

# Data Mining: Concepts and Techniques

The Second Edition

Jiawei Han  
Micheline Kamber  
University of Illinois at Urbana-Champaign

Morgan Kaufmann Publishers  
340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA  
<http://www.mkp.com>

©2006 Academic Press All rights reserved  
Printed in the United States of America

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, recording, or otherwise—without the prior written permission of the publisher.

To Y. Dora and Lawrence for your love and encouragement

J.H.

To Erik, Kevan, Kian, and Mikael for your love and inspiration

M.K.

# Foreword

by Jim Gray  
*Microsoft Research*

We are deluged by data—scientific data, medical data, demographic data, financial data, and marketing data. People have no time to look at this data. Human attention has become a precious resource. So, we must find ways to automatically analyze the data, to automatically classify it, to automatically summarize it, to automatically discover and characterize trends in it, and to automatically flag anomalies. This is one of the most active and exciting areas of the database research community. Researchers in areas such as statistics, visualization, artificial intelligence, and machine learning are contributing to this field. The breadth of the field makes it difficult to grasp its extraordinary progress over the last few years.

Jiawei Han and Micheline Kamber have done a wonderful job of organizing and presenting data mining in this very readable textbook. They begin by giving quick introductions to database and data mining concepts with particular emphasis on data analysis. They review the current product offerings by presenting a general framework that covers them all. They then cover in a chapter-by-chapter tour the concepts and techniques that underlie classification, prediction, association, and clustering. These topics are presented with examples, a tour of the best algorithms for each problem class, and pragmatic rules of thumb about when to apply each technique. I found this presentation style to be very readable, and I certainly learned a lot from reading the book. Jiawei Han and Micheline Kamber have been leading contributors to data mining research. This is the text they use with their students to bring them up to speed on the field. The field is evolving very rapidly, but this book is a quick way to learn the basic ideas, and to understand where the field is today. I found it very informative and stimulating, and I expect you will too.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	What Motivated Data Mining? Why Is It Important? . . . . .	1
1.2	So, What Is Data Mining? . . . . .	3
1.3	Data Mining—On What Kind of Data? . . . . .	7
1.3.1	Relational Databases . . . . .	7
1.3.2	Data Warehouses . . . . .	9
1.3.3	Transactional Databases . . . . .	11
1.3.4	Advanced Database Systems and Advanced Database Applications . . . . .	11
1.4	Data Mining Functionalities—What Kinds of Patterns Can Be Mined? . . . . .	15
1.4.1	Concept/Class Description: Characterization and Discrimination . . . . .	16
1.4.2	Mining Frequent Patterns, Associations, and Correlations . . . . .	17
1.4.3	Classification and Prediction . . . . .	18
1.4.4	Cluster Analysis . . . . .	19
1.4.5	Outlier Analysis . . . . .	19
1.4.6	Evolution Analysis . . . . .	20
1.5	Are All of the Patterns Interesting? . . . . .	20
1.6	Classification of Data Mining Systems . . . . .	21
1.7	Data Mining Task Primitives . . . . .	23
1.8	Integration of a Data Mining System with a Database or Data Warehouse System . . . . .	25
1.9	Major Issues in Data Mining . . . . .	26
1.10	Summary . . . . .	28
1.11	Exercises . . . . .	29
1.12	Bibliographic Notes . . . . .	31
<b>2</b>	<b>Data Preprocessing</b>	<b>35</b>
2.1	Why Preprocess the Data? . . . . .	35
2.2	Descriptive Data Summarization . . . . .	38
2.2.1	Measuring the Central Tendency . . . . .	38

2.2.2	Measuring the Dispersion of Data . . . . .	39
2.2.3	Graphic Displays of Basic Descriptive Data Summaries . . . . .	42
2.3	Data Cleaning . . . . .	45
2.3.1	Missing Values . . . . .	46
2.3.2	Noisy Data . . . . .	47
2.3.3	Data Cleaning as a Process . . . . .	48
2.4	Data Integration and Transformation . . . . .	50
2.4.1	Data Integration . . . . .	50
2.4.2	Data Transformation . . . . .	52
2.5	Data Reduction . . . . .	54
2.5.1	Data Cube Aggregation . . . . .	55
2.5.2	Attribute Subset Selection . . . . .	56
2.5.3	Dimensionality Reduction . . . . .	57
2.5.4	Numerosity Reduction . . . . .	60
2.6	Data Discretization and Concept Hierarchy Generation . . . . .	63
2.6.1	Discretization and Concept Hierarchy Generation for Numerical Data . . . . .	65
2.6.2	Concept Hierarchy Generation for Categorical Data . . . . .	69
2.7	Summary . . . . .	71
2.8	Exercises . . . . .	72
2.9	Bibliographic Notes . . . . .	75
<b>3</b>	<b>Data Warehouse and OLAP Technology: An Overview</b>	<b>77</b>
3.1	What Is a Data Warehouse? . . . . .	77
3.1.1	Differences between Operational Database Systems and Data Warehouses . . . . .	79
3.1.2	But, Why Have a Separate Data Warehouse? . . . . .	80
3.2	A Multidimensional Data Model . . . . .	81
3.2.1	From Tables and Spreadsheets to Data Cubes . . . . .	81
3.2.2	Stars, Snowflakes, and Fact Constellations: Schemas for Multidimensional Databases . . . . .	83
3.2.3	Examples for Defining Star, Snowflake, and Fact Constellation Schemas . . . . .	85
3.2.4	Measures: Their Categorization and Computation . . . . .	87
3.2.5	Concept Hierarchies . . . . .	89
3.2.6	OLAP Operations in the Multidimensional Data Model . . . . .	91
3.2.7	A Starnet Query Model for Querying Multidimensional Databases . . . . .	92
3.3	Data Warehouse Architecture . . . . .	93
3.3.1	Steps for the Design and Construction of Data Warehouses . . . . .	93
3.3.2	A Three-Tier Data Warehouse Architecture . . . . .	95
3.3.3	Data Warehouse Back-End Tools and Utilities . . . . .	98

3.3.4	Metadata Repository . . . . .	98
3.3.5	Types of OLAP Servers: ROLAP versus MOLAP versus HOLAP . . . . .	99
3.4	Data Warehouse Implementation . . . . .	100
3.4.1	Efficient Computation of Data Cubes . . . . .	100
3.4.2	Indexing OLAP Data . . . . .	103
3.4.3	Efficient Processing of OLAP Queries . . . . .	105
3.5	From Data Warehousing to Data Mining . . . . .	106
3.5.1	Data Warehouse Usage . . . . .	107
3.5.2	From On-Line Analytical Processing to On-Line Analytical Mining . . . . .	108
3.6	Summary . . . . .	110
3.7	Exercises . . . . .	111
3.8	Bibliographic Notes . . . . .	113
<b>4</b>	<b>Data Cube Computation and Data Generalization</b>	<b>115</b>
4.1	Efficient Methods for Data Cube Computation . . . . .	115
4.1.1	A Road Map for Materialization of Different Kinds of Cubes . . . . .	116
4.1.2	Multiway Array Aggregation for Full Cube Computation . . . . .	120
4.1.3	BUC: Computing Iceberg Cubes from the Apex Cuboid Downwards . . . . .	123
4.1.4	Star-cubing: Computing Iceberg Cubes Using a Dynamic Star-tree Structure . . . . .	126
4.1.5	Precomputing Shell Fragments for Fast High-Dimensional OLAP . . . . .	131
4.1.6	Computing Cubes with Complex Iceberg Conditions . . . . .	137
4.2	Further Development of Data Cube and OLAP Technology . . . . .	138
4.2.1	Discovery-Driven Exploration of Data Cubes . . . . .	138
4.2.2	Complex Aggregation at Multiple Granularity: Multifeature Cubes . . . . .	141
4.2.3	Constrained Gradient Analysis in Data Cubes . . . . .	142
4.3	Attribute-Oriented Induction—An Alternative Method for Data Generalization and Concept Description . . . . .	144
4.3.1	Attribute-Oriented Induction for Data Characterization . . . . .	145
4.3.2	Efficient Implementation of Attribute-Oriented Induction . . . . .	149
4.3.3	Presentation of the Derived Generalization . . . . .	150
4.3.4	Mining Class Comparisons: Discriminating between Different Classes . . . . .	154
4.3.5	Class Description: Presentation of Both Characterization and Comparison . . . . .	157
4.4	Summary . . . . .	159
4.5	Exercises . . . . .	160
4.6	Bibliographic Notes . . . . .	163
<b>5</b>	<b>Mining Frequent Patterns, Associations, and Correlations</b>	<b>165</b>

5.1	Basic Concepts and a Road Map . . . . .	165
5.1.1	Market Basket Analysis: A Motivating Example . . . . .	166
5.1.2	Frequent Itemsets, Closed Itemsets, and Association Rules . . . . .	167
5.1.3	Frequent Pattern Mining: A Road Map . . . . .	169
5.2	Efficient and Scalable Frequent Itemset Mining Methods . . . . .	170
5.2.1	The Apriori Algorithm: Finding Frequent Itemsets Using Candidate Generation . . . . .	171
5.2.2	Generating Association Rules from Frequent Itemsets . . . . .	173
5.2.3	Improving the Efficiency of Apriori . . . . .	174
5.2.4	Mining Frequent Itemsets without Candidate Generation . . . . .	177
5.2.5	Mining Frequent Itemsets Using Vertical Data Format . . . . .	179
5.2.6	Mining Closed Frequent Itemsets . . . . .	181
5.3	Mining Various Kinds of Association Rules . . . . .	182
5.3.1	Mining Multilevel Association Rules . . . . .	182
5.3.2	Mining Multidimensional Association Rules from Relational Databases and Data Warehouses . . . . .	185
5.4	From Association Mining to Correlation Analysis . . . . .	189
5.4.1	Strong Rules Are Not Necessarily Interesting: An Example . . . . .	189
5.4.2	From Association Analysis to Correlation Analysis . . . . .	190
5.5	Constraint-Based Association Mining . . . . .	194
5.5.1	Metarule-Guided Mining of Association Rules . . . . .	194
5.5.2	Constraint Pushing: Mining Guided by Rule Constraints . . . . .	195
5.6	Summary . . . . .	198
5.7	Exercises . . . . .	200
5.8	Bibliographic Notes . . . . .	205
<b>6</b>	<b>Classification and Prediction . . . . .</b>	<b>207</b>
6.1	What Is Classification? What Is Prediction? . . . . .	207
6.2	Issues Regarding Classification and Prediction . . . . .	210
6.2.1	Preparing the Data for Classification and Prediction . . . . .	210
6.2.2	Comparing Classification and Prediction Methods . . . . .	210
6.3	Classification by Decision Tree Induction . . . . .	211
6.3.1	Decision Tree Induction . . . . .	212
6.3.2	Attribute Selection Measures . . . . .	215
6.3.3	Tree Pruning . . . . .	221
6.3.4	Scalability and Decision Tree Induction . . . . .	222
6.4	Bayesian Classification . . . . .	225
6.4.1	Bayes' Theorem . . . . .	226
6.4.2	Naive Bayesian Classification . . . . .	226



6.4.3	Bayesian Belief Networks . . . . .	229
6.4.4	Training Bayesian Belief Networks . . . . .	230
6.5	Rule-Based Classification . . . . .	232
6.5.1	Using IF-THEN Rules For Classification . . . . .	232
6.5.2	Rule Extraction from a Decision Tree . . . . .	233
6.5.3	Rule Extraction from the Training Data . . . . .	234
6.6	Classification by Backpropagation . . . . .	238
6.6.1	A Multilayer Feed-Forward Neural Network . . . . .	239
6.6.2	Defining a Network Topology . . . . .	239
6.6.3	Backpropagation . . . . .	240
6.6.4	Inside the Black Box: Backpropagation and Interpretability . . . . .	244
6.7	Support Vector Machines . . . . .	246
6.7.1	The Case When the Data are Linearly Separable . . . . .	246
6.7.2	The Case When the Data Are Linearly Inseparable . . . . .	249
6.8	Associative Classification: Classification by Association Rule Analysis . . . . .	252
6.9	Lazy Learners (or Learning from Your Neighbors) . . . . .	254
6.9.1	$k$ -Nearest Neighbor Classifiers . . . . .	254
6.9.2	Case-Based Reasoning . . . . .	255
6.10	Other Classification Methods . . . . .	256
6.10.1	Genetic Algorithms . . . . .	256
6.10.2	Rough Set Approach . . . . .	257
6.10.3	Fuzzy Set Approaches . . . . .	257
6.11	Prediction . . . . .	259
6.11.1	Linear Regression . . . . .	259
6.11.2	Nonlinear Regression . . . . .	261
6.11.3	Other Regression-Based Methods . . . . .	262
6.12	Accuracy and Error Measures . . . . .	262
6.12.1	Classifier Accuracy Measures . . . . .	263
6.12.2	Predictor Error Measures . . . . .	265
6.13	Evaluating the Accuracy of a Classifier or Predictor . . . . .	266
6.13.1	Holdout Method and Random Subsampling . . . . .	266
6.13.2	Cross-validation . . . . .	267
6.13.3	Bootstrap . . . . .	267
6.14	Ensemble Methods—Increasing the Accuracy . . . . .	268
6.14.1	Bagging . . . . .	268
6.14.2	Boosting . . . . .	269

6.15	Model Selection . . . . .	271
6.15.1	Estimating Confidence Intervals . . . . .	271
6.15.2	ROC Curves . . . . .	272
6.16	Summary . . . . .	273
6.17	Exercises . . . . .	275
6.18	Bibliographic Notes . . . . .	277
<b>7</b>	<b>Cluster Analysis</b>	<b>281</b>
7.1	What Is Cluster Analysis? . . . . .	281
7.2	Types of Data in Cluster Analysis . . . . .	283
7.2.1	Interval-Scaled Variables . . . . .	284
7.2.2	Binary Variables . . . . .	286
7.2.3	Categorical, Ordinal, and Ratio-Scaled Variables . . . . .	287
7.2.4	Variables of Mixed Types . . . . .	290
7.2.5	Vector objects . . . . .	291
7.3	A Categorization of Major Clustering Methods . . . . .	292
7.4	Partitioning Methods . . . . .	294
7.4.1	Classical Partitioning Methods: $k$ -Means and $k$ -Medoids . . . . .	294
7.4.2	Partitioning Methods in Large Databases: From $k$ -Medoids to CLARANS . . . . .	298
7.5	Hierarchical Methods . . . . .	299
7.5.1	Agglomerative and Divisive Hierarchical Clustering . . . . .	299
7.5.2	BIRCH: Balanced Iterative Reducing and Clustering Using Hierarchies . . . . .	302
7.5.3	ROCK: A Hierarchical Clustering Algorithm for Categorical Attributes . . . . .	304
7.5.4	Chameleon: A Hierarchical Clustering Algorithm Using Dynamic Modeling . . . . .	305
7.6	Density-Based Methods . . . . .	306
7.6.1	DBSCAN: A Density-Based Clustering Method Based on Connected Regions with Sufficiently High Density . . . . .	306
7.6.2	OPTICS: Ordering Points To Identify the Clustering Structure . . . . .	308
7.6.3	DENCLUE: Clustering Based on Density Distribution Functions . . . . .	309
7.7	Grid-Based Methods . . . . .	312
7.7.1	STING: STatistical INformation Grid . . . . .	312
7.7.2	WaveCluster: Clustering Using Wavelet Transformation . . . . .	313
7.8	Model-Based Clustering Methods . . . . .	314
7.8.1	Expectation-Maximization . . . . .	315
7.8.2	Conceptual Clustering . . . . .	316
7.8.3	Neural Network Approach . . . . .	318
7.9	Clustering High-Dimensional Data . . . . .	319

