical database, on top of which this (unnamed) algorithm was built.

Algorithm *VerticalApriori*(Transactions: $\mathcal{T}$, Minimum Support: *minsup*)
begin
  $k = 1$;
  $\mathcal{F}_1 = \{$ All Frequent 1-itemsets $\}$;
  Construct vertical *tid* lists of each frequent item;
  while $\mathcal{F}_k$ is not empty **do begin**
    Generate $\mathcal{C}_{k+1}$ by joining itemset-pairs in $\mathcal{F}_k$;
    Prune itemsets from $\mathcal{C}_{k+1}$ that violate downward closure;
    Generate *tid* list of each candidate itemset in $\mathcal{C}_{k+1}$ by intersecting
      tid lists of the itemset-pair in $\mathcal{F}_k$ that was used to create it;
    Determine supports of itemsets in $\mathcal{C}_{k+1}$ using lengths of their *tid* lists;
    $\mathcal{F}_{k+1}$= Frequent itemsets of $\mathcal{C}_{k+1}$ together with their *tid* lists;
    $k = k + 1$;
  end;
  return$(\cup_{i=1}^{k} \mathcal{F}_i)$;
end

Figure 4.7: The vertical *Apriori* algorithm of Savasere et al. [446]

rithm in Fig. 4.7. In *Eclat* [537], an independent *Apriori*-like breadth-first strategy is used on each of the sublattices of itemsets with a common prefix. These groups of itemsets are referred to as equivalence classes. Such an approach can reduce the memory requirements by partitioning the candidate space into groups that are processed independently in conjunction with the relevant vertical lists of their prefixes. This kind of candidate partitioning is similar to parallel versions of *Apriori*, such as the *Candidate Distribution* algorithm [54]. Instead of using the candidate partitioning to distribute various sublattices to different processors, the *Eclat* approach sequentially processes the sublattices one after another to reduce peak memory requirements. Therefore, *Eclat* can avoid the postprocessing overheads associated with Savasere et al.'s *data* partitioning approach, if the database is too large to be processed in core by *Partition-1*, but small enough to be processed in core by *Eclat*. In such cases, *Eclat* is faster than *Partition*. Note that the number of computational operations for support counting in *Partition-1* is fundamentally no different from that of *Eclat* because the *tid* list intersections between any pair of itemsets remain the same. Furthermore, *Eclat* implicitly assumes an upper bound on the database size. This is because it assumes that multiple *tid* lists, each of size at least a fraction *minsup* of the number of database records, fit in main memory. The cumulative memory overhead of the multiple *tid* lists always scales proportionally with database size, whereas the memory overhead of the ensemble-based *Partition* algorithm is independent of database size.

4.4.4 Recursive Suffix-Based Pattern Growth Methods

Enumeration trees are constructed by extending *prefixes* of itemsets that are expressed in a lexicographic order. It is also possible to express some classes of itemset exploration methods recursively with *suffix*-based exploration. Although recursive pattern-growth is often understood as a completely different class of methods, it can be viewed as a special case of the generic enumeration-tree algorithm presented in the previous section. This relationship between recursive pattern-growth methods and enumeration-tree methods will be explored in greater detail in Sect. 4.4.4.5.

Recursive suffix-based pattern growth methods are generally understood in the context of the well-known FP-Tree data structure. While the FP-Tree provides a space- and time-efficient way to implement the recursive pattern exploration, these methods can also be implemented with the use of arrays and pointers. This section will present the recursive pattern growth approach in a simple way without introducing any specific data structure. We also present a number of simplified implementations[3] with various data structures to facilitate better understanding. The idea is to move from the simple to the complex by providing a top-down data structure-agnostic presentation, rather than a tightly integrated presentation with the commonly used FP-Tree data structure. This approach provides a clear understanding of how the search space of patterns is explored and the relational with conventional enumeration tree algorithms.

Consider the transaction database $\mathcal{T}$ which is expressed in terms of only frequent 1-items. It is assumed that a counting pass has already been performed on $\mathcal{T}$ to remove the infrequent items and count the supports of the items. Therefore, the input to the recursive procedure described here is slightly different from the other algorithms discussed in this chapter in which this database pass has not been performed. The items in the database are ordered with decreasing support. This lexicographic ordering is used to define the ordering of items within itemsets and transactions. This ordering is also used to define the notion of prefixes and suffixes of itemsets and transactions. The input to the algorithm is the transaction database $\mathcal{T}$ (expressed in terms of frequent 1-items), a current frequent itemset suffix P, and the minimum support *minsup*. The goal of a recursive call to the algorithm is to determine all the frequent patterns that have the suffix P. Therefore, at the top-level recursive call of the algorithm, the suffix P is empty. At deeper-level recursive calls, the suffix P is not empty. The assumption for deeper-level calls is that $\mathcal{T}$ contains only those transactions from the original database that include the itemset P. Furthermore, each transaction in $\mathcal{T}$ is represented using only those frequent extension items of P that are lexicographically smaller than all items of P. Therefore $\mathcal{T}$ is a *conditional transaction set*, or *projected database* with respect to suffix P. This suffix-based projection is similar to the prefix-based projection in *TreeProjection* and *DepthProject*.

In any given recursive call, the first step is to construct the itemset $P_i = \{i\} \cup P$ by concatenating each item i in the transaction database $\mathcal{T}$ to the beginning of suffix P, and reporting it as frequent. The itemset P_i is frequent because $\mathcal{T}$ is defined in terms of frequent items of the projected database of suffix P. For each item i, it is desired to further extend P_i by using a recursive call with the projected database of the (newly extended) frequent suffix P_i. The projected database for extended suffix P_i is denoted by $\mathcal{T}_i$, and it is created as follows. The first step is to extract all transactions from $\mathcal{T}$ that contain the item i. Because it is desired to extend the suffix P_i backwards, all items that are lexicographically greater than or equal to i are removed from the extracted transactions in $\mathcal{T}_i$. In other words, the part of the transaction occurring lexicographically after (and including) i is not relevant for counting frequent patterns ending in P_i. The frequency of each item in $\mathcal{T}_i$ is counted, and the infrequent items are removed.

It is easy to see that the transaction set $\mathcal{T}_i$ is sufficient to generate all the frequent patterns with P_i as a suffix. The problem of finding all frequent patterns ending in P_i using the transaction set $\mathcal{T}_i$ is an identical but smaller problem than the original one on $\mathcal{T}$. Therefore, the original procedure is called recursively with the smaller projected database $\mathcal{T}_i$ and extended suffix P_i. This procedure is repeated for each item i in $\mathcal{T}$.

[3]Variations of these strategies are actually used in some implementations of these methods. We stress that the simplified versions are not optimized for efficiency but are provided for clarity.

Algorithm *RecursiveSuffixGrowth*(Transactions in terms of frequent 1-items: $\mathcal{T}$, Minimum Support: *minsup*, Current Suffix: P)
begin
  for each item i in $\mathcal{T}$ **do begin**
    report itemset $P_i = \{i\} \cup P$ as frequent;
    Extract all transactions $\mathcal{T}_i$ from $\mathcal{T}$ containing item i;
    Remove all items from $\mathcal{T}_i$ that are lexicographically $\geq i$;
    Remove all infrequent items from $\mathcal{T}_i$;
    if $(\mathcal{T}_i \neq \phi)$ **then** *RecursiveSuffixGrowth*$(\mathcal{T}_i, minsup, P_i)$;
  end
end

Figure 4.8: Generic recursive suffix growth on transaction database expressed in terms of frequent 1-items

The projected transaction set $\mathcal{T}_i$ will become successively smaller at deeper levels of the recursion in terms of the number of items and the number of transactions. As the number of transactions reduces, all items in it will eventually fall below the minimum support, and the resulting projected database (constructed on only the frequent items) will be empty. In such cases, a recursive call with $\mathcal{T}_i$ is not initiated; therefore, this branch of the recursion is not explored. For some data structures, such as the FP-Tree, it is possible to impose stronger boundary conditions to terminate the recursion even earlier. This boundary condition will be discussed in a later section.

The overall recursive approach is presented in Fig. 4.8. While the parameter *minsup* has always been assumed to be a (relative) fractional value in this chapter, it is assumed to be an absolute integer support value in this section and in Fig. 4.8. This deviation from the usual convention ensures consistency of the minimum support value across different recursive calls in which the size of the conditional transaction database reduces.

4.4.4.1 Implementation with Arrays but No Pointers

So, how can the projected database $\mathcal{T}$ be decomposed into the *conditional transaction sets* $\mathcal{T}_1 \ldots \mathcal{T}_d$, corresponding to d different 1-item suffixes? The simplest solution is to use arrays. In this solution, the original transaction database $\mathcal{T}$ and the conditional transaction sets $\mathcal{T}_1 \ldots \mathcal{T}_d$ can be represented in arrays. The transaction database $\mathcal{T}$ may be scanned within the "**for**" loop of Fig. 4.8, and the set $\mathcal{T}_i$ is created from $\mathcal{T}$. The infrequent items from $\mathcal{T}_i$ are removed within the loop. However, it is expensive and wasteful to repeatedly scan the database $\mathcal{T}$ inside a "**for**" loop. One alternative is to extract all projections $\mathcal{T}_i$ of $\mathcal{T}$ corresponding to the different suffix items simultaneously in a single scan of the database just before the "**for**" loop is initiated. On the other hand, the simultaneous creation of many such item-specific projected data sets can be memory-intensive. One way of obtaining an excellent trade-off between computational and storage requirements is by using pointers. This approach is discussed in the next section.

4.4.4.2 Implementation with Pointers but No FP-Tree

The array-based solution either needs to repeatedly scan the database $\mathcal{T}$ or simultaneously create many smaller item-specific databases in a single pass. Typically, the latter achieves

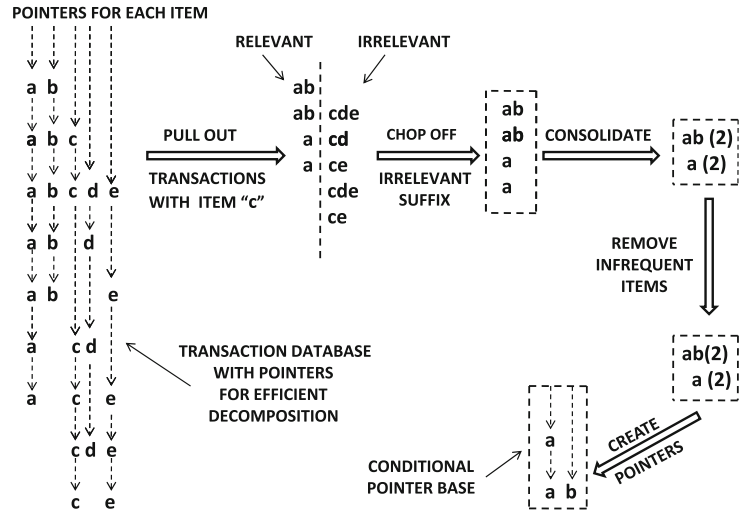


Figure 4.9: Illustration of recursive pattern growth with pointers and no FP-Tree