7.9.1	CLIQUE: A Dimension-Growth Subspace Clustering Method . . . . .	320
7.9.2	PROCLUS: A Dimension-Reduction Subspace Clustering Method . . . . .	322
7.9.3	Frequent Pattern-Based Clustering Methods . . . . .	322
7.10	Constraint-Based Cluster Analysis . . . . .	325
7.10.1	Clustering with Obstacle Objects . . . . .	327
7.10.2	User-Constrained Cluster Analysis . . . . .	329
7.10.3	Semi-Supervised Cluster Analysis . . . . .	329
7.11	Outlier Analysis . . . . .	331
7.11.1	Statistical Distribution-Based Outlier Detection . . . . .	331
7.11.2	Distance-Based Outlier Detection . . . . .	333
7.11.3	Density-Based Local Outlier Detection . . . . .	334
7.11.4	Deviation-Based Outlier Detection . . . . .	335
7.12	Summary . . . . .	337
7.13	Exercises . . . . .	338
7.14	Bibliographic Notes . . . . .	341
<b>8</b>	<b>Mining Stream, Time-Series, and Sequence Data</b>	<b>343</b>
8.1	Mining Data Streams . . . . .	344
8.1.1	Methodologies for Stream Data Processing and Stream Data Systems . . . . .	344
8.1.2	Stream OLAP and Stream Data Cubes . . . . .	348
8.1.3	Frequent-Pattern Mining in Data Streams . . . . .	351
8.1.4	Classification of Dynamic Data Streams . . . . .	353
8.1.5	Clustering Evolving Data Streams . . . . .	356
8.2	Mining Time-Series Data . . . . .	359
8.2.1	Trend Analysis . . . . .	359
8.2.2	Similarity Search in Time-Series Analysis . . . . .	361
8.3	Mining Sequence Patterns in Transactional Databases . . . . .	365
8.3.1	Sequential Pattern Mining: Concepts and Primitives . . . . .	365
8.3.2	Scalable Methods for Mining Sequential Patterns . . . . .	366
8.3.3	Constraint-Based Mining of Sequential Patterns . . . . .	373
8.3.4	Periodicity Analysis for Time-Related Sequence Data . . . . .	375
8.4	Mining Sequence Patterns in Biological Data . . . . .	376
8.4.1	Alignment of Biological Sequences . . . . .	376
8.4.2	Hidden Markov Model for Biological Sequence Analysis . . . . .	379
8.5	Summary . . . . .	386
8.6	Exercises . . . . .	387
8.7	Bibliographic Notes . . . . .	389

<b>9</b>	<b>Graph Mining, Social Network Analysis, and Multi-Relational Data Mining</b>	<b>393</b>
9.1	Graph Mining . . . . .	393
9.1.1	Methods for Mining Frequent Subgraphs . . . . .	394
9.1.2	Mining Variant and Constrained Substructure Patterns . . . . .	400
9.1.3	Applications: Graph Indexing, Similarity Search, Classification and Clustering . . . . .	404
9.2	Social Network Analysis . . . . .	407
9.2.1	What is a Social Network? . . . . .	407
9.2.2	Characteristics of Social Networks . . . . .	409
9.2.3	Link Mining: Tasks and Challenges . . . . .	411
9.2.4	Mining on Social Networks . . . . .	414
9.3	Multi-Relational Data Mining . . . . .	418
9.3.1	What Is Multi-Relational Data Mining? . . . . .	419
9.3.2	ILP Approach to Multi-relational Classification . . . . .	420
9.3.3	Tuple ID Propagation . . . . .	421
9.3.4	Multi-Relational Classification Using Tuple ID Propagation . . . . .	422
9.3.5	Multi-Relational Clustering with User Guidance . . . . .	424
9.4	Summary . . . . .	427
9.5	Exercises . . . . .	428
9.6	Bibliographic Notes . . . . .	429
<b>10</b>	<b>Mining Object, Spatial, Multimedia, Text, and Web Data</b>	<b>431</b>
10.1	Multidimensional Analysis and Descriptive Mining of Complex Data Objects . . . . .	431
10.1.1	Generalization of Structured Data . . . . .	432
10.1.2	Aggregation and Approximation in Spatial and Multimedia Data Generalization . . . . .	433
10.1.3	Generalization of Object Identifiers and Class/Subclass Hierarchies . . . . .	433
10.1.4	Generalization of Class Composition Hierarchies . . . . .	434
10.1.5	Construction and Mining of Object Cubes . . . . .	434
10.1.6	Generalization-Based Mining of Plan Databases by Divide-and-Conquer . . . . .	435
10.2	Spatial Data Mining . . . . .	438
10.2.1	Spatial Data Cube Construction and Spatial OLAP . . . . .	438
10.2.2	Mining Spatial Association and Co-location Patterns . . . . .	441
10.2.3	Spatial Clustering Methods . . . . .	442
10.2.4	Spatial Classification and Spatial Trend Analysis . . . . .	442
10.2.5	Mining Raster Databases . . . . .	443
10.3	Multimedia Data Mining . . . . .	443
10.3.1	Similarity Search in Multimedia Data . . . . .	443
10.3.2	Multidimensional Analysis of Multimedia Data . . . . .	444

10.3.3	Classification and Prediction Analysis of Multimedia Data . . . . .	446
10.3.4	Mining Associations in Multimedia Data . . . . .	446
10.3.5	Audio and Video Data Mining . . . . .	447
10.4	Text Mining . . . . .	448
10.4.1	Text Data Analysis and Information Retrieval . . . . .	448
10.4.2	Dimensionality Reduction for Text . . . . .	453
10.4.3	Text Mining Approaches . . . . .	455
10.5	Mining the World Wide Web . . . . .	458
10.5.1	Mining the Web Page Layout Structure . . . . .	460
10.5.2	Mining the Web's Link Structures to Identify Authoritative Web Pages . . . . .	461
10.5.3	Mining Multimedia Data on the Web . . . . .	464
10.5.4	Automatic Classification of Web Documents . . . . .	466
10.5.5	Web Usage Mining . . . . .	466
10.6	Summary . . . . .	468
10.7	Exercises . . . . .	468
10.8	Bibliographic Notes . . . . .	471
<b>11</b>	<b>Applications and Trends in Data Mining</b>	<b>475</b>
11.1	Data Mining Applications . . . . .	475
11.1.1	Data Mining for Financial Data Analysis . . . . .	475
11.1.2	Data Mining for the Retail Industry . . . . .	476
11.1.3	Data Mining for the Telecommunication Industry . . . . .	477
11.1.4	Data Mining for Biological Data Analysis . . . . .	478
11.1.5	Data Mining in Other Scientific Applications . . . . .	480
11.1.6	Data Mining for Intrusion Detection . . . . .	481
11.2	Data Mining System Products and Research Prototypes . . . . .	482
11.2.1	How to Choose a Data Mining System . . . . .	483
11.2.2	Examples of Commercial Data Mining Systems . . . . .	484
11.3	Additional Themes on Data Mining . . . . .	486
11.3.1	Theoretical Foundations of Data Mining . . . . .	486
11.3.2	Statistical Data Mining . . . . .	487
11.3.3	Visual and Audio Data Mining . . . . .	488
11.3.4	Data Mining and Collaborative Filtering . . . . .	489
11.4	Social Impacts of Data Mining . . . . .	491
11.4.1	Ubiquitous and Invisible Data Mining . . . . .	491
11.4.2	Data Mining, Privacy, and Data Security . . . . .	495
11.5	Trends in Data Mining . . . . .	498

11.6 Summary . . . . .	499
11.7 Exercises . . . . .	500
11.8 Bibliographic Notes . . . . .	502
<b>A An Introduction to Microsoft's OLE DB for Data Mining</b>	<b>505</b>

# Preface

Our capabilities of both generating and collecting data have been increasing rapidly. Contributing factors include the computerization of business, scientific, and government transactions; the widespread use of digital cameras, publication tools, and bar codes for most commercial products; and advances in data collection tools ranging from scanned text and image platforms to satellite remote sensing systems. In addition, popular use of the World Wide Web as a global information system has flooded us with a tremendous amount of data and information. This explosive growth in stored or transient data has generated an urgent need for new techniques and automated tools that can intelligently assist us in transforming the vast amounts of data into useful information and knowledge.

This book explores the concepts and techniques of *data mining*, a promising and flourishing frontier in data and information systems and their applications. Data mining, also popularly referred to as *knowledge discovery from data (KDD)*, is the automated or convenient extraction of patterns representing knowledge implicitly stored or catchable in large databases, data warehouses, the Web, other massive information repositories, or data streams.

Data mining is a multidisciplinary field, drawing work from areas including database technology, machine learning, statistics, pattern recognition, information retrieval, neural networks, knowledge-based systems, artificial intelligence, high-performance computing, and data visualization. We present techniques for the discovery of patterns hidden *in large data sets*, focusing on issues relating to their feasibility, usefulness, effectiveness, and scalability. As a result, this book is not intended as an introduction to database systems, machine learning, statistics, or other such areas, although we do provide the background necessary in these areas in order to facilitate the reader's comprehension of their respective roles in data mining. Rather, the book is a comprehensive introduction to data mining, presented with the effectiveness and scalability issues in focus. It should be useful for computing science students, application developers, and business professionals, as well as researchers involved in any of the disciplines listed above.

Data mining emerged during the late 1980s, made great strides during the 1990s, and continues to flourish into the new millennium. This book presents an overall picture of the field, introducing interesting data mining techniques and systems, and discussing applications and research directions. An important motivation for writing this book was the need to build an organized framework for the study of data mining—a challenging task owing to the extensive multidisciplinary nature of this fast developing field. We hope that this book will encourage people with different backgrounds and experiences to exchange their views regarding data mining so as to contribute toward the further promotion and shaping of this exciting and dynamic field.

## Organization of the book

Since the publication of the first edition of this book, great progress has been made in the field of data mining. Many new data mining methods, systems, and applications have been developed. This new edition substantially revises the first edition of the book, with numerous enhancements and a reorganization of the technical contents of the entire book. In addition, several new chapters are included to address recent developments on mining complex types of data, including stream data, sequence data, graph structured data, social network data and multi-relational data.

The chapters are described briefly as follows, with emphasis on the new material.

**Chapter 1** provides an introduction to the multidisciplinary field of data mining. It discusses the evolutionary path of database technology, which has led to the need for data mining, and the importance of its applications. It examines the types of data to be mined, including relational, transactional, and data warehouse data, as well as complex types of data such as data streams, time-series, sequences, graphs, social networks, multi-relational data, spatiotemporal data, multimedia data, text data and Web data. The chapter presents a general classification of data mining tasks, based on the different kinds of knowledge to be mined. In comparison with the first edition, two new sections are introduced: Section 1.7 is on data mining primitives, which allow users to interactively communicate with data mining systems in order to direct the mining process, and Section 1.8 discusses the issues regarding how to integrate a data mining system with a database or data warehouse system. These two sections represent the condensed materials of Chapter 4 “*Data Mining Primitives, Languages and Architectures*” in the first edition. Finally, major challenges in the field are discussed.

**Chapter 2** introduces techniques for preprocessing the data prior to mining. This corresponds to Chapter 3 of the first edition. Since data preprocessing precedes the construction of data warehouses, we address this topic here, and then follow with an introduction to data warehouses in the subsequent chapter. This chapter describes various statistical methods for descriptive data summarization, including measuring both central tendency and dispersion of data. The description of data cleaning methods has been enhanced. Methods for data integration and transformation, and data reduction are discussed, including the use of concept hierarchies for dynamic and static discretization. The automatic generation of concept hierarchies is also described.

Chapters 3 and 4 provides a solid introduction to data warehouse, OLAP (On-Line Analytical Processing), and data generalization. These two chapters correspond to Chapters 2 and 5 of the first edition, but with substantial enhancement regarding data warehouse implementation methods. **Chapter 3** introduces the basic concepts, architectures and general implementations of data warehouse and on-line analytical processing, as well as the relationship between data warehousing and data mining. **Chapter 4** takes a more in-depth look at data warehouse and OLAP technology, presenting a detailed study of methods of data cube computation, including the recently developed star-cubing and high-dimensional OLAP methods. Further exploration of data warehouse and OLAP are discussed, such as discovery-driven cube exploration, multifeature cubes for complex data mining queries, and cube gradient analysis. Attribute-oriented induction, an alternative method for data generalization and concept description, is also discussed.

**Chapter 5** presents methods for mining frequent patterns, associations, and correlations in transactional and relational databases and data warehouses. In addition to introducing the basic concepts, such as market basket analysis, many techniques for frequent itemset mining are presented in an organized way. These range from the basic Apriori algorithm and its variations, to more advanced methods that improve on efficiency, including the frequent-pattern growth approach, frequent-pattern mining with vertical data format, and mining closed frequent itemsets. The chapter also presents techniques for mining multilevel association rules, multidimensional association rules, and quantitative association rules. In comparison with the previous edition, this chapter has placed greater emphasis on the generation of meaningful association and correlation rules. Strategies for constraint-based mining and the use of interestingness measures to focus the rule search are also described.

**Chapter 6** describes methods for data classification and prediction, including decision tree induction, Bayesian classification, rule-based classification, the neural network technique of backpropagation, support vector machines, associative classification,  $k$ -nearest neighbor classifiers, case-based reasoning, genetic algorithms, rough set theory, and fuzzy set approaches. Methods of regression are introduced. Issues regarding accuracy and how to choose the best classifier or predictor are discussed. In comparison with the corresponding chapter in the first edition, the sections on rule-based classification and support vector machines are new, and the discussion of measuring and enhancing classification and prediction accuracy has been greatly expanded.

Cluster analysis forms the topic of **Chapter 7**. Several major data clustering approaches are presented, including partitioning methods, hierarchical methods, density-based methods, grid-based methods, and model-based methods. New sections in this edition introduce techniques for clustering high-dimensional data, as well as for constraint-based cluster analysis. Outlier analysis is also discussed.

Chapters 8 to 10 treat advanced topics in data mining and cover a large body of materials on recent progress in this frontier. These three chapters now replace our previous single chapter on advanced topics. **Chapter 8**



focuses on the mining of stream data, time-series data, and sequence data (covering both transactional sequences and biological sequences). The basic data mining techniques (such as frequent pattern mining, classification, clustering, and constraint-based mining) are extended for these types of data. **Chapter 9** discusses methods for graph and structural pattern mining, social network analysis, and multi-relational data mining. **Chapter 10** presents methods for mining object, spatial, multimedia, text, and Web data, which cover a great deal of new progress in these areas.

Finally, in **Chapter 11**, we summarize the concepts presented in this book and discuss applications and trends in data mining. New material has been added on data mining for biological and biomedical data analysis, other scientific applications, intrusion detection, and collaborative filtering. Social impacts of data mining, such as privacy and data security issues, are discussed, in addition to challenging research issues. Further discussion of ubiquitous data mining has also been added.

The **Appendix** provides an introduction to Microsoft's OLE DB for Data Mining (OLEDB for DM).

Throughout the text, italic font is used to emphasize terms that are defined, while bold font is used to highlight or summarize main ideas. Sans serif font is used for reserved words and system names.

This book has several strong features that set it apart from other texts on data mining. It presents a very broad yet in-depth coverage from the spectrum of data mining, especially regarding several recent research topics on data stream mining, graph mining, social network analysis and multi-relational data mining. The chapters preceding the advanced topics are written to be as self-contained as possible, so they may be read in order of interest by the reader. All of the major methods of data mining are presented. Because we take a database point of view to data mining, the book also presents many important topics in data mining, such as scalable algorithms, and multidimensional OLAP analysis, that are often overlooked or minimally treated in other books.

## To the Instructor

This book is designed to give a broad, yet detailed overview of the field of data mining. It can be used to teach an *introductory* course on data mining at an advanced undergraduate level, or at the first-year graduate level. In addition, it can also be used to teach an *advanced* course on data mining.