better efficiency but is more memory-intensive. One simple solution to this dilemma is to set up a data structure in the form of pointers in the first pass, which *implicitly* stores the decomposition of $\mathcal{T}$ into different item-specific data sets at a lower memory cost. This data structure is set up at the time that infrequent items are removed from the transaction database $\mathcal{T}$, and then utilized for extracting different conditional transaction sets $\mathcal{T}_i$ from $\mathcal{T}$. For each item i in $\mathcal{T}$, a pointer threads through the transactions containing that item in lexicographically sorted (dictionary) order. In other words, after arranging the database $\mathcal{T}$ in lexicographically sorted order, each item i in each transaction has a pointer to the same item i in the next transaction that contains it. Because a pointer is required at each item in each transaction, the storage overhead in this case is proportional to that of the original transaction database $\mathcal{T}$. An additional optimization is to consolidate repeated transactions and store them with their counts. An example of a sample database with nine transactions on the five items $\{a, b, c, d, e\}$ is illustrated in Fig. 4.9. It is clear from the figure that there are five sets of pointers, one for each item in the database.

After the pointers have been set up, $\mathcal{T}_i$ is extracted by just "chasing" the pointer thread for item i. The time for doing this is proportional to the number of transactions in $\mathcal{T}_i$. The infrequent items in $\mathcal{T}_i$ are removed, and the pointers for the conditional transaction data need to be reconstructed to create a *conditional pointer base* which is basically the conditional transaction set augmented with pointers. The modified pseudocode with the use of pointers is illustrated in Fig. 4.10. Note that the only difference between the pseudocode of Figs. 4.8 and 4.10 is the setting up of pointers after extraction of conditional transaction sets and the use of these pointers to efficiently extract the conditional transaction data sets $\mathcal{T}_i$. A recursive call is initiated at the next level with the extended suffix $P_i = \{i\} \cup P$, and conditional database $\mathcal{T}_i$.

To illustrate how $\mathcal{T}_i$ can be extracted, an example of a transaction database with 5 items and 9 transactions is illustrated in Fig. 4.9. For simplicity, we use a (raw) minimum support value of 1. The transactions corresponding to the item c are extracted, and the irrelevant suffix including and after item c are removed for further recursive calls. Note that this leads to shorter transactions, some of which are repeated. As a result, the conditional database

Algorithm *RecursiveGrowthPointers*(Transactions in terms of frequent 1-items: $\mathcal{T}$, Minimum Support: *minsup*, Current Suffix: P)
begin
  for each item i in $\mathcal{T}$ **do begin**
    report itemset $P_i = \{i\} \cup P$ as frequent;
    Use pointers to extract all transactions $\mathcal{T}_i$ from $\mathcal{T}$ containing item i;
    Remove all items from $\mathcal{T}_i$ that are lexicographically $\geq i$;
    Remove all infrequent items from $\mathcal{T}_i$;
    Set up pointers for $\mathcal{T}_i$;
    if $(\mathcal{T}_i \neq \phi)$ **then** *RecursiveGrowthPointers*$(\mathcal{T}_i, minsup, P_i)$;
  end
end

Figure 4.10: Generic recursive suffix growth with pointers

for $\mathcal{T}_i$ contains only two distinct transactions after consolidation. The infrequent items from this conditional database need to be removed. No items are removed at a minimum support of 1. Note that if the minimum support had been 3, then the item b would have been removed. The pointers for the new conditional transaction set do need to be set up again because they will be different for the conditional transaction database than in the original transactions. Unlike the pseudocode of Fig. 4.8, an additional step of setting up pointers is included in the pseudocode of Fig. 4.10.

The pointers provide an efficient way to extract the conditional transaction database. Of course, the price for this is that the pointers are a space overhead, with size exactly proportional to the original transaction database $\mathcal{T}$. Consolidating repeated transactions does save some space. The FP-Tree, which will be discussed in the next section, takes this approach one step further by consolidating not only repeated transactions, but also repeated *prefixes* of transactions with the use of a trie data structure. This representation reduces the space-overhead by consolidating prefixes of the transaction database.

4.4.4.3 Implementation with Pointers and FP-Tree

The FP-Tree is designed with the primary goal of space efficiency of the projected database. The FP-Tree is a trie data structure representation of the conditional transaction database by consolidating the prefixes. This trie replaces the array-based implementation of the previous sections, but it retains the pointers. The path from the root to the leaf in the trie represents a (possibly repeated) transaction in the database. The path from the root to an internal node may represent either a transaction or the prefix of a transaction in the database. Each internal node is associated with a count representing the number of transactions in the original database that contain the prefix corresponding to the path from the root to that node. The count on a leaf represents the number of repeated instances of the transaction defined by the path from the root to that leaf. Thus, the FP-Tree maintains all counts of all the repeated transactions as well as their prefixes in the database. As in a standard trie data-structure, the prefixes are sorted in dictionary order. The lexicographic ordering of items is from the most frequent to the least frequent to maximize the advantages of prefix-based compression. This ordering also provides excellent selectivity in reducing the size of various conditional transaction sets in a balanced way. An example of the FP-Tree