If you plan to use the book to teach an introductory course, you may find that the materials in Chapters 1 to 7 are essential, among which Chapter 4 may be omitted if you do not plan to cover the implementation methods for data cubing and online analytical processing in depth. Alternatively, you may omit some sections in Chapters 1 to 7 and use Chapter 11 as the final coverage of applications and trends on data mining.

If you plan to use the book to teach an advanced course on data mining, you may use Chapters 8 through 11. Moreover, additional materials and some recent research papers may supplement selected themes from among the advanced topics of these chapters.

Individual chapters in this book can also be used for tutorials or for special topics in related courses, such as database systems, machine learning, pattern recognition, and intelligent data analysis.

Each chapter ends with a set of exercises, suitable as assigned homework. The exercises are either short questions that test basic mastery of the material covered, longer questions that require analytical thinking, or implementation projects. Some exercises can also be used as research discussion topics. The bibliographic notes at the end of each chapter can be used to find the research literature that contains the origin of the concepts and methods presented, in-depth treatment of related topics, and possible extensions. Extensive teaching aids are available from the book's websites, such as lecture slides, reading lists, and course syllabi, as mentioned above.

## To the Student

We hope that this textbook will spark your interest in this young, yet fast evolving field of data mining. We have attempted to present the material in a clear manner, with careful explanation of the topics covered. Each chapter



- **Links to data mining data sets and software.** We will provide a set of links to the data mining data sets and some sites containing interesting data mining software packages.
- **Sample assignments, exams, course projects.** A set of sample assignments, exams, and course projects will be made available to instructors from the publisher's website.
- **Table of Contents of the book in PDF.**
- **Errata on the different printings of the book.** We welcome you to point out any errors in the book. Once the error is confirmed, we will update this errata list, associated with the acknowledgement of your contribution.

Comments or suggestions can be sent to [hanj@cs.uiuc.edu](mailto:hanj@cs.uiuc.edu). We would be happy to hear from you.

## Acknowledgments for the first edition of the book

We would like to express our sincere thanks to all those who have worked or are currently working with us on data mining related research and/or the DBMiner project, or have provided us with various support in data mining. These include Rakesh Agrawal, Stella Atkins, Yvan Bedard, Binay Bhattacharya, (Yandong) Dora Cai, Nick Cercone, Surajit Chaudhuri, Sonny H. S. Chee, Jianping Chen, Ming-Syan Chen, Qing Chen, Qiming Chen, Shan Cheng, David Cheung, Shi Cong, Son Dao, Umeshwar Dayal, James Delgrande, Guozhu Dong, Carole Edwards, Max Egenhofer, Martin Ester, Usama Fayyad, Ling Feng, Ada Fu, Yongjian Fu, Daphne Gelbart, Randy Goebel, Jim Gray, Robert Grossman, Wan Gong, Yike Guo, Eli Hagen, Howard Hamilton, Jing He, Larry Henschen, Jean Hou, Mei-Chun Hsu, Kan Hu, Haiming Huang, Yue Huang, Julia Itskevitch, Wen Jin, Tiko Kameda, Hiroyuki Kawano, Rizwan Kheraj, Eddie Kim, Won Kim, Krzysztof Koperski, Hans-Peter Kriegel, Vipin Kumar, Laks V.S. Lakshmanan, Joyce Man Lam, James Lau, Deyi Li, George (Wenmin) Li, Jin Li, Ze-Nian Li, Nancy Liao, Gang Liu, Junqiang Liu, Ling Liu, Alan (Yijun) Lu, Hongjun Lu, Tong Lu, Wei Lu, Xuebin Lu, Wo-Shun Luk, Heikki Mannila, Runying Mao, Abhay Mehta, Gabor Melli, Alberto Mendelzon, Tim Merrett, Harvey Miller, Drew Miners, Behzad Mortazavi-Asl, Richard Muntz, Raymond T. Ng, Vicent Ng, Shojiro Nishio, Beng-Chin Ooi, Tamer Ozsü, Jian Pei, Gregory Piatetsky-Shapiro, Helen Pinto, Fred Popowich, Aymn Mohamed Rajan, Peter Scheuermann, Shashi Shekhar, Wei-Min Shen, Avi Silberschatz, Evangelos Simoudis, Nebojsa Stefanovic, Yin Jenny Tam, Simon Tang, Zhaohui Tang, Dick Tsur, Anthony K. H. Tung, Ke Wang, Wei Wang, Zhaoxia Wang, Tony Wind, Lara Winstone, Ju Wu, Betty (Bin) Xia, Cindy M. Xin, Xiaowei Xu, Qiang Yang, Yiwen Yin, Clement Yu, Jeffrey Yu, Philip S. Yu, Osmar R. Zaiane, Carlo Zaniolo, Shuhua Zhang, Zhong Zhang, Yvonne Zheng, Xiaofang Zhou, and Hua Zhu. We are also grateful to Jean Hou, Helen Pinto, Lara Winstone, and Hua Zhu for their help with some of the original figures in this book, and to Eugene Belchev for his careful proofreading of each chapter.

We also wish to thank Diane Cerra, our Executive Editor at Morgan Kaufmann Publishers, for her enthusiasm, patience, and support during our writing of this book, as well as Howard Severson, our Production Editor, and his staff for their conscientious efforts regarding production. We are indebted to all of the reviewers for their invaluable feedback. Finally, we thank our families for their wholehearted support throughout this project.

## Acknowledgments for the second edition of the book

We would like to express our grateful thanks to all the previous and current members of the Data Mining Group at UIUC, the faculty and students in the Data and Information Systems (DAIS) Laboratory in the Department of Computer Science, the University of Illinois at Urbana-Champaign, and many friends and colleagues, whose constant support and encouragement have made our work on this edition a rewarding experience. These include Gul Agha, Rakesh Agrawal, Loretta Auvil, Peter Bajcsy, Geneva Belford, Deng Cai, Y. Dora Cai, Roy Cambell, Kevin C.-C. Chang, Surajit Chaudhuri, Chen Chen, Yixin Chen, Hong Cheng, David Cheung, Shengnan Cong, Gerald DeJong, AnHai Doan, Guozhu Dong, Charis Ermopoulos, Martin Ester, Christos Faloutsos, Wei Fan, Ada Fu, Michael Garland, Johannes Gehrke, Hector Gonzalez, Mehdi Harandi, Thomas Huang, Wen Jin, Sangkyum

Kim, Won Kim, Won-Young Kim, David Kuck, Young-Koo Lee, Harris Lewin, Xiaolei Li, Yifan Li, Chao Liu, Han Liu, Hongyan Liu, Lei Liu, Ying Lu, Klara Nahrstedt, David Padua, Jian Pei, Lenny Pitt, Daniel Reed, Dan Roth, Bruce Schatz, Zheng Shao, Marc Snir, Bhavani M. Thuraisingham, Josep Torrellas, Peter Tzvetkov, Benjamin W. Wah, Haixun Wang, Ke Wang, Jianyong Wang, Wei Wang, Michael Welge, Marianne Winslett, Ouri Wolfson, Andrew Wu, Dong Xin, Xifeng Yan, Jiong Yang, Xiaoxin Yin, Hwanjo Yu, Jeffrey X. Yu, Philip S. Yu, Maria Zemankova, ChengXiang Zhai, Yuanyuan Zhou, and Wei Zou. Deng Cai and ChengXiang Zhai have contributed to the text mining and Web mining sections, Xifeng Yan to the graph mining section, and Xiaoxin Yin to the multi-relational data mining section. Charis Ermopoulos, Hector Gonzalez, Sangkyum Kim, Chao Liu, Hongyan Liu, Xifeng Yan, and Xiaoxin Yin have contributed to the proofreading of the individual chapters of the manuscript.

We also wish to thank Diane Cerra, our Executive Editor at Morgan Kaufmann Publishers, for her enthusiasm, patience, and support during our writing of this book. We are indebted to all of the reviewers for their invaluable feedback. Finally, we thank our families for their wholehearted support throughout this project.

# Chapter 1

## Introduction

This book is an introduction to a young and promising field, called *data mining* and *knowledge discovery from data*. The material in this book is presented from a database perspective, where emphasis is placed on basic data mining concepts and techniques for uncovering interesting data patterns hidden in *large data sets*. The implementation methods discussed are particularly oriented towards the development of *scalable* and *efficient* data mining tools. In this chapter, you will learn how data mining is part of the natural evolution of database technology, why data mining is important, and how it is defined. You will learn about the general architecture of data mining systems, as well as gain insight into the kinds of data on which mining can be performed, the types of patterns that can be found, and how to tell which patterns represent useful knowledge. You will study data mining primitives, from which data mining query languages can be designed. Issues regarding how to integrate a data mining system with a database or data warehouse are also discussed. In addition to studying a classification of data mining systems, you will read about challenging research issues for building data mining tools of the future.

### 1.1 What Motivated Data Mining? Why Is It Important?

Necessity is the mother of invention.—English proverb.

The major reason that data mining has attracted a great deal of attention in the information industry and in society as a whole in recent years is due to the wide availability of huge amounts of data and the imminent need for turning such data into useful information and knowledge. The information and knowledge gained can be used for applications ranging from market analysis, fraud detection, and customer retention, to production control and science exploration.

Data mining can be viewed as a result of the natural evolution of information technology. The database system industry has witnessed an evolutionary path in the development of the following functionalities (Figure 1.1): *data collection and database creation*, *data management* (including data storage and retrieval, and database transaction processing), and *advanced data analysis* (involving data warehousing and data mining). For instance, the early development of data collection and database creation mechanisms served as a prerequisite for later development of effective mechanisms for data storage and retrieval, and query and transaction processing. With numerous database systems offering query and transaction processing as common practice, advanced data analysis has naturally become the next target.

Since the 1960s, database and information technology has been evolving systematically from primitive file processing systems to sophisticated and powerful database systems. The research and development in database systems since the 1970s has progressed from early hierarchical and network database systems to the development of relational database systems (where data are stored in relational table structures; see Section 1.3.1), data modeling tools, and indexing and accessing methods. In addition, users gained convenient and flexible data access through query languages, user interfaces, optimized query processing, and transaction management. Efficient methods for

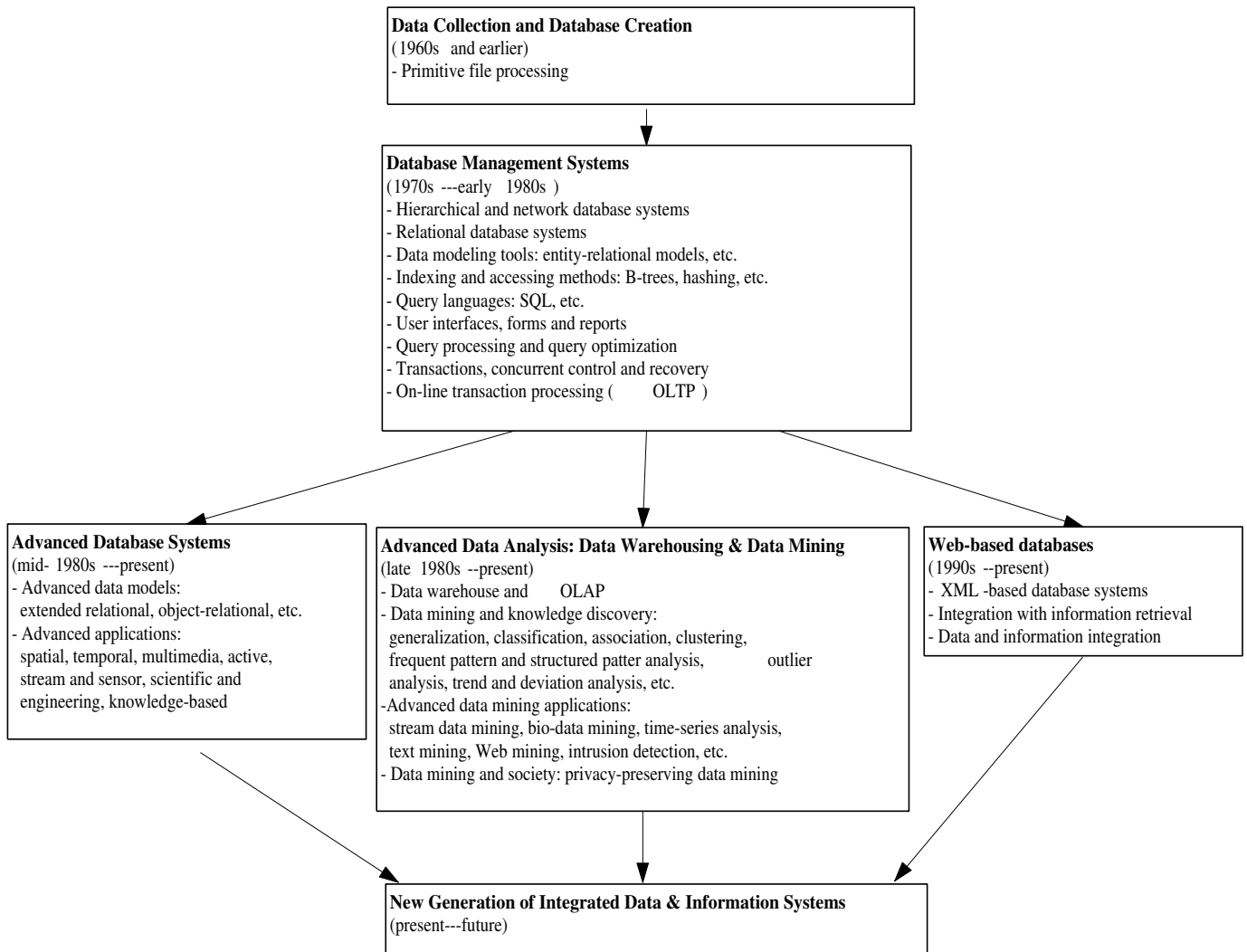


Figure 1.1: The evolution of database system technology.

**on-line transaction processing (OLTP)**, where a query is viewed as a read-only transaction, have contributed substantially to the evolution and wide acceptance of relational technology as a major tool for efficient storage, retrieval, and management of large amounts of data.

Database technology since the mid-1980s has been characterized by the popular adoption of relational technology and an upsurge of research and development activities on new and powerful database systems. These promote the development of advanced data models such as extended-relational, object-oriented, object-relational, and deductive models. Application-oriented database systems, including spatial, temporal, multimedia, active, and scientific and engineering databases, knowledge bases, and office information bases, have flourished. Issues related to the distribution, diversification, and sharing of data have been studied extensively. Heterogeneous database systems and Internet-based global information systems such as the World Wide Web (WWW) have also emerged and play a vital role in the information industry.

The steady and amazing progress of computer hardware technology in the past three decades has led to large supplies of powerful and affordable computers, data collection equipment, and storage media. This technology provides a great boost to the database and information industry, and makes a huge number of databases and

information repositories available for transaction management, information retrieval, and data analysis.

Data can now be stored in many different kinds of databases and information repositories. One data repository architecture that has emerged is the **data warehouse** (Section 1.3.2), a repository of multiple heterogeneous data sources, organized under a unified schema at a single site in order to facilitate management decision making. Data warehouse technology includes data cleaning, data integration, and **On-Line Analytical Processing (OLAP)**, that is, analysis techniques with functionalities such as summarization, consolidation, and aggregation, as well as the ability to view information from different angles. Although OLAP tools support multidimensional analysis and decision making, additional data analysis tools are required for in-depth analysis, such as data classification, clustering, and the characterization of data changes over time. In addition, huge volumes of data can be accumulated beyond databases and data warehouses. Typical such examples include the World-Wide-Web and *data streams*, where data flow in and out like streams as in applications like video surveillance, telecommunication, and sensor networks. The effective and efficient analysis of data in such different forms becomes a challenging task.



Figure 1.2: We are data rich, but information poor.

The abundance of data, coupled with the need for powerful data analysis tools, has been described as a *data rich but information poor* situation. The fast-growing, tremendous amount of data, collected and stored in large and numerous data repositories, has far exceeded our human ability for comprehension without powerful tools (Figure 1.2). As a result, data collected in large data repositories become “data tombs”—data archives that are seldom visited. Consequently, important decisions are often made based not on the information-rich data stored in data repositories but rather on a decision maker’s intuition, simply because the decision maker does not have the tools to extract the valuable knowledge embedded in the vast amounts of data. In addition, consider expert system technologies, which typically rely on users or domain experts to *manually* input knowledge into knowledge bases. Unfortunately, this procedure is prone to biases and errors, and is extremely time-consuming and costly. Data mining tools perform data analysis and may uncover important data patterns, contributing greatly to business strategies, knowledge bases, and scientific and medical research. The widening gap between data and information calls for a systematic development of *data mining tools* that will turn data tombs into “golden nuggets” of knowledge.

## 1.2 So, What Is Data Mining?

Simply stated, **data mining** refers to *extracting or “mining” knowledge from large amounts of data*. The term is actually a misnomer. Remember that the mining of gold from rocks or sand is referred to as *gold mining* rather than rock or sand mining. Thus, data mining should have been more appropriately named “knowledge mining

from data,” which is unfortunately somewhat long. “Knowledge mining,” a shorter term, may not reflect the emphasis on mining from large amounts of data. Nevertheless, mining is a vivid term characterizing the process that finds a small set of precious nuggets from a great deal of raw material (Figure 1.3). Thus, such a misnomer that carries both “data” and “mining” became a popular choice. There are many other terms carrying a similar or slightly different meaning to data mining, such as **knowledge mining from data**, **knowledge extraction**, **data/pattern analysis**, **data archaeology**, and **data dredging**.

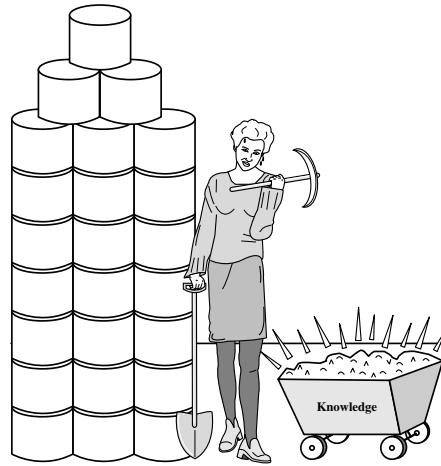


Figure 1.3: Data mining—searching for knowledge (interesting patterns) in your data.

Many people treat data mining as a synonym for another popularly used term, **Knowledge Discovery from Data**, or **KDD**. Alternatively, others view data mining as simply an essential step in the process of knowledge discovery. Knowledge discovery as a process is depicted in Figure 1.4 and consists of an iterative sequence of the following steps:

1. **Data cleaning** (to remove noise and inconsistent data)
2. **Data integration** (where multiple data sources may be combined)<sup>1</sup>
3. **Data selection** (where data relevant to the analysis task are retrieved from the database)
4. **Data transformation** (where data are transformed or consolidated into forms appropriate for mining by performing summary or aggregation operations, for instance)<sup>2</sup>
5. **Data mining** (an essential process where intelligent methods are applied in order to extract data patterns)
6. **Pattern evaluation** (to identify the truly interesting patterns representing knowledge based on some **interestingness measures**; Section 1.5)
7. **Knowledge presentation** (where visualization and knowledge representation techniques are used to present the mined knowledge to the user)

Steps 1 to 4 are different forms of data preprocessing, where the data are prepared for mining. The data mining step may interact with the user or a knowledge base. The interesting patterns are presented to the user, and may

<sup>1</sup>A popular trend in the information industry is to perform data cleaning and data integration as a preprocessing step where the resulting data are stored in a data warehouse.

<sup>2</sup>Sometimes data transformation and consolidation are performed before the data selection process, particularly in the case of data warehousing. *Data reduction* may also be performed to obtain a smaller representation of the original data without sacrificing its integrity.



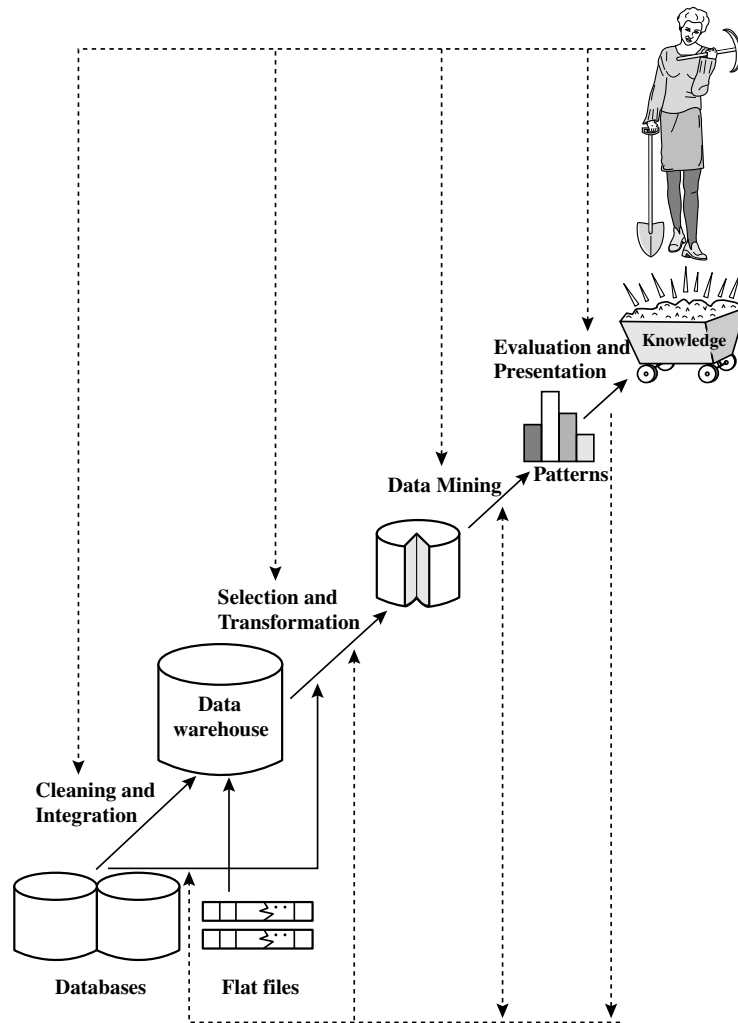


Figure 1.4: Data mining as a step in the process of knowledge discovery.

be stored as new knowledge in the knowledge base. Note that according to this view, data mining is only one step in the entire process, albeit an essential one since it uncovers hidden patterns for evaluation.

We agree that data mining is a step in the knowledge discovery process. However, in industry, in media, and in the database research milieu, the term data mining is becoming more popular than the longer term of knowledge discovery from data. Therefore, in this book, we choose to use the term data mining. We adopt a broad view of data mining functionality: data mining is the process of discovering interesting knowledge from large amounts of data stored either in databases, data warehouses, or other information repositories.

Based on this view, the architecture of a typical data mining system may have the following major components (Figure 1.5):

- **Database, data warehouse, World Wide Web, or other information repository:** This is one or a set of databases, data warehouses, spreadsheets, or other kinds of information repositories. Data cleaning and data integration techniques may be performed on the data.
- **Database or data warehouse server:** The database or data warehouse server is responsible for fetching the relevant data, based on the user's data mining request.

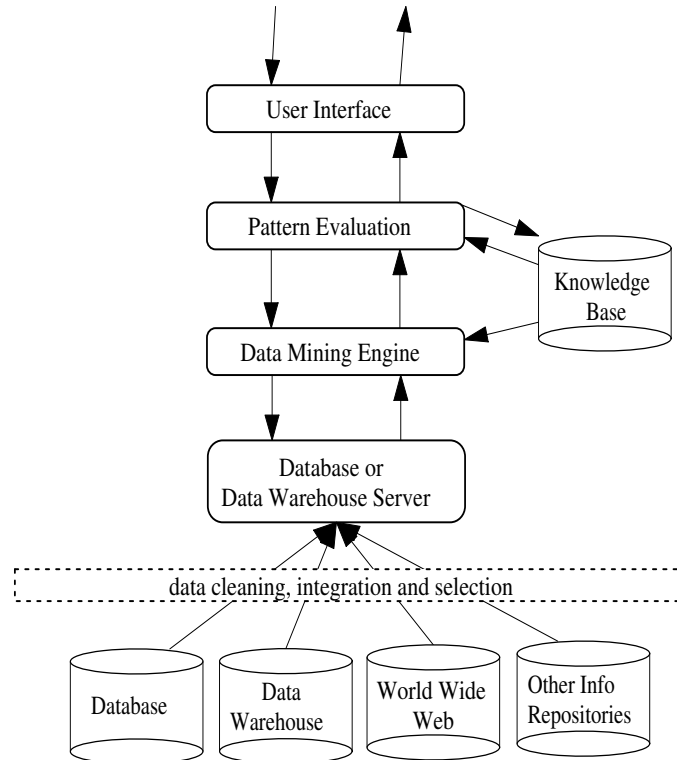


Figure 1.5: Architecture of a typical data mining system.

- **Knowledge base:** This is the domain knowledge that is used to guide the search, or evaluate the interestingness of resulting patterns. Such knowledge can include **concept hierarchies**, used to organize attributes or attribute values into different levels of abstraction. Knowledge such as user beliefs, which can be used to assess a pattern's interestingness based on its unexpectedness, may also be included. Other examples of domain knowledge are additional interestingness constraints or thresholds, and metadata (e.g., describing data from multiple heterogeneous sources).
- **Data mining engine:** This is essential to the data mining system and ideally consists of a set of functional modules for tasks such as characterization, association and correlation analysis, classification, prediction, cluster analysis, outlier analysis, and evolution analysis.
- **Pattern evaluation module:** This component typically employs interestingness measures (Section 1.5) and interacts with the data mining modules so as to *focus* the search towards interesting patterns. It may use interestingness thresholds to filter out discovered patterns. Alternatively, the pattern evaluation module may be integrated with the mining module, depending on the implementation of the data mining method used. For efficient data mining, it is highly recommended to push the evaluation of pattern interestingness as deep as possible into the mining process so as to confine the search to only the interesting patterns.
- **User interface:** This module communicates between users and the data mining system, allowing the user to interact with the system by specifying a data mining query or task, providing information to help focus the search, and performing exploratory data mining based on the intermediate data mining results. In addition, this component allows the user to browse database and data warehouse schemas or data structures, evaluate mined patterns, and visualize the patterns in different forms.

From a data warehouse perspective, data mining can be viewed as an advanced stage of on-line analytical

processing (OLAP). However, data mining goes far beyond the narrow scope of summarization-style analytical processing of data warehouse systems by incorporating more advanced techniques for data analysis.

While there may be many “data mining systems” on the market, not all of them can perform true data mining. A data analysis system that does not handle large amounts of data should be more appropriately categorized as a machine learning system, a statistical data analysis tool, or an experimental system prototype. A system that can only perform data or information retrieval, including finding aggregate values, or that performs deductive query answering in large databases should be more appropriately categorized as a database system, an information retrieval system, or a deductive database system.

Data mining involves an integration of techniques from multiple disciplines such as database and data warehouse technology, statistics, machine learning, high-performance computing, pattern recognition, neural networks, data visualization, information retrieval, image and signal processing, and spatial or temporal data analysis. We adopt a database perspective in our presentation of data mining in this book. That is, emphasis is placed on *efficient* and *scalable* data mining techniques. For an algorithm to be **scalable**, its running time should grow approximately linearly in proportion to the size of the data, given the available system resources such as main memory and disk space. By performing data mining, interesting knowledge, regularities, or high-level information can be extracted from databases and viewed or browsed from different angles. The discovered knowledge can be applied to decision making, process control, information management, and query processing. Therefore, data mining is considered one of the most important frontiers in database and information systems and one of the most promising interdisciplinary developments in the information technology.

## 1.3 Data Mining—On What Kind of Data?

In this section, we examine a number of different data repositories on which mining can be performed. In principle, data mining should be applicable to any kind of data repository, as well as to transient data, such as data streams. Thus the scope of our examination of data repositories will include relational databases, data warehouses, transactional databases, advanced database systems, flat files, data streams, and the World Wide Web. Advanced database systems include object-relational databases, and specific application-oriented databases, such as spatial databases, time-series databases, text databases, and multimedia databases. The challenges and techniques of mining may differ for each of the repository systems.

Although this book assumes that readers have primitive knowledge of information systems, we provide a brief introduction to each of the major data repository systems listed above. In this section, we also introduce the fictitious *AllElectronics* store, which will be used to illustrate concepts throughout the text.

### 1.3.1 Relational Databases

A database system, also called a **database management system (DBMS)**, consists of a collection of interrelated data, known as a **database**, and a set of software programs to manage and access the data. The software programs involve mechanisms for the definition of database structures; for data storage; for concurrent, shared, or distributed data access; and for ensuring the consistency and security of the information stored, despite system crashes or attempts at unauthorized access.

A **relational database** is a collection of **tables**, each of which is assigned a unique name. Each table consists of a set of **attributes** (*columns* or *fields*) and usually stores a large set of **tuples** (*records* or *rows*). Each tuple in a relational table represents an object identified by a unique *key* and described by a set of attribute values. A semantic data model, such as an **entity-relationship (ER)** data model, is often constructed for relational databases. An ER data model represents the database as a set of entities and their relationships.

Consider the following example.

**Example 1.1 A relational database for *AllElectronics*.** The *AllElectronics* company is described by the following relation tables: *customer*, *item*, *employee*, and *branch*. Fragments of the tables described here are shown

<i>customer</i>								
<u>cust_ID</u>	name	address	age	income	credit_info	category	...	
C1	Smith, Sandy	1223 Lake Ave., Chicago, IL	31	\$78000	1	3	...	
...	...	...	...	...	...	...	...	

<i>item</i>								
<u>item_ID</u>	name	brand	category	type	price	place_made	supplier	cost
I3	hi-res-TV	Toshiba	high resolution	TV	\$988.00	Japan	NikoX	\$600.00
I8	Laptop	Dell	laptop	computer	\$1369.00	USA	Dell	\$983.00
...	...	...	...	...	...	...	...	...

<i>employee</i>						
<u>empl_ID</u>	name	category	group	salary	commission	
E55	Jones, Jane	home entertainment	manager	\$118,000	2%	
...	...	...	...	...	...	

<i>branch</i>		
<u>branch_ID</u>	name	address
B1	City Square	396 Michigan Ave, Chicago, IL
...	...	...

<i>purchases</i>						
<u>trans_ID</u>	cust_ID	empl_ID	date	time	method_paid	amount
T100	C1	E55	03/21/2005	15:45	Visa	\$1357.00
...	...	...	...	...	...	...

<i>items_sold</i>		
<u>trans_ID</u>	<u>item_ID</u>	qty
T100	I3	1
T100	I8	2
...	...	...

<i>works_at</i>	
<u>empl_ID</u>	<u>branch_ID</u>
E55	B1
...	...

Figure 1.6: Fragments of relations from a relational database for *AllElectronics*.

in Figure 1.6.