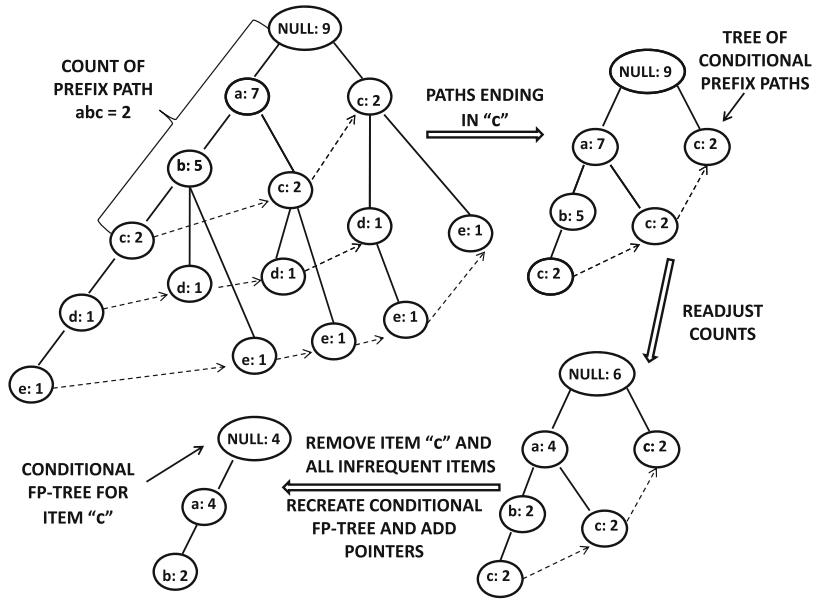


Figure 4.11: Illustration of recursive pattern growth with pointers and FP-Tree

data structure for the same database (as the previous example of Fig. 4.9) is shown in Fig. 4.11. In the example, the number "2" associated with the leftmost item c in the FP-Tree, represents the count of prefix path abc, as illustrated in Fig. 4.11.

The initial FP-Tree $\mathcal{FPT}$ can be constructed as follows. First the infrequent items in the database are removed. The resulting transactions are then successively inserted into the trie. The counts on the overlapping nodes are incremented by 1 when the prefix of the inserted transaction overlaps with an existing path in the trie. For the non-overlapping portion of the transaction, a new path needs to be created containing this portion. The newly created nodes are assigned a count of 1. This process of insertion is identical to that of trie creation, except that counts are also associated with nodes. The resulting tree is a compressed representation because common items in the prefixes of multiple transactions are represented by a single node.

The pointers can be constructed in an analogous way to the simpler array data structure of the previous section. The pointer for each item points to the next occurrence of the same item in the trie. Because a trie stores the transactions in dictionary order, it is easy to create pointers threading each of the items. However, the number of pointers is smaller, because many nodes have been consolidated. As an illustrative example, one can examine the relationship between the array-based data structure of Fig. 4.9, and the FP-Tree in Fig. 4.11. The difference is that the prefixes of the arrays in Fig. 4.9 are consolidated and compressed into a trie in Fig. 4.11.

The conditional FP-Tree $\mathcal{FPT}_i$ (representing the conditional database $\mathcal{T}_i$) needs to be extracted and reorganized for each item $i \in \mathcal{FPT}$. This extraction is required to initiate recursive calls with conditional FP-Trees. As in the case of the simple pointer-based structure of the previous section, it is possible to use the pointers of an item to extract the subset of the projected database containing that item. The following steps need to be performed for extraction of the conditional FP-Tree of item i:

1. The pointers for item i are chased to extract the *tree of conditional prefix paths* for the item. These are the paths from the item to the root. The remaining branches are pruned.
2. The counts of the nodes in the tree of prefix-paths are adjusted to account for the pruned branches. The counts can be adjusted by aggregating the counts on the leaves upwards.
3. The frequency of each item is counted by aggregating the counts over all occurrences of that item in the tree of prefix paths. The items that do not meet the minimum support requirement are removed from the prefix paths. Furthermore, the last item i is also removed from each prefix path. The resulting conditional FP-Tree might have a completely different organization than the extracted tree of prefix-paths because of the removal of infrequent items. Therefore, the conditional FP-Tree may need to be recreated by reinserting the conditional prefix paths obtained after removing infrequent items. The pointers for the conditional FP-Tree need to be reconstructed as well.

Consider the example in Fig. 4.11 which is the same data set as in Fig. 4.9. As in Fig. 4.9, it is possible to follow the pointers for item c in Fig. 4.11 to extract a tree of conditional prefix paths (shown in Fig. 4.11). The counts on many nodes in the tree of conditional prefix paths need to be reduced because many branches from the original FP-Tree (that do not contain the item c) are not included. These reduced counts can be determined by aggregating the counts on the leaves upwards. After removing the item c and infrequent items, two frequency-annotated conditional prefix paths $ab(2)$ and $a(2)$ are obtained, which are identical to the two projected and consolidated transactions of Fig. 4.9. The conditional FP-tree is then constructed for item c by reinserting these two conditional prefix paths into a new conditional FP-Tree. Again, this conditional FP-Tree is a trie representation of the conditional pointer base of Fig. 4.9. In this case, there are no infrequent items because a minimum support of 1 is used. If a minimum support of 3 had been used, then the item b would have to be removed. The resulting conditional FP-Tree is used in the next level recursive call. After extracting the conditional FP-Tree $\mathcal{FPT}_i$, it is checked whether it is empty. An empty conditional FP-Tree could occur when there are no frequent items in the extracted tree of conditional prefix paths. If the tree is not empty, then the next level recursive call is initiated with suffix $P_i = \{i\} \cup P$, and the conditional FP-Tree $\mathcal{FPT}_i$.

The use of the FP-Tree allows an additional optimization in the form of a boundary condition for quickly extracting frequent patterns at deeper levels of the recursion. In particular, it is checked whether all the nodes of the FP-Tree lie on a single path. In such a case, the frequent patterns can be directly extracted from this path by extracting all combinations of nodes on this path together with the aggregated support counts. For example, in the case of Fig. 4.11, all nodes on the conditional FP-Tree lie on a single path. Therefore, in the next recursive call, the bottom of the recursion will be reached. The pseudocode for *FP-growth* is illustrated in Fig. 4.12. This pseudocode is similar to the pointer-based pseudocode of Fig. 4.10, except that a compressed FP-Tree is used.