- The relation *customer* consists of a set of attributes, including a unique customer identity number (*cust\_ID*), customer name, address, age, occupation, annual income, credit information, category, and so on.
- Similarly, each of the relations *item*, *employee*, and *branch* consists of a set of attributes, describing their properties.
- Tables can also be used to represent the relationships between or among multiple relation tables. For our example, these include *purchases* (customer purchases items, creating a sales transaction that is handled by an employee), *items\_sold* (lists the items sold in a given transaction), and *works\_at* (employee works at a branch of *AllElectronics*). ■

Relational data can be accessed by **database queries** written in a relational query language, such as SQL, or with the assistance of graphical user interfaces. In the latter, the user may employ a menu, for example, to specify attributes to be included in the query, and the constraints on these attributes. A given query is transformed into a set of relational operations, such as join, selection, and projection, and is then optimized for efficient processing. A query allows retrieval of specified subsets of the data. Suppose that your job is to analyze the *AllElectronics* data. Through the use of relational queries, you can ask things like “Show me a list of all items that were sold in the last quarter.” Relational languages also include aggregate functions such as **sum**, **avg** (average), **count**, **max** (maximum), and **min** (minimum). These allow you to ask things like “Show me the total sales of the last month,

grouped by branch,” or “How many sales transactions occurred in the month of December?” or “Which sales person had the highest amount of sales?”

When data mining is applied to relational databases, one can go further by *searching for trends or data patterns*. For example, data mining systems can analyze customer data to predict the credit risk of new customers based on their income, age, and previous credit information. Data mining systems may also detect deviations, such as items whose sales are far from those expected in comparison with the previous year. Such deviations can then be further investigated (e.g., has there been a change in packaging of such items, or a significant increase in price?).

Relational databases are one of the most popularly available and rich information repositories, and thus they are a major data form in our study of data mining.

### 1.3.2 Data Warehouses

Suppose that *AllElectronics* is a successful international company, with branches around the world. Each branch has its own set of databases. The president of *AllElectronics* has asked you to provide an analysis of the company’s sales per item type per branch for the third quarter. This is a difficult task, particularly since the relevant data are spread out over several databases, physically located at numerous sites.

If *AllElectronics* had a data warehouse, this task would be easy. A **data warehouse** is a repository of information collected from multiple sources, stored under a unified schema, and which usually resides at a single site. Data warehouses are constructed via a process of data cleaning, data integration, data transformation, data loading, and periodic data refreshing. This process is discussed in Chapters 2 and 3. Figure 1.7 shows the typical framework for construction and use of a data warehouse for *AllElectronics*.

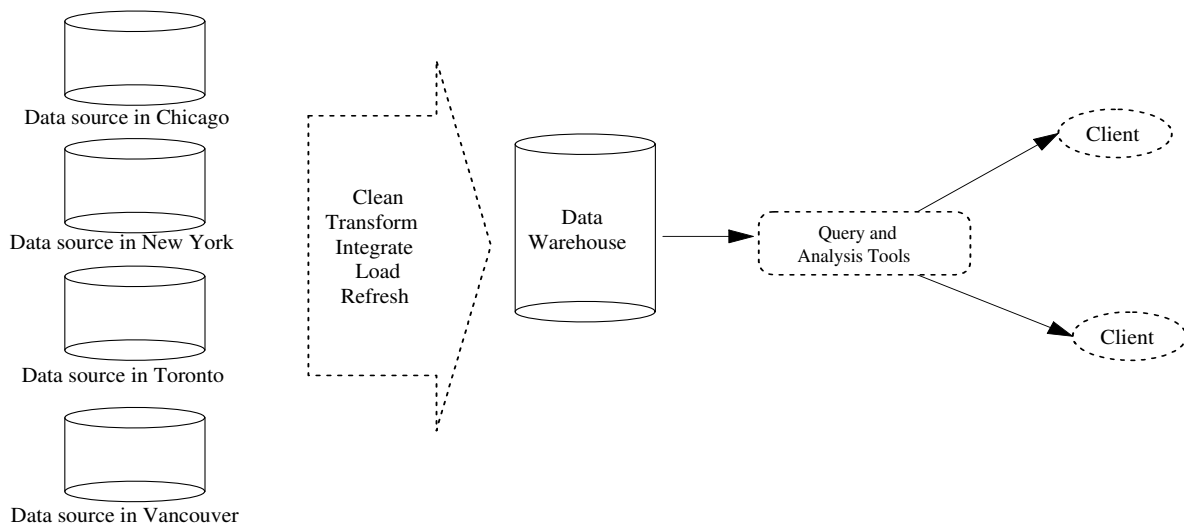


Figure 1.7: Typical framework of a data warehouse for *AllElectronics*.

In order to facilitate decision making, the data in a data warehouse are *organized around major subjects*, such as customer, item, supplier, and activity. The data are stored to provide information from a *historical perspective* (such as from the past 5–10 years) and are typically *summarized*. For example, rather than storing the details of each sales transaction, the data warehouse may store a summary of the transactions per item type for each store or, summarized to a higher level, for each sales region.

A data warehouse is usually modeled by a multidimensional database structure, where each **dimension** corresponds to an attribute or a set of attributes in the schema, and each **cell** stores the value of some aggregate measure, such as *count* or *sales\_amount*. The actual physical structure of a data warehouse may be a relational

data store or a **multidimensional data cube**. A data cube provides a multidimensional view of data and allows the precomputation and fast accessing of summarized data.

**Example 1.2 A data cube for *AllElectronics*.** A data cube for summarized sales data of *AllElectronics* is presented in Figure 1.8(a). The cube has three dimensions: *address* (with city values *Chicago*, *New York*, *Toronto*, *Vancouver*), *time* (with quarter values *Q1*, *Q2*, *Q3*, *Q4*), and *item* (with item type values *home entertainment*, *computer*, *phone*, *security*). The aggregate value stored in each cell of the cube is *sales\_amount* (in thousands). For example, the total sales for the first quarter, *Q1*, for items relating to security systems in Vancouver is \$400,000, as stored in cell  $\langle \text{Vancouver}, Q1, \text{security} \rangle$ . Additional cubes may be used to store aggregate sums over each dimension, corresponding to the aggregate values obtained using different SQL group-bys (e.g., the total sales amount per city and quarter, or per city and item, or per quarter and item, or per each individual dimension). ■

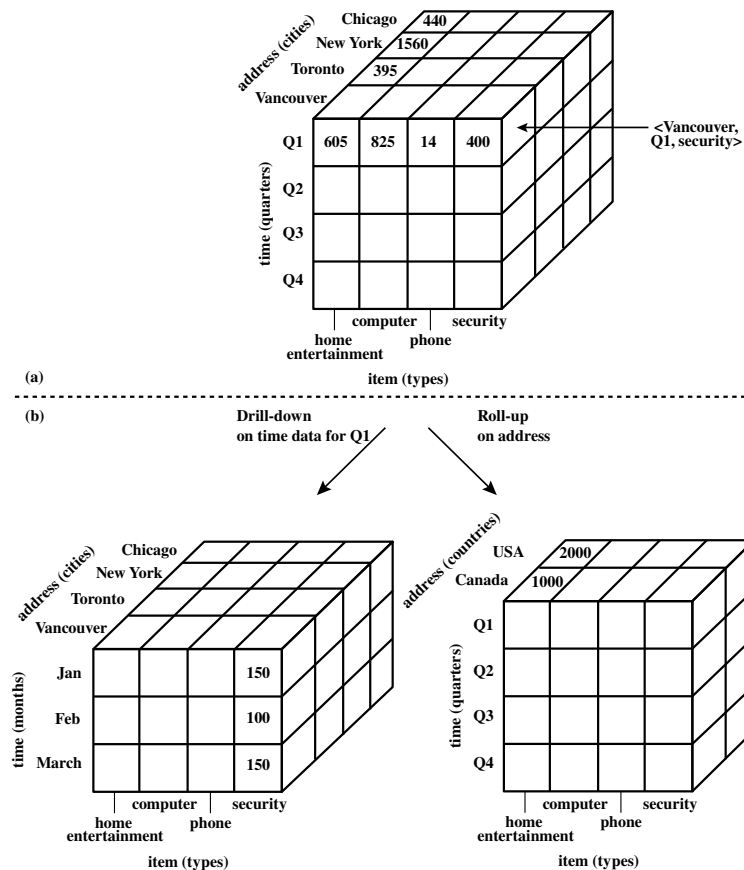


Figure 1.8: A multidimensional data cube, commonly used for data warehousing, (a) showing summarized data for *AllElectronics* and (b) showing summarized data resulting from drill-down and roll-up operations on the cube in (a). For improved readability, only some of the cube cell values are shown. [TO EDITOR For convention, please kindly show *address*, *item* and *time* in italics since they are attribute names. Thank you.]

“I have also heard about data marts. What is the difference between a data warehouse and a data mart?” you may ask. A data warehouse collects information about subjects that span an *entire organization*, and thus its scope is *enterprise-wide*. A **data mart**, on the other hand, is a department subset of a data warehouse. It focuses on selected subjects, and thus its scope is *department-wide*.

By providing multidimensional data views and the precomputation of summarized data, data warehouse systems are well suited for **On-Line Analytical Processing**, or **OLAP**. OLAP operations make use of background

knowledge regarding the domain of the data being studied in order to allow the presentation of data at *different levels of abstraction*. Such operations accommodate different user viewpoints. Examples of OLAP operations include **drill-down** and **roll-up**, which allow the user to view the data at differing degrees of summarization, as illustrated in Figure 1.8(b). For instance, we can drill down on sales data summarized by *quarter* to see the data summarized by *month*. Similarly, we can roll up on sales data summarized by *city* to view the data summarized by *country*.

Although data warehouse tools help support data analysis, additional tools for data mining are required to allow more in-depth and automated analysis. An overview of data warehouse and OLAP technology is provided in Chapter 3. Advanced issues regarding data warehouse and OLAP implementation and data generalization are discussed in Chapter 4.

### 1.3.3 Transactional Databases

In general, a **transactional database** consists of a file where each record represents a transaction. A transaction typically includes a unique transaction identity number (*trans\_ID*), and a list of the **items** making up the transaction (such as items purchased in a store). The transactional database may have additional tables associated with it, which contain other information regarding the sale, such as the date of the transaction, the customer ID number, the ID number of the sales person and of the branch at which the sale occurred, and so on.

<i>trans_ID</i>	<i>list of item_IDs</i>
T100	I1, I3, I8, I16
T200	I2, I8
...	...

Figure 1.9: Fragment of a transactional database for sales at *AllElectronics*.

**Example 1.3 A transactional database for *AllElectronics*.** Transactions can be stored in a table, with one record per transaction. A fragment of a transactional database for *AllElectronics* is shown in Figure 1.9. From the relational database point of view, the *sales* table in Figure 1.9 is a nested relation because the attribute *list of item\_IDs* contains a set of *items*. Since most relational database systems do not support nested relational structures, the transactional database is usually either stored in a flat file in a format similar to that of the table in Figure 1.9, or unfolded into a standard relation in a format similar to that of the *items\_sold* table in Figure 1.6. ■

As an analyst of the *AllElectronics* database, you may like to ask, “Show me all the items purchased by Sandy Smith” or “How many transactions include item number I3?” Answering such queries may require a scan of the entire transactional database.

Suppose you would like to dig deeper into the data by asking, “Which items sold well together?” This kind of *market basket data analysis* would enable you to bundle groups of items together as a strategy for maximizing sales. For example, given the knowledge that printers are commonly purchased together with computers, you could offer an expensive model of printers at a discount to customers buying selected computers, in the hopes of selling more of the expensive printers. A regular data retrieval system is not able to answer queries like the one above. However, data mining systems for transactional data can do so by identifying *frequent itemsets*, that is, sets of items that are frequently sold together. The mining of such frequent patterns for transactional data is discussed in Chapter 5.

### 1.3.4 Advanced Database Systems and Advanced Database Applications

Relational database systems have been widely used in business applications. With the advances of database technology, various kinds of advanced database systems have emerged and are undergoing development to address

the requirements of new database applications.

The new database applications include handling spatial data (such as maps), engineering design data (such as the design of buildings, system components, or integrated circuits), hypertext and multimedia data (including text, image, video, and audio data), time-related data (such as historical records or stock exchange data), stream data (such as video surveillance and sensor data, where data flow in and out like streams), and the World Wide Web (a huge, widely distributed information repository made available by the Internet). These applications require efficient data structures and scalable methods for handling complex object structures; variable-length records; semistructured or unstructured data; text, spatiotemporal, and multimedia data; and database schemas with complex structures and dynamic changes.

In response to these needs, advanced database systems and specific application-oriented database systems have been developed. These include object-relational database systems, temporal and time-series database systems, spatial and spatiotemporal database systems, text and multimedia database systems, heterogeneous and legacy database systems, data stream management systems, and Web-based global information systems.

While such databases or information repositories require sophisticated facilities to efficiently store, retrieve, and update large amounts of complex data, they also provide fertile grounds and raise many challenging research and implementation issues for data mining. In this section, we describe each of the advanced database systems listed above.

### Object-Relational Databases

**Object-relational databases** are constructed based on an object-relational data model. This model extends the relational model by providing a rich data type for handling complex objects and object orientation.

Conceptually, the object-relational data model inherits the essential concepts of **object-oriented databases**, where in general terms, each entity is considered as an **object**. Following the *AllElectronics* example, objects can be individual employees, customers, or items. Data and code relating to an object are *encapsulated* into a single unit. Each object has associated with it the following:

- A set of **variables** that describe the objects. These correspond to attributes in the entity-relationship and relational models.
- A set of **messages** that the object can use to communicate with other objects, or with the rest of the database system.
- A set of **methods**, where each method holds the code to implement a message. Upon receiving a message, the method returns a value in response. For instance, the method for the message *get-photo(employee)* will retrieve and return a photo of the given employee object.

Objects that share a common set of properties can be grouped into an **object class**. Each object is an **instance** of its class. Object classes can be organized into class/subclass hierarchies so that each class represents properties that are common to objects in that class. For instance, an *employee* class can contain variables like *name*, *address*, and *birthdate*. Suppose that the class, *sales\_person*, is a subclass of the class, *employee*. A *sales\_person* object would **inherit** all of the variables pertaining to its superclass of *employee*. In addition, it has all of the variables that pertain specifically to being a sales person (e.g., *commission*). Such a class inheritance feature benefits information sharing.

The object-relational model extends the basic relational data model by adding the power to handle complex data types, class hierarchies, and object inheritance as described above. Since most sophisticated database applications need to handle complex objects and structures, object-relational databases are becoming increasingly popular in industry and applications.

For data mining in object-relational systems, techniques need to be developed for handling complex object structures, complex data types, class and subclass hierarchies, property inheritance, and methods and procedures.



### Temporal Databases and Time-Series Databases

Temporal databases and time-series databases both store time-related data. A **temporal database** usually stores relational data that include time-related attributes. These attributes may involve several timestamps, each having different semantics. A **time-series database** stores sequences of values that change with time, such as data collected regarding the stock exchange.

Data mining techniques can be used to find the characteristics of object evolution, or the trend of changes for objects in the database. Such information can be useful in decision making and strategy planning. For instance, the mining of banking data may aid in the scheduling of bank tellers according to the volume of customer traffic. Stock exchange data can be mined to uncover trends that could help you plan investment strategies (e.g., when is the best time to purchase *AllElectronics* stock?). Such analyses typically require defining multiple granularity of time. For example, time may be decomposed according to fiscal years, academic years, or calendar years. Years may be further decomposed into quarters or months.

### Spatial Databases and Spatiotemporal Databases

**Spatial databases** contain spatial-related information. Examples include geographic (map) databases, VLSI or computed-aided design databases, and medical and satellite image databases. Spatial data may be represented in **raster format**, consisting of  $n$ -dimensional bit maps or pixel maps. For example, a 2-D satellite image may be represented as raster data, where each pixel registers the rainfall in a given area. Maps can be represented in **vector format**, where roads, bridges, buildings, and lakes are represented as unions or overlays of basic geometric constructs, such as points, lines, polygons, and the partitions and networks formed by these components.