4.4.4.4 Trade-offs with Different Data Structures

The main advantage of an FP-Tree over pointer-based implementation is one of space compression. The FP-Tree requires less space than pointer-based implementation because of trie-based compression, although it might require more space than an array-based implementation because of the pointer overhead. The precise space requirements depend on the

Algorithm *FP-growth*(FP-Tree of frequent items: $\mathcal{FPT}$, Minimum Support: *minsup*, Current Suffix: P)
begin
  if $\mathcal{FPT}$ is a single path
    then determine all combinations C of nodes on the path, and report $C \cup P$ as frequent;
  else (Case when $\mathcal{FPT}$ is not a single path)
  for each item i in $\mathcal{FPT}$ **do begin**
    report itemset $P_i = \{i\} \cup P$ as frequent;
    Use pointers to extract conditional prefix paths from $\mathcal{FPT}$ containing item i;
    Readjust counts of prefix paths and remove i;
    Remove infrequent items from prefix paths and reconstruct conditional FP-Tree $\mathcal{FPT}_i$;
    if $(\mathcal{FPT}_i \neq \phi)$ **then** *FP-growth*$(\mathcal{FPT}_i, minsup, P_i)$;
  end
end

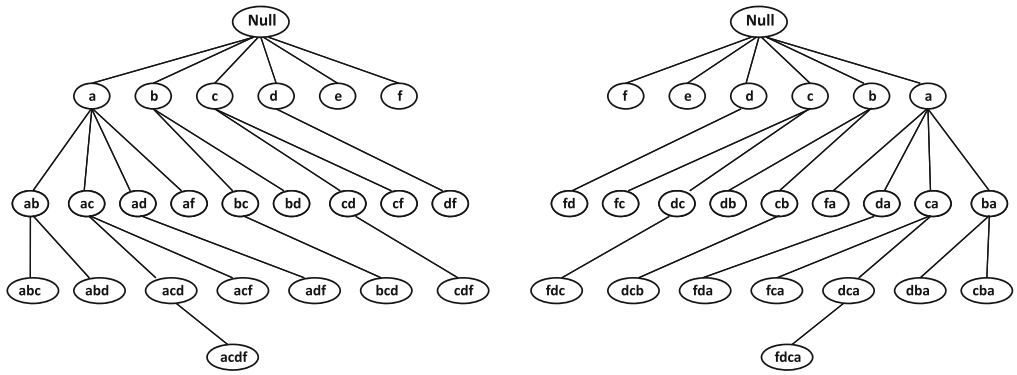
Figure 4.12: The *FP-growth* algorithm with an FP-Tree representation of the transaction database expressed in terms of frequent 1-items

level of consolidation at higher level nodes in the trie-like FP-Tree structure for a particular data set. Different data structures may be more suitable for different data sets.

Because projected databases are repeatedly constructed and scanned during recursive calls, it is crucial to maintain them in main memory. Otherwise, drastic disk-access costs will be incurred by the potentially exponential number of recursive calls. The sizes of the projected databases increase with the original database size. For certain kinds of databases with limited consolidation of repeated transactions, the number of *distinct* transactions in the projected database will always be approximately proportional to the number of transactions in the original database, where the proportionality factor f is equal to the (fractional) minimum support. For databases that are larger than a factor $1/f$ of the main memory availability, projected databases may not fit in main memory either. Therefore, the limiting factor on the use of the approach is the size of the original transaction database. This issue is specific to almost all projection-based methods and vertical counting methods. Memory is always at a premium in such methods and therefore it is crucial for projected transaction data structures to be designed as compactly as possible. As we will discuss later, the *Partition* framework of Savasere et al. [446] provides a partial solution to this issue at the expense of running time.

4.4.4.5 Relationship Between FP-Growth and Enumeration-Tree Methods

FP-growth is popularly believed to be radically different from enumeration-tree methods. This is, in part, because *FP-growth* was originally presented as a method that extracts frequent patterns without candidate generation. However, such an exposition provides an incomplete understanding of how the search space of patterns is explored. *FP-growth* is an instantiation of enumeration-tree methods. All enumeration-tree methods generate candidate extensions to grow the tree. In the following, we will show the equivalence between enumeration-tree methods and *FP-growth*.



(a) Prefix extensions with ordering of a, b, c, d, e, f (Enumeration Tree Prefixes shown)

(b) *FP-growth* with ordering of f, e, d, c, b, a (Recursion Tree Suffixes shown)

Figure 4.13: Enumeration trees are identical to *FP-growth* recursion trees with reverse lexicographic ordering

FP-growth is a recursive algorithm that extends suffixes of frequent patterns. Any recursive approach has a tree-structure associated with it that is referred to as its *recursion tree*, and a dynamic *recursion stack* that stores the recursion variables on the current path of the recursion tree during execution. Therefore, it is instructive to examine the suffix-based recursion tree created by the *FP-growth* algorithm, and compare it with the classical prefix-based enumeration tree used by enumeration-tree algorithms.