Geographic databases have numerous applications, ranging from forestry and ecology planning, to providing public service information regarding the location of telephone and electric cables, pipes, and sewage systems. In addition, geographic databases are commonly used in vehicle navigation and dispatching systems. An example of such a system for taxis would store a city map with information regarding one-way streets, suggested routes for moving from region A to region B during rush hour, the location of restaurants and hospitals, as well as the current location of each driver.

“What kind of data mining can be performed on spatial databases?” you may ask. Data mining may uncover patterns describing the characteristics of houses located near a specified kind of location, such as a park, for instance. Other patterns may describe the climate of mountainous areas located at various altitudes, or describe the change in trend of metropolitan poverty rates based on city distances from major highways. The relationships among a set of spatial objects can be examined in order to discover which subsets of objects are spatially auto-correlated or associated. Clusters and outliers can be identified by spatial cluster analysis. Moreover, spatial classification can be performed to construct models for prediction based on the relevant set of features of the spatial objects. Furthermore, “spatial data cubes” may be constructed to organize data into multidimensional structures and hierarchies, on which OLAP operations (such as drill-down and roll-up) can be performed.

A spatial database that stores spatial objects that change with time is called a **spatiotemporal database**, from which interesting information can be mined. For example, we may be able to group the trends of moving objects and identify some strangely moving vehicles, or distinguish a bioterrorist attack from a normal outbreak of the flu based on the geographic spread of a disease with time.

### Text Databases and Multimedia Databases

**Text databases** are databases that contain word descriptions for objects. These word descriptions are usually not simple keywords but rather long sentences or paragraphs, such as product specifications, error or bug reports, warning messages, summary reports, notes, or other documents. Text databases may be highly unstructured (such as some Web pages on the World Wide Web). Some text databases may be somewhat structured, that is, *semistructured* (such as e-mail messages and many HTML/XML Web pages), while others are relatively well structured (such as library catalogue databases). Text databases with highly regular structures typically can be

implemented using relational database systems.

“*What can data mining on text databases uncover?*” By mining text data one may uncover general and concise descriptions of the text documents, keyword or content associations, as well as the clustering behavior of text objects. To do this, standard data mining methods need to be integrated with information retrieval techniques and the construction or use of hierarchies specifically for text data (such as dictionaries and thesauruses), as well as discipline-oriented term classification systems (such as in biochemistry, medicine, law, or economics).

**Multimedia databases** store image, audio, and video data. They are used in applications such as picture content-based retrieval, voice-mail systems, video-on-demand systems, the World Wide Web, and speech-based user interfaces that recognize spoken commands. Multimedia databases must support large objects, since data objects such as video can require gigabytes of storage. Specialized storage and search techniques are also required. Since video and audio data require real-time retrieval at a steady and predetermined rate in order to avoid picture or sound gaps and system buffer overflows, such data are referred to as **continuous-media** data.

For multimedia data mining, storage and search techniques need to be integrated with standard data mining methods. Promising approaches include the construction of multimedia data cubes, the extraction of multiple features from multimedia data, and similarity-based pattern matching.

### Heterogeneous Databases and Legacy Databases

A **heterogeneous database** consists of a set of interconnected, autonomous component databases. The components communicate in order to exchange information and answer queries. Objects in one component database may differ greatly from objects in other component databases, making it difficult to assimilate their semantics into the overall heterogeneous database.

Many enterprises acquire legacy databases as a result of the long history of information technology development (including the application of different hardware and operating systems). A **legacy database** is a group of *heterogeneous databases* that combines different kinds of data systems, such as relational or object-oriented databases, hierarchical databases, network databases, spreadsheets, multimedia databases, or file systems. The heterogeneous databases in a legacy database may be connected by intra- or inter-computer networks.

Information exchange across such databases is difficult since it would require precise transformation rules from one representation to another, considering diverse semantics. Consider, for example, the problem in exchanging information regarding student academic performance among different schools. Each school may have its own computer system and use its own curriculum and grading system. One university may adopt a quarter system, offer three courses on database systems, and assign grades from A+ to F, while another may adopt a semester system, offer two courses on databases, and assign grades from 1 to 10. It is very difficult to work out precise course-to-grade transformation rules between the two universities, making information exchange difficult. Data mining techniques may provide an interesting solution to the information exchange problem by performing statistical data distribution and correlation analysis, and transforming the given data into higher, more generalized, conceptual levels (such as *fair*, *good*, or *excellent* for student grades), from which information exchange can then more easily be performed.

### Data Streams

Many applications involve the generation and analysis of a new kind of data, called **stream data**, where data flow in-and-out of an observation platform (or window) dynamically. Such data streams have the following unique features: *huge or possibly infinite volume, dynamically changing, flowing in-and-out in a fixed order, allowing only one or a small number of scans, and demanding fast (often real-time) response time*. Typical examples of data streams include various kinds of scientific and engineering data, time-series data, and data produced in other dynamic environments, such as power supply, network traffic, stock exchange, telecommunication data flow, Web click streams, video surveillance data, and weather or environment monitoring.

Since data streams are normally not stored in any kind of data repository, effective and efficient management

and analysis of stream data poses great challenges to researchers. Currently, many researchers are investigating various issues relating to the development of data stream management systems. A typical query model in such a system is the *continuous query model* where predefined queries constantly evaluate incoming streams, collecting aggregate data, reporting the current status of data streams, and responding to their changes.

Mining data streams involves efficient discovery of general patterns and dynamic changes within data streams. For example, we may like to detect intrusions of a computer network based on the anomaly of message flow, which may be discovered by clustering data streams, dynamic construction of stream models, or comparing the current frequent patterns with that at a certain previous time. Most stream data reside at a rather low level of abstraction, whereas analysts are often more interested in higher and multiple levels of abstraction. Thus, multilevel, multidimensional on-line analysis and mining should be performed on stream data as well.

### The World Wide Web

The World Wide Web and its associated distributed information services, such as Yahoo!, Google, America Online, and AltaVista, provide rich, world-wide, on-line information services, where data objects are linked together to facilitate interactive access. Users seeking information of interest traverse from one object via links to another. Such systems provide ample opportunities and challenges for data mining. For example, understanding user access patterns will not only help improve system design (by providing efficient access between highly correlated objects), but also leads to better marketing decisions (e.g., by placing advertisements in frequently visited documents, or by providing better customer/user classification and behavior analysis). Capturing user access patterns in such distributed information environments is called **Web usage mining** (or **Weblog mining**).

Although Web pages may appear fancy and informative to human readers, they can be highly unstructured and lack a predefined schema, type, or pattern. Thus it is difficult for computers to understand the semantic meaning of diverse Web pages and structure them in an organized way for systematic information retrieval and data mining. Web services that provide keyword-based searches without understanding the context behind the Web pages can only offer limited help to users. For example, a Web search based on a single keyword may return hundreds of Web page pointers containing the keyword, but most of the pointers will be very weakly related to what the user wants to find. Data mining can often provide additional help here than Web search services. For example, **authoritative Web page analysis** based on linkages among Web pages can help rank Web pages based on their importance, influence, and topics. **Automated Web page clustering and classification** helps group and arrange Web pages in a multidimensional manner based on their contents. **Web community analysis** helps identify hidden Web social networks and communities and observe their evolution. Web mining is the development of scalable and effective Web data analysis and mining methods. It may help us learn about the distribution of information on the Web in general, characterize and classify Web pages, and uncover Web dynamics and the association and other relationships among different Web pages, users, communities, and Web-based activities.

Data mining in advanced database and information systems is discussed primarily in Chapter 8.

## 1.4 Data Mining Functionalities—What Kinds of Patterns Can Be Mined?

We have observed various types of databases and information repositories on which data mining can be performed. Let us now examine the kinds of data patterns that can be mined.

Data mining functionalities are used to specify the kind of patterns to be found in data mining tasks. In general, data mining tasks can be classified into two categories: **descriptive** and **predictive**. Descriptive mining tasks characterize the general properties of the data in the database. Predictive mining tasks perform inference on the current data in order to make predictions.

In some cases, users may have no idea regarding what kinds of patterns in their data may be interesting, and hence may like to search for several different kinds of patterns in parallel. Thus it is important to have a data

mining system that can mine multiple kinds of patterns to accommodate different user expectations or applications. Furthermore, data mining systems should be able to discover patterns at various granularity (i.e., different levels of abstraction). Data mining systems should also allow users to specify hints to guide or focus the search for interesting patterns. Since some patterns may not hold for all of the data in the database, a measure of certainty or “trustworthiness” is usually associated with each discovered pattern.

Data mining functionalities, and the kinds of patterns they can discover, are described below.

### 1.4.1 Concept/Class Description: Characterization and Discrimination

Data can be associated with classes or concepts. For example, in the *AllElectronics* store, classes of items for sale include *computers* and *printers*, and concepts of customers include *bigSpenders* and *budgetSpenders*. It can be useful to describe individual classes and concepts in summarized, concise, and yet precise terms. Such descriptions of a class or a concept are called **class/concept descriptions**. These descriptions can be derived via (1) *data characterization*, by summarizing the data of the class under study (often called the **target class**) in general terms, or (2) *data discrimination*, by comparison of the target class with one or a set of comparative classes (often called the **contrasting classes**), or (3) both data characterization and discrimination.

**Data characterization** is a summarization of the general characteristics or features of a target class of data. The data corresponding to the user-specified class are typically collected by a database query. For example, to study the characteristics of software products whose sales increased by 10% in the last year, the data related to such products can be collected by executing an SQL query.

There are several methods for effective data summarization and characterization. Simple data summaries based on statistical measures and plots are described in Chapter 2. The data cube-based OLAP roll-up operation (Section 1.3.2) can be used to perform user-controlled data summarization along a specified dimension. This process is further detailed in Chapters 3 and 4, which discuss data warehousing. An *attribute-oriented induction* technique can be used to perform data generalization and characterization without step-by-step user interaction. This technique is described in Chapter 4.

The output of data characterization can be presented in various forms. Examples include **pie charts**, **bar charts**, **curves**, **multidimensional data cubes**, and **multidimensional tables**, including crosstabs. The resulting descriptions can also be presented as **generalized relations** or in rule form (called **characteristic rules**). These different output forms and their transformations are discussed in Chapter 4.

**Example 1.4 Data characterization.** A data mining system should be able to produce a description summarizing the characteristics of customers who spend more than \$1,000 a year at *AllElectronics*. The result could be a general profile of the customers, such as they are 40–50 years old, employed, and have excellent credit ratings. The system should allow users to drill down on any dimension, such as on *occupation* in order to view these customers according to their type of employment. ■

**Data discrimination** is a comparison of the general features of target class data objects with the general features of objects from one or a set of contrasting classes. The target and contrasting classes can be specified by the user, and the corresponding data objects retrieved through database queries. For example, the user may like to compare the general features of software products whose sales increased by 10% in the last year with those whose sales decreased by at least 30% during the same period. The methods used for data discrimination are similar to those used for data characterization.

“How are discrimination descriptions output?” The forms of output presentation are similar to those for characteristic descriptions, although discrimination descriptions should include comparative measures that help distinguish between the target and contrasting classes. Discrimination descriptions expressed in rule form are referred to as **discriminant rules**.

**Example 1.5 Data discrimination.** A data mining system should be able to compare two groups of *AllElectronics* customers, such as those who shop for computer products regularly (more than two times a month) versus

those who rarely shop for such products (i.e., less than three times a year). The resulting description provides a general comparative profile of the customers, such as 80% of the customers who frequently purchase computer products are between 20 and 40 years old and have a university education, whereas 60% of the customers who infrequently buy such products are either seniors or youths, and have no university degree. Drilling down on a dimension, such as *occupation*, or adding new dimensions, such as *income\_level*, may help in finding even more discriminative features between the two classes. ■

Concept description, including characterization and discrimination, is described in Chapter 4.

### 1.4.2 Mining Frequent Patterns, Associations, and Correlations

**Frequent patterns**, as the name suggests, are patterns that occur frequently in data. There are many kinds of frequent patterns, including itemsets, subsequences, and substructures. A *frequent itemset* typically refers to a set of items that frequently appear together in a transactional data set, such as milk and bread. A frequently occurring subsequence, such as the pattern that customers tend to purchase first a PC, followed by a digital camera, and then a memory card, is a (*frequent*) *sequential pattern*. A substructure can refer to different structural forms, such as graphs, trees, or lattices, which may be combined with itemsets or subsequences. If a substructure occurs frequently, it is called a (*frequent*) *structured pattern*. Mining frequent patterns leads to the discovery of interesting associations and correlations within data.

**Example 1.6 Association analysis.** Suppose, as a marketing manager of *AllElectronics*, you would like to determine which items are frequently purchased together within the same transactions. An example of such a rule, mined from the *AllElectronics* transactional database, is

$$\text{buys}(X, \text{"computer"}) \Rightarrow \text{buys}(X, \text{"software"}) \quad [\text{support} = 1\%, \text{confidence} = 50\%]$$

where  $X$  is a variable representing a customer. A **confidence**, or certainty, of 50% means that if a customer buys a computer, there is a 50% chance that she will buy software as well. A 1% **support** means that 1% of all of the transactions under analysis showed that computer and software were purchased together. This association rule involves a single attribute or predicate (i.e., *buys*) that repeats. Association rules that contain a single predicate are referred to as **single-dimensional association rules**. Dropping the predicate notation, the above rule can be written simply as “*computer*  $\Rightarrow$  *software* [1%, 50%]”.

Suppose, instead, that we are given the *AllElectronics* relational database relating to purchases. A data mining system may find association rules like

$$\text{age}(X, \text{"20...29"}) \wedge \text{income}(X, \text{"20K...29K"}) \Rightarrow \text{buys}(X, \text{"CD player"}) \quad [\text{support} = 2\%, \text{confidence} = 60\%]$$

The rule indicates that of the *AllElectronics* customers under study, 2% are 20 to 29 years of age with an income of 20K to 29K and have purchased a CD player at *AllElectronics*. There is a 60% probability that a customer in this age and income group will purchase a CD player. Note that this is an association between more than one attribute, or predicate (i.e., *age*, *income*, and *buys*). Adopting the terminology used in multidimensional databases, where each attribute is referred to as a dimension, the above rule can be referred to as a **multidimensional association rule**. ■

Typically, association rules are discarded as uninteresting if they do not satisfy both a **minimum support threshold** and a **minimum confidence threshold**. Additional analysis can be performed to uncover interesting statistical **correlations** between associated attribute-value pairs.

*Frequent itemset mining* is the simplest form of frequent pattern mining. The mining of frequent patterns, associations, and correlations is discussed in Chapter 5, where particular emphasis is placed on efficient algorithms for frequent itemset mining. The mining of subsequences and substructures is discussed in the second volume of the book.

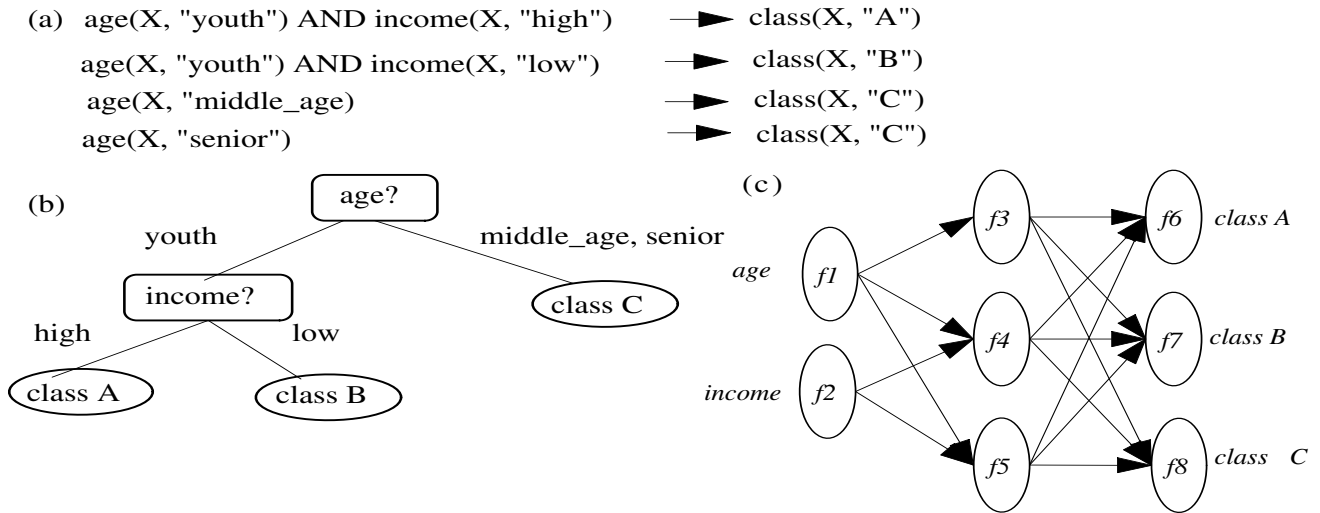


Figure 1.10: A classification model can be represented in various forms, such as (a) IF-THEN rules, (b) a decision tree, or a (c) neural network.

### 1.4.3 Classification and Prediction

**Classification** is the process of finding a **model** (or function) that describes and distinguishes data classes or concepts, for the purpose of being able to use the model to predict the class of objects whose class label is unknown. The derived model is based on the analysis of a set of **training data** (i.e., data objects whose class label is known).

*“How is the derived model presented?”* The derived model may be represented in various forms, such as *classification (IF-THEN) rules*, *decision trees*, *mathematical formulae*, or *neural networks* (Figure 1.10). A **decision tree** is a flow-chart-like tree structure, where each node denotes a test on an attribute value, each branch represents an outcome of the test, and tree leaves represent classes or class distributions. Decision trees can easily be converted to classification rules. A **neural network**, when used for classification, is typically a collection of neuron-like processing units with weighted connections between the units. There are many other methods for constructing classification models, such as naïve Bayesian classification, support vector machines, and *k*-nearest neighbor classification.

Whereas classification predicts categorical (discrete, unordered) labels, **prediction** models continuous-valued functions. That is, it is used to predict missing or unavailable *numerical data values* rather than class labels. Although the term *prediction* may refer to both numeric prediction and class label prediction, in this book we use it to refer primarily to numeric prediction. **Regression analysis** is a statistical methodology that is most often used for numeric prediction, although other methods exist as well. Prediction also encompasses the identification of distribution *trends* based on the available data.

Classification and prediction may need to be preceded by **relevance analysis**, which attempts to identify attributes that do not contribute to the classification or prediction process. These attributes can then be excluded.

**Example 1.7 Classification and prediction.** Suppose, as sales manager of *AllElectronics*, you would like to classify a large set of items in the store, based on three kinds of responses to a sales campaign: *good response*, *mild response*, and *no response*. You would like to derive a model for each of these three classes based on the descriptive features of the items, such as *price*, *brand*, *place\_made*, *type*, and *category*. The resulting classification should maximally distinguish each class from the others, presenting an organized picture of the data set. Suppose that the resulting classification is expressed in the form of a decision tree. The decision tree, for instance, may identify *price* as being the single factor that best distinguishes the three classes. The tree may reveal that, after *price*, other features that help further distinguish objects of each class from another include *brand* and *place\_made*.

Such a decision tree may help you understand the impact of the given sales campaign and design a more effective campaign for the future.

Suppose instead, that rather than predicting categorical response labels for each store item, you would like to predict the amount of revenue that each item will generate during an upcoming sale at *AllElectronics*, based on previous sales data. This is an example of (numeric) prediction since the model constructed will predict a continuous-valued function, or ordered value. ■

Chapter 6 discusses classification and prediction in further detail.

#### 1.4.4 Cluster Analysis

“What is cluster analysis?” Unlike classification and prediction, which analyze class-labeled data objects, **cluster-ing** analyzes data objects without consulting a known class label. In general, the class labels are not present in the training data simply because they are not known to begin with. Clustering can be used to generate such labels. The objects are clustered or grouped based on the principle of *maximizing the intraclass similarity and minimizing the interclass similarity*. That is, clusters of objects are formed so that objects within a cluster have high similarity in comparison to one another, but are very dissimilar to objects in other clusters. Each cluster that is formed can be viewed as a class of objects, from which rules can be derived. Clustering can also facilitate **taxonomy formation**, that is, the organization of observations into a hierarchy of classes that group similar events together.

**Example 1.8 Cluster analysis.** Cluster analysis can be performed on *AllElectronics* customer data in order to identify homogeneous subpopulations of customers. These clusters may represent individual target groups for marketing. Figure 1.11 shows a 2-D plot of customers with respect to customer locations in a city. Three clusters of data points are evident. ■

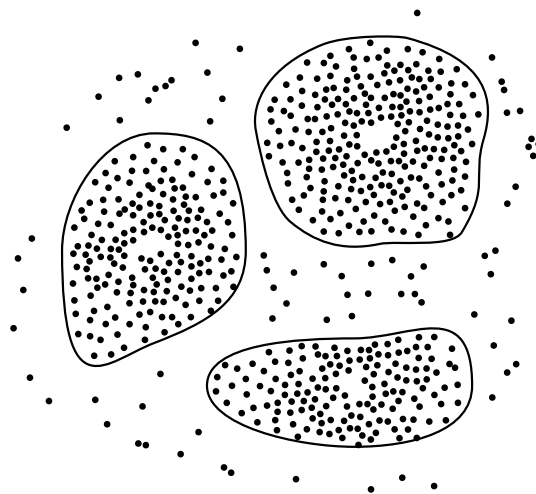


Figure 1.11: A 2-D plot of customer data with respect to customer locations in a city, showing three data clusters. Each cluster “center” is marked with a “+”.

Cluster analysis forms the topic of Chapter 7.

#### 1.4.5 Outlier Analysis

A database may contain data objects that do not comply with the general behavior or model of the data. These data objects are **outliers**. Most data mining methods discard outliers as noise or exceptions. However, in some

applications such as fraud detection, the rare events can be more interesting than the more regularly occurring ones. The analysis of outlier data is referred to as **outlier mining**.

Outliers may be detected using statistical tests that assume a distribution or probability model for the data, or using distance measures where objects that are a substantial distance from any other cluster are considered outliers. Rather than using statistical or distance measures, deviation-based methods identify outliers by examining differences in the main characteristics of objects in a group.

**Example 1.9 Outlier analysis.** Outlier analysis may uncover fraudulent usage of credit cards by detecting purchases of extremely large amounts for a given account number in comparison to regular charges incurred by the same account. Outlier values may also be detected with respect to the location and type of purchase, or the purchase frequency. ■

Outlier analysis is also discussed in Chapter 7.

### 1.4.6 Evolution Analysis

Data **evolution analysis** describes and models regularities or trends for objects whose behavior changes over time. Although this may include characterization, discrimination, association and correlation analysis, classification, prediction, or clustering of *time-related* data, distinct features of such an analysis include time-series data analysis, sequence or periodicity pattern matching, and similarity-based data analysis.

**Example 1.10 Evolution analysis.** Suppose that you have the major stock market (time-series) data of the last several years available from the New York Stock Exchange and you would like to invest in shares of high-tech industrial companies. A data mining study of stock exchange data may identify stock evolution regularities for overall stocks and for the stocks of particular companies. Such regularities may help predict future trends in stock market prices, contributing to your decision making regarding stock investments. ■

Data evolution analysis is discussed in Chapter 8.

## 1.5 Are All of the Patterns Interesting?

A data mining system has the potential to generate thousands or even millions of patterns, or rules.

“So,” you may ask, “*are all of the patterns interesting?*” Typically not—only a small fraction of the patterns potentially generated would actually be of interest to any given user.

This raises some serious questions for data mining. You may wonder, “*What makes a pattern interesting? Can a data mining system generate all of the interesting patterns? Can a data mining system generate only interesting patterns?*”

To answer the first question, a pattern is **interesting** if (1) it is *easily understood* by humans, (2) *valid* on new or test data with some degree of *certainty*, (3) potentially *useful*, and (4) *novel*. A pattern is also interesting if it validates a hypothesis that the user *sought to confirm*. An interesting pattern represents **knowledge**.

Several **objective measures of pattern interestingness** exist. These are based on the structure of discovered patterns and the statistics underlying them. An objective measure for association rules of the form  $X \Rightarrow Y$  is rule **support**, representing the percentage of transactions from a transaction database that the given rule satisfies. This is taken to be the probability  $P(X \cup Y)$ , where  $X \cup Y$  indicates that a transaction contains both  $X$  and  $Y$ , that is, the union of item sets  $X$  and  $Y$ . Another objective measure for association rules is **confidence**, which assesses the degree of certainty of the detected association. This is taken to be the conditional probability  $P(Y|X)$ , that is, the probability that a transaction containing  $X$  also contains  $Y$ . More formally, support and confidence are defined as

$$\text{support}(X \Rightarrow Y) = P(X \cup Y).$$



$$\text{confidence}(X \Rightarrow Y) = P(Y|X).$$

In general, each interestingness measure is associated with a threshold, which may be controlled by the user. For example, rules that do not satisfy a confidence threshold of, say, 50% can be considered uninteresting. Rules below the threshold likely reflect noise, exceptions, or minority cases and are probably of less value.

Although objective measures help identify interesting patterns, they are insufficient unless combined with subjective measures that reflect the needs and interests of a particular user. For example, patterns describing the characteristics of customers who shop frequently at *AllElectronics* should interest the marketing manager, but may be of little interest to analysts studying the same database for patterns on employee performance. Furthermore, many patterns that are interesting by objective standards may represent common knowledge and, therefore, are actually uninteresting. **Subjective interestingness measures** are based on user beliefs in the data. These measures find patterns interesting if they are **unexpected** (contradicting a user's belief) or offer strategic information on which the user can act. In the latter case, such patterns are referred to as **actionable**. Patterns that are **expected** can be interesting if they confirm a hypothesis that the user wished to validate, or resemble a user's hunch.

The second question—“*Can a data mining system generate all of the interesting patterns?*”—refers to the **completeness** of a data mining algorithm. It is often unrealistic and inefficient for data mining systems to generate all of the possible patterns. Instead, user-provided constraints and interestingness measures should be used to focus the search. For some mining tasks, such as association, this is often sufficient to ensure the completeness of the algorithm. Association rule mining is an example where the use of constraints and interestingness measures can ensure the completeness of mining. The methods involved are examined in detail in Chapter 5.

Finally, the third question—“*Can a data mining system generate only interesting patterns?*”—is an optimization problem in data mining. It is highly desirable for data mining systems to generate only interesting patterns. This would be much more efficient for users and data mining systems, since neither would have to search through the patterns generated in order to identify the truly interesting ones. Progress has been made in this direction. However, such optimization remains a challenging issue in data mining.

Measures of pattern interestingness are essential for the efficient discovery of patterns of value to the given user. Such measures can be used after the data mining step in order to rank the discovered patterns according to their interestingness, filtering out the uninteresting ones. More importantly, such measures can be used to guide and constrain the discovery process, improving the search efficiency by pruning away subsets of the pattern space that do not satisfy prespecified interestingness constraints. Such constraint-based mining is described in Chapter 5 (with respect to association mining) and Chapter 7 (with respect to clustering).

Methods to assess pattern interestingness, and their use to improve data mining efficiency, are discussed throughout the book, with respect to each kind of pattern that can be mined.

## 1.6 Classification of Data Mining Systems

Data mining is an interdisciplinary field, the confluence of a set of disciplines (as shown in Figure 1.12), including database systems, statistics, machine learning, visualization, and information science. Moreover, depending on the data mining approach used, techniques from other disciplines may be applied, such as neural networks, fuzzy and/or rough set theory, knowledge representation, inductive logic programming, or high performance computing. Depending on the kinds of data to be mined or on the given data mining application, the data mining system may also integrate techniques from spatial data analysis, information retrieval, pattern recognition, image analysis, signal processing, computer graphics, Web technology, economics, business, bioinformatics, or psychology.

Because of the diversity of disciplines contributing to data mining, data mining research is expected to generate a large variety of data mining systems. Therefore, it is necessary to provide a clear classification of data mining systems. Such a classification may help potential users distinguish data mining systems and identify those that best match their needs. Data mining systems can be categorized according to various criteria, as follows.

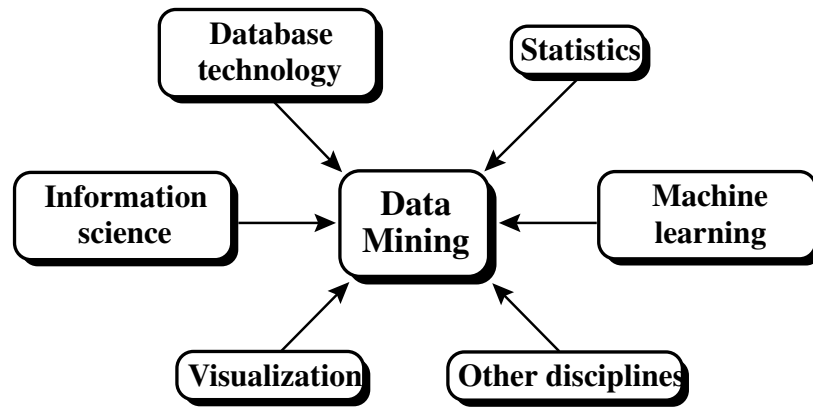


Figure 1.12: Data mining as a confluence of multiple disciplines.

Classification according to the *kinds of databases* mined: A data mining system can be classified according to the kinds of databases mined. Database systems themselves can be classified according to different criteria (such as data models, or the types of data or applications involved), each of which may require its own data mining technique. Data mining systems can therefore be classified accordingly.

For instance, if classifying according to data models, we may have a relational, transactional, object-relational, or data warehouse mining system. If classifying according to the special types of data handled, we may have a spatial, time-series, text, stream data, multimedia data mining system, or a World Wide Web mining system.

Classification according to the *kinds of knowledge* mined: Data mining systems can be categorized according to the kinds of knowledge they mine, that is, based on data mining functionalities, such as characterization, discrimination, association and correlation analysis, classification, prediction, clustering, outlier analysis, and evolution analysis. A comprehensive data mining system usually provides multiple and/or integrated data mining functionalities.

Moreover, data mining systems can be distinguished based on the granularity or levels of abstraction of the knowledge mined, including generalized knowledge (at a high level of abstraction), primitive-level knowledge (at a raw data level), or knowledge at multiple levels (considering several levels of abstraction). An advanced data mining system should facilitate the discovery of knowledge at multiple levels of abstraction.

Data mining systems can also be categorized as those that mine data regularities (commonly occurring patterns) versus those that mine data irregularities (such as exceptions, or outliers). In general, concept description, association and correlation analysis, classification, prediction, and clustering mine data regularities, rejecting outliers as noise. These methods may also help to detect outliers.

Classification according to the *kinds of techniques* utilized: Data mining systems can be categorized according to the underlying data mining techniques employed. These techniques can be described according to the degree of user interaction involved (e.g., autonomous systems, interactive exploratory systems, query-driven systems) or the methods of data analysis employed (e.g., database-oriented or data warehouse-oriented techniques, machine learning, statistics, visualization, pattern recognition, neural networks, and so on). A sophisticated data mining system will often adopt multiple data mining techniques or work out an effective, integrated technique that combines the merits of a few individual approaches.