In Fig. 4.13a, the enumeration tree from the earlier example of Fig. 4.3 has been replicated. This tree of frequent patterns is counted by all enumeration-tree algorithms along with a single layer of infrequent candidate extensions of this tree corresponding to failed candidate tests. Each call of *FP-growth* discovers the set of frequent patterns extending a particular suffix of items, just as each branch of an enumeration tree explores the itemsets for a particular prefix. So, what is the hierarchical recursive relationship among the suffixes whose conditional pattern bases are explored? First, we need to decide on an ordering of items. Because the recursion is performed on suffixes and enumeration trees are constructed on prefixes, the *opposite* ordering $\{f, e, d, c, b, a\}$ is assumed to adjust for the different convention in the two methods. Indeed, most enumeration-tree methods order items from the least frequent to the most frequent, whereas *FP-growth* does the reverse. The corresponding recursion tree of *FP-growth*, when the 1-itemsets are ordered from left to right in dictionary order, is illustrated in Fig. 4.13b. The trees in Figs 4.13a and 4.13b are identical, with the only difference being that they are drawn differently, to account for the opposite lexicographic ordering. The *FP-growth* recursion tree on the reverse lexicographic ordering has an identical structure to the traditional enumeration tree on the prefixes. During any given recursive call of *FP-growth*, the current (recursion) stack of suffix items is the path in the enumeration tree that is currently being explored. This enumeration tree is explored in depth-first order by *FP-growth* because of its recursive nature.

Traditional enumeration-tree methods typically count the support of a single layer of infrequent extensions of the frequent patterns in the enumeration-tree, as (failed) candidates, to rule them out. Therefore, it is instructive to explore whether *FP-growth* avoids counting these infrequent candidates. Note that when conditional transaction databases $\mathcal{FPT}_i$ are

created (see Fig. 4.12), infrequent items must be removed from them. This requires the counting of the support of these (implicitly failed) *candidate extensions*. In a traditional candidate generate-and-test algorithm, the frequent candidate extensions would be reported immediately after the counting step as a successful candidate test. However, in *FP-growth*, these frequent extensions are encoded back into the conditional transaction database $\mathcal{FPT}_i$, and the reporting is delayed to the next level recursive call. In the next level recursive call, these frequent extensions are then extracted from $\mathcal{FPT}_i$ and reported. The counting and removal of infrequent items from conditional transaction sets is an implicit candidate evaluation and testing step. The number of such failed candidate tests[4] in *FP-growth* is exactly equal to that of enumeration-tree algorithms, such as *Apriori* (without the level-wise pruning step). This equality follows directly from the relationship of all these algorithms to how they explore the enumeration tree and rule out infrequent portions. All pattern-growth methods, including *FP-growth*, should be considered enumeration-tree methods, as should *Apriori*. Whereas traditional enumeration trees are constructed on prefixes, the (implicit) *FP-growth* enumeration trees are constructed using suffixes. This is a difference only in the item-ordering convention.

The depth-first strategy is the approach of choice in database projection methods because it is more memory-efficient to maintain the conditional transaction sets along a (relatively small) depth of the enumeration (recursion) tree rather than along the (much larger) breadth of the enumeration tree. As discussed in the previous section, memory management becomes a problem even with the depth-first strategy beyond a certain database size. However, the specific strategy used for tree exploration does not have any impact on the size of the enumeration tree (or candidates) explored over the course of the entire algorithm execution. The only difference is that breadth-first methods process candidates in large batches based on pattern size, whereas depth-first methods process candidates in smaller batches of immediate siblings in the enumeration tree. From this perspective, *FP-growth* cannot avoid the exponential candidate search space exploration required by enumeration-tree methods, such as *Apriori*.

Whereas methods such as *Apriori* can also be interpreted as counting methods on an enumeration-tree of exactly the same size as the recursion tree of *FP-growth*, the counting work done at the higher levels of the enumeration tree is lost. This loss is because the counting is done from scratch at each level in *Apriori* with the entire transaction database rather than a projected database that remembers and *reuses* the work done at the higher levels of the tree. Projection-based reuse is also utilized by Savasere et al.'s vertical counting methods [446] and *DepthProject*. The use of a pointer-trie combination data structure for projected transaction representation is the primary difference of *FP-growth* from other projection-based methods. In the context of depth-first exploration, these methods can be understood either as divide-and-conquer strategies or as projection-based reuse strategies. The notion of projection-based reuse is more general because it applies to both the breadth-first and depth-first versions of the algorithm, and it provides a clearer picture of how computational savings are achieved by avoiding wasteful and repetitive counting. Projection-based reuse enables the efficient testing of candidate *item extensions* in a restricted portion of the database rather than the testing of candidate *itemsets* in the full database. Therefore, the efficiencies in *FP-growth* are a result of more efficient counting *per candidate* and not because of fewer candidates. The only differences in search space size between various methods are

[4] An *ad hoc* pruning optimization in *FP-growth* terminates the recursion when all nodes in the FP-Tree lie on a single path. This pruning optimization reduces the number of successful candidate tests but not the number of failed candidate tests. Failed candidate tests often dominate successful candidate tests in real data sets.

the result of *ad hoc* pruning optimizations, such as level-wise pruning in *Apriori*, bucketing in the *DepthProject* algorithm, and the single-path boundary condition of *FP-growth*.

The bookkeeping of the projected transaction sets can be done differently with the use of different data structures, such as arrays, pointers, or a pointer-trie combination. Many different data structure variations are explored in different projection algorithms, such as *TreeProjection*, *DepthProject*, *FP-growth*, and *H-Mine* [419]. Each data structure is associated with a different set of efficiencies and overheads.

In conclusion, the enumeration tree[5] is the most general framework to describe all previous frequent pattern mining algorithms. This is because the enumeration tree is a subgraph of the lattice (candidate space) and it provides a way to explore the candidate patterns in a systematic and non-redundant way. The support testing of the frequent portion of the enumeration tree along with a single layer of infrequent candidate extensions of these nodes is fundamental to all frequent itemset mining algorithms for ruling in and ruling out *possible* (or *candidate*) frequent patterns. Any algorithm, such as *FP-growth*, which uses the enumeration tree to rule in and rule out possible extensions of frequent patterns with support counting, is a candidate generate-and-test algorithm.

4.5 Alternative Models: Interesting Patterns

The traditional model for frequent itemset generation has found widespread popularity and acceptance because of its simplicity. The simplicity of using raw frequency counts for the support, and that of using the conditional probabilities for the confidence is very appealing. Furthermore, the downward closure property of frequent itemsets enables the design of efficient algorithms for frequent itemset mining. This algorithmic convenience does not, however, mean that the patterns found are always significant from an *application-specific* perspective. Raw frequencies of itemsets do not always correspond to the most *interesting* patterns.

For example, consider the transaction database illustrated in Fig. 4.1. In this database, *all* the transactions contain the item $Milk$. Therefore, the item $Milk$ can be appended to *any* set of items, without changing its frequency. However, this does not mean that $Milk$ is truly associated with any set of items. Furthermore, for any set of items X, the association rule $X \Rightarrow \{Milk\}$ has 100% confidence. However, it would not make sense for the supermarket merchant to assume that the basket of items X is *discriminatively* indicative of $Milk$. Herein lies the limitation of the traditional support-confidence model.

Sometimes, it is also desirable to design measures that can adjust to the skew in the individual item support values. This adjustment is especially important for negative pattern mining. For example, the support of the pair of items $\{Milk, Butter\}$ is very different from that of $\{\neg Milk, \neg Butter\}$. Here, $\neg$ indicates negation. On the other hand, it can be argued that the statistical coefficient of correlation is exactly the same in both cases. Therefore, the measure should quantify the association between both pairs in exactly the same way. Clearly, such measures are important for negative pattern mining. Measures that satisfy this property are said to satisfy the *bit symmetric* property because values of 0 in the binary matrix are treated in a similar way to values of 1.

[5] *FP-growth* has been presented in a separate section from enumeration tree methods only because it uses a different convention of constructing *suffix*-based enumeration trees. It is not necessary to distinguish "pattern growth" methods from "candidate-based" methods to meaningfully categorize various frequent pattern mining methods. Enumeration tree methods are best categorized on the basis of their (i) tree exploration strategy, (ii) projection-based reuse properties, and (iii) relevant data structures.

Although it is possible to quantify the affinity of sets of items in ways that are statistically more robust than the support-confidence framework, the major computational problem faced by most such *interestingness-based models* is that the downward closure property is generally not satisfied. This makes algorithmic development rather difficult on the exponentially large search space of patterns. In some cases, the measure is defined only for the special case of 2-itemsets. In other cases, it is possible to design more efficient algorithms. The following contains a discussion of some of these models.

4.5.1 Statistical Coefficient of Correlation

A natural statistical measure is the Pearson coefficient of correlation between a pair of items. The Pearson coefficient of correlation between a pair of random variables X and Y is defined as follows:

$$\rho = \frac{E[X \cdot Y] - E[X] \cdot E[Y]}{\sigma(X) \cdot \sigma(Y)}. \tag{4.4}$$

In the case of market basket data, X and Y are binary variables whose values reflect presence or absence of items. The notation $E[X]$ denotes the expectation of X, and $\sigma(X)$ denotes the standard deviation of X. Then, if $sup(i)$ and $sup(j)$ are the relative supports of individual items, and $sup(\{i, j\}$ is the relative support of itemset $\{i, j\}$, then the overall correlation can be estimated from the data as follows:

$$\rho_{ij} = \frac{sup(\{i, j\}) - sup(i) \cdot sup(j)}{\sqrt{sup(i) \cdot sup(j) \cdot (1 - sup(i)) \cdot (1 - sup(j))}}. \tag{4.5}$$

The coefficient of correlation always lies in the range $[-1, 1]$, where the value of $+1$ indicates perfect positive correlation, and the value of -1 indicates perfect negative correlation. A value near 0 indicates weakly correlated data. This measure satisfies the bit symmetric property. While the coefficient of correlation is statistically considered the most robust way of measuring correlations, it is often intuitively hard to interpret when dealing with items of varying but low support values.

4.5.2 χ^2 Measure

The χ^2 measure is another bit-symmetric measure that treats the presence and absence of items in a similar way. Note that for a set of k binary random variables (items), denoted by X, there are 2^k-possible states representing presence or absence of different items of X in the transaction. For example, for $k = 2$ items $\{Bread, Butter\}$, the 2^2 states are $\{Bread, Butter\}$, $\{Bread, \neg Butter\}$, $\{\neg Bread, Butter\}$, and $\{\neg Bread, \neg Butter\}$. The expected fractional presence of each of these combinations can be quantified as the product of the supports of the states (presence or absence) of the individual items. For a given data set, the *observed* value of the support of a state may vary significantly from the expected value of the support. Let O_i and E_i be the observed and expected values of the absolute support of state i. For example, the expected support E_i of $\{Bread, \neg Butter\}$ is given by the total number of transactions multiplied by each of the fractional supports of $Bread$ and $\neg Butter$, respectively. Then, the χ^2-measure for set of items X is defined as follows:

$$\chi^2(X) = \sum_{i=1}^{2^{|X|}} \frac{(O_i - E_i)^2}{E_i}. \tag{4.6}$$

For example, when $X = \{Bread, Butter\}$, one would need to perform the summation in Eq. 4.6 over the $2^2 = 4$ states corresponding to $\{Bread, Butter\}$, $\{Bread, \neg Butter\}$, $\{\neg Bread, Butter\}$, and $\{\neg Bread, \neg Butter\}$. A value that is close to 0 indicates statistical independence among the items. Larger values of this quantity indicate greater dependence between the variables. However, large χ^2 values do not reveal whether the dependence between items is positive or negative. This is because the χ^2 test measures dependence between variables, rather than the nature of the correlation between the specific states of these variables.