Classification according to the *applications adapted*: Data mining systems can also be categorized according to the applications they adapt. For example, there could be data mining systems tailored specifically for finance, telecommunications, DNA, stock markets, e-mail, and so on. Different applications often require the integration of application-specific methods. Therefore, a generic, all-purpose data mining system may not fit domain-specific mining tasks.

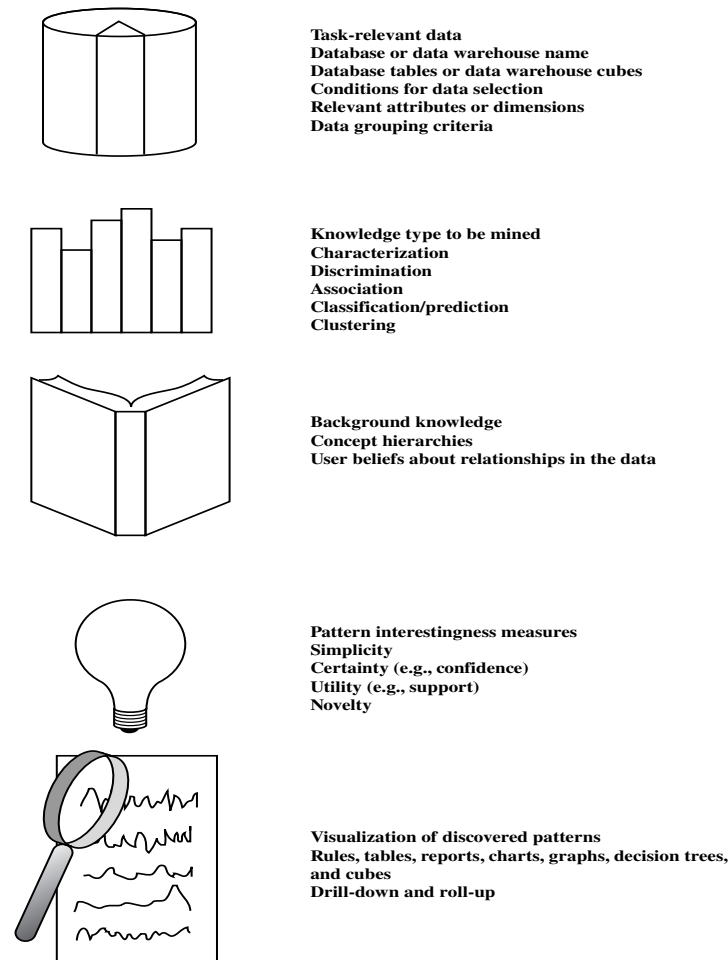


Figure 1.13: Primitives for specifying a data mining task.

[TO EDITOR Please change “association” to “association/correlation”. Thanks.]

In general, Chapters 4 to 7 of this book are organized according to the various kinds of knowledge mined. In Chapter 8, we discuss the mining of complex types of data on a variety of advanced database systems. In Chapter 9, some data mining applications are discussed.

## 1.7 Data Mining Task Primitives

Each user will have a **data mining task** in mind, that is, some form of data analysis that she would like to have performed. A data mining task can be specified in the form of a **data mining query**, which is input to the data mining system. A data mining query is defined in terms of **data mining task primitives**. These primitives allow the user to *interactively* communicate with the data mining system during discovery in order to direct the mining process, or examine the findings from different angles or depths. The data mining primitives specify the following, as illustrated in Figure 1.13.

- The set of *task-relevant data* to be mined: This specifies the portions of the database or the set of data in which the user is interested. This includes the database attributes or data warehouse dimensions of interest (referred to as the *relevant attributes or dimensions*).

- The *kind of knowledge* to be mined: This specifies the *data mining functions* to be performed, such as characterization, discrimination, association or correlation analysis, classification, prediction, clustering, outlier analysis, or evolution analysis.

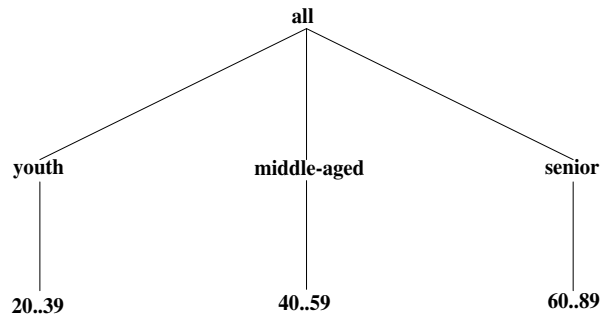


Figure 1.14: A concept hierarchy for the attribute (or dimension) *age*. The root node represents the most general abstraction level, denoted as *all*.

- The *background knowledge* to be used in the discovery process: This knowledge about the domain to be mined is useful for guiding the knowledge discovery process and for evaluating the patterns found. *Concept hierarchies* are a popular form of background knowledge, which allow data to be mined at multiple levels of abstraction. An example of a concept hierarchy for the attribute (or dimension) *age* is shown in Figure 1.14. User beliefs regarding relationships in the data are another form of background knowledge.
- The *interestingness measures and thresholds* for pattern evaluation: They may be used to guide the mining process or, after discovery, to evaluate the discovered patterns. Different kinds of knowledge may have different interestingness measures. For example, interestingness measures for association rules include *support* and *confidence*. Rules whose support and confidence values are below user-specified thresholds are considered uninteresting.
- The expected *representation for visualizing* the discovered patterns: This refers to the form in which discovered patterns are to be displayed, which may include rules, tables, charts, graphs, decision trees, and cubes.

A **data mining query language** can be designed to incorporate these primitives, allowing users to flexibly interact with data mining systems. Having a data mining query language provides a foundation on which user-friendly graphical interfaces can be built. This facilitates a data mining system's communication with other information systems and its integration with the overall information processing environment.

Designing a comprehensive data mining language is challenging because data mining covers a wide spectrum of tasks, from data characterization to evolution analysis. Each task has different requirements. The design of an effective data mining query language requires a deep understanding of the power, limitation, and underlying mechanisms of the various kinds of data mining tasks.

There are several proposals on data mining languages and standards. In this book, we use a data mining query language known as **DMQL** (Data Mining Query Language), which was designed as a teaching tool, based on the above primitives. Examples of its use to specify data mining queries appear throughout this book. The language adopts an SQL-like syntax, so that it can easily be integrated with the relational query language, SQL. Let's have a look at how it can be used to specify a data mining task.

**Example 1.11 Mining classification rules.** Suppose, as a marketing manager of *AllElectronics*, you would like to classify customers based on their buying patterns. You are especially interested in those customers whose salary is no less than \$40,000, and who have bought more than \$1,000 worth of items, each of which is priced at

no less than \$100. In particular, you are interested in the customer's age, income, the types of items purchased, the purchase location, and where the items were made. You would like to view the resulting classification in the form of rules. This data mining query is expressed in DMQL<sup>3</sup> as follows, where each line of the query has been enumerated to aid in our discussion.

- (1) use database AllElectronics\_db
- (2) use hierarchy location\_hierarchy for T.branch, age\_hierarchy for C.age
- (3) mine classification as promising\_customers
- (4) in relevance to C.age, C.income, I.type, I.place\_made, T.branch
- (5) from customer C, item I, transaction T
- (6) where I.item\_ID = T.item\_ID and C.cust\_ID = T.cust\_ID  
and C.income  $\geq$  40,000 and I.price  $\geq$  100
- (7) group by T.cust\_ID
- (8) having sum(I.price)  $\geq$  1,000
- (9) display as rules

The data mining query is parsed to form an SQL query that retrieves the set of task-relevant data specified by lines 1, and 4 to 8. That is, line 1 specifies the *AllElectronics* database, line 4 lists the relevant attributes (i.e., on which mining is to be performed) from the relations specified in line 5 for the conditions given in lines 6 and 7. Line 2 specifies that the concept hierarchies *location\_hierarchy* and *age\_hierarchy* be used as background knowledge to generalize branch locations and customer age values, respectively. Line 3 specifies that the kind of knowledge to be mined for this task is classification. Note that we want to generate a classification model for “promising\_customers” versus “non\_promising\_customers”. In classification, often, an attribute may be specified as the **class label attribute**, whose values *explicitly* represent the classes. However, in this example, the two classes are *implicit*. That is, the set of specified data are retrieved and considered examples of “promising\_customers”, whereas the remaining customers in the customer table are considered as “non-promising\_customers”. Classification is performed based on this training set. Line 9 specifies that the mining results are to be displayed as a set of rules. Several detailed classification methods are introduced in Chapter 6. ■

There is no standard data mining query language today. However, researchers and industry have been making good progress at developing new data mining standards in this area. Such work includes *DMX*, an XML-styled data mining language for Microsoft SQLServer by Microsoft Corporation, and *PMML* (Programming data Model Markup Language) by DMG ([www.dmg.org](http://www.dmg.org)), an independent group led by a group of major vendors in data mining. A brief introduction to each of these is provided in Appendices A and B, respectively, of this book.

## 1.8 Integration of a Data Mining System with a Database or Data Warehouse System

Section 1.2 outlined the major components of the architecture for a typical data mining system (Figure 1.5). A good system architecture will facilitate the data mining system to make best use of the software environment, accomplish data mining tasks in an efficient and timely manner, interoperate and exchange information with other information systems, be adaptable to users' diverse requirements, and evolve with time.

A critical question in the design of a data mining (DM) system is how to integrate or *couple* the DM system with a database (DB) system and/or a data warehouse (DW) system. If a DM system works as a stand-alone system or is embedded in an application program, there are no DB or DW systems with which it has to communicate. This simple scheme is called *no coupling*, where the main focus of the DM design rests on developing effective and efficient algorithms for mining the available data sets. However, when a DM system works in an environment that requires it to communicate with other information system components, such as DB and DW systems, possible integration schemes include *no coupling*, *loose coupling*, *semitight coupling*, and *tight coupling*. We examine each of these schemes, one by one.

<sup>3</sup>Note that in this book, query language keywords are displayed in sans serif font.

- **No coupling:** *No coupling* means that a DM system will not utilize any function of a DB or DW system. It may fetch data from a particular source (such as a file system), process data using some data mining algorithms, and then store the mining results in another file.

Such a system, though simple, suffers from several drawbacks. First, a DB system provides a great deal of flexibility and efficiency at storing, organizing, accessing, and processing data. Without using a DB/DW system, a DM system may spend a substantial amount of time finding, collecting, cleaning, and transforming data. In DB and/or DW systems, data tend to be well organized, indexed, cleaned, integrated, or consolidated, so that finding the task-relevant, high-quality data becomes an easy task. Second, there are many tested, scalable algorithms and data structures implemented in DB and DW systems. It is feasible to realize efficient, scalable implementations using such systems. Moreover, most data have been or will be stored in DB/DW systems. Without any coupling of such systems, a DM system will need to use other tools to extract data, making it difficult to integrate such a system into an information processing environment. Thus, no coupling represents a poor design.

- **Loose coupling:** *Loose coupling* means that a DM system will use some facilities of a DB or DW system, fetching data from a data repository managed by these systems, performing data mining, and then storing the mining results either in a file or in a designated place in a database or data warehouse.

Loose coupling is better than no coupling since it can fetch any portion of data stored in databases or data warehouses by using query processing, indexing, and other system facilities. It incurs some advantages of the flexibility, efficiency, and other features provided by such systems. However, many loosely coupled mining systems are main memory-based. Since mining itself does not explore data structures and query optimization methods provided by DB or DW systems, it is difficult for loose coupling to achieve high scalability and good performance with large data sets.

- **Semitight coupling:** *Semitight coupling* means that besides linking a DM system to a DB/DW system, efficient implementations of a few essential data mining primitives (identified by the analysis of frequently encountered data mining functions) can be provided in the DB/DW system. These primitives can include sorting, indexing, aggregation, histogram analysis, multiway join, and precomputation of some essential statistical measures, such as sum, count, max, min, standard deviation, and so on. Moreover, some frequently used intermediate mining results can be precomputed and stored in the DB/DW system. Since these intermediate mining results are either precomputed or can be computed efficiently, this design will enhance the performance of a DM system.
- **Tight coupling:** *Tight coupling* means that a DM system is smoothly integrated into the DB/DW system. The data mining subsystem is treated as one functional component of an information system. Data mining queries and functions are optimized based on mining query analysis, data structures, indexing schemes, and query processing methods of a DB or DW system. With further technology advances, DM, DB, and DW systems will evolve and integrate together as one information system with multiple functionalities. This will provide a uniform information processing environment.

This approach is highly desirable since it facilitates efficient implementations of data mining functions, high system performance, and an integrated information processing environment.

With this analysis, it is easy to see that a data mining system should be coupled with a DB/DW system. Loose coupling, though not efficient, is better than no coupling since it makes use of both data and system facilities of a DB/DW system. Tight coupling is highly desirable, but its implementation is nontrivial and more research is needed in this area. Semitight coupling is a compromise between loose and tight coupling. It is important to identify commonly used data mining primitives and provide efficient implementations of such primitives in DB or DW systems.

## 1.9 Major Issues in Data Mining

The scope of this book addresses major issues in data mining regarding mining methodology, user interaction, performance, and diverse data types. These issues are introduced below:

**Mining methodology and user interaction issues:** These reflect the kinds of knowledge mined, the ability to mine knowledge at multiple granularity, the use of domain knowledge, ad hoc mining, and knowledge visualization.

- *Mining different kinds of knowledge in databases:* Since different users can be interested in different kinds of knowledge, data mining should cover a wide spectrum of data analysis and knowledge discovery tasks, including data characterization, discrimination, association and correlation analysis, classification, prediction, clustering, outlier analysis, and evolution analysis (which includes trend and similarity analysis). These tasks may use the same database in different ways and require the development of numerous data mining techniques.
- *Interactive mining of knowledge at multiple levels of abstraction:* Since it is difficult to know exactly what can be discovered within a database, the data mining process should be *interactive*. For databases containing a huge amount of data, appropriate sampling techniques can first be applied to facilitate interactive data exploration. Interactive mining allows users to focus the search for patterns, providing and refining data mining requests based on returned results. Specifically, knowledge should be mined by drilling down, rolling up, and pivoting through the data space and knowledge space interactively, similar to what OLAP can do on data cubes. In this way, the user can interact with the data mining system to view data and discovered patterns at multiple granularities and from different angles.
- *Incorporation of background knowledge:* Background knowledge, or information regarding the domain under study, may be used to guide the discovery process and allow discovered patterns to be expressed in concise terms and at different levels of abstraction. Domain knowledge related to databases, such as integrity constraints and deduction rules, can help focus and speed up a data mining process, or judge the interestingness of discovered patterns.
- *Data mining query languages and ad hoc data mining:* Relational query languages (such as SQL) allow users to pose ad hoc queries for data retrieval. In a similar vein, high-level data mining query languages need to be developed to allow users to describe ad hoc data mining tasks by facilitating the specification of the relevant sets of data for analysis, the domain knowledge, the kinds of knowledge to be mined, and the conditions and constraints to be enforced on the discovered patterns. Such a language should be integrated with a database or data warehouse query language, and optimized for efficient and flexible data mining.
- *Presentation and visualization of data mining results:* Discovered knowledge should be expressed in high-level languages, visual representations, or other expressive forms so that the knowledge can be easily understood and directly usable by humans. This is especially crucial if the data mining system is to be interactive. This requires the system to adopt expressive knowledge representation techniques, such as trees, tables, rules, graphs, charts, crosstabs, matrices, or curves.
- *Handling noisy or incomplete data:* The data stored in a database may reflect noise, exceptional cases, or incomplete data objects. When mining data regularities, these objects may confuse the process, causing the knowledge model constructed to overfit the data. As a result, the accuracy of the discovered patterns can be poor. Data cleaning methods and data analysis methods that can handle noise are required, as well as outlier mining methods for the discovery and analysis of exceptional cases