The χ^2 measure is bit-symmetric because it treats the presence and absence of items in a similar way. The χ^2-test satisfies the *upward closure property* because of which an efficient algorithm can be devised for discovering interesting k-patterns. On the other hand, the computational complexity of the measure in Eq. 4.6 increases exponentially with $|X|$.

4.5.3 Interest Ratio

The interest ratio is a simple and intuitively interpretable measure. The interest ratio of a set of items $\{i_1 \ldots i_k\}$ is denoted as $I(\{i_1, \ldots i_k\})$, and is defined as follows:

$$I(\{i_1 \ldots i_k\}) = \frac{sup(\{i_1 \ldots i_k\})}{\prod_{j=1}^{k} sup(i_j)}. \tag{4.7}$$

When the items are statistically independent, the joint support in the numerator will be equal to the product of the supports in the denominator. Therefore, an interest ratio of 1 is the break-even point. A value greater than 1 indicates that the variables are positively correlated, whereas a ratio of less than 1 is indicative of negative correlation.

When some items are extremely rare, the interest ratio can be misleading. For example, if an item occurs in only a single transaction in a large transaction database, each item that co-occurs with it in that transaction can be paired with it to create a 2-itemset with a very high interest ratio. This is statistically misleading. Furthermore, because the interest ratio does not satisfy the downward closure property, it is difficult to design efficient algorithms for computing it.

4.5.4 Symmetric Confidence Measures

The traditional confidence measure is *asymmetric* between the antecedent and consequent. However, the support measure is symmetric. Symmetric confidence measures can be used to replace the support-confidence framework with a single measure. Let X and Y be two 1-itemsets. Symmetric confidence measures can be derived as a function of the confidence of $X \Rightarrow Y$ and the confidence of $Y \Rightarrow X$. The various symmetric confidence measures can be any one of the minimum, average, or maximum of these two confidence values. The minimum is not desirable when either X or Y is very infrequent, causing the combined measure to be too low. The maximum is not desirable when either X or Y is very frequent, causing the combined measure to be too high. The average provides the most robust trade-off in many scenarios. The measures can be generalized to k-itemsets by using all k possible individual items in the consequent for computation. Interestingly, the *geometric* mean of the two confidences evaluates to the cosine measure, which is discussed below. The computational problem with symmetric confidence measures is that the relevant itemsets satisfying a specific threshold on the measure do not satisfy the downward closure property.

4.5.5 Cosine Coefficient on Columns

The cosine coefficient is usually applied to the rows to determine the similarity among transactions. However, it can also be applied to the columns, to determine the similarity between items. The cosine coefficient is best computed using the vertical *tid* list representation on the corresponding binary vectors. The cosine value on the binary vectors computes to the following:

$$cosine(i, j) = \frac{sup(\{i, j\})}{\sqrt{sup(i)} \cdot \sqrt{sup(j)}}. \tag{4.8}$$

The numerator can be evaluated as the length of the intersection of the *tid* lists of items i and j. The cosine measure can be viewed as the geometric mean of the confidences of the rules $\{i\} \Rightarrow \{j\}$ and $\{j\} \Rightarrow \{i\}$. Therefore, the cosine is a kind of symmetric confidence measure.

4.5.6 Jaccard Coefficient and the Min-hash Trick

The Jaccard coefficient was introduced in Chap. 3 to measure similarity between sets. The *tid* lists on a column can be viewed as a set, and the Jaccard coefficient between two *tid* lists can be used to compute the similarity. Let S_1 and S_2 be two sets. As discussed in Chap. 3, the Jaccard coefficient $J(S_1, S_2)$ between the two sets can be computed as follows:

$$J(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}. \tag{4.9}$$

The Jaccard coefficient can easily be generalized to multiway sets, as follows:

$$J(S_1 \ldots S_k) = \frac{|\cap S_i|}{|\cup S_i|}. \tag{4.10}$$

When the sets $S_1 \ldots S_k$ correspond to the *tid* lists of k items, the intersection and union of the *tid* lists can be used to determine the numerator and denominator of the aforementioned expression. This provides the Jaccard-based significance for that k-itemset. It is possible to use a minimum threshold on the Jaccard coefficient to determine all the relevant itemsets.

A nice property of Jaccard-based significance is that it satisfies the set-wise monotonicity property. The k-way Jaccard coefficient $J(S_1 \ldots S_k)$ is always no smaller than the $(k+1)$-way Jaccard coefficient $J(S_1 \ldots S_{k+1})$. This is because the numerator of the Jaccard coefficient is monotonically non-increasing with increasing values of k (similar to support), whereas the denominator is monotonically non-decreasing. Therefore, the Jaccard coefficient cannot increase with increasing values of k. Therefore, when a minimum threshold is used on the Jaccard-based significance of an itemset, the resulting itemsets satisfy the downward closure property, as well. This means that most of the traditional algorithms, such as *Apriori* and enumeration tree methods, can be generalized to the Jaccard coefficient quite easily.

It is possible to use sampling to speed up the computation of the Jaccard coefficient further, and transform it to a standard frequent pattern mining problem. This kind of sampling uses hash functions to simulate sorted samples of the data. So, how can the Jaccard coefficient be computed using sorted sampling? Let D be the $n \times d$ binary data matrix representing the n rows and d columns. Without loss of generality, consider the case when the Jaccard coefficient needs to be computed on the first k columns. Suppose one were to sort the rows in D, and pick the first row in which *at least* one of the first k columns in this row has a value of 1 in this column. Then, it is easy to see that the probability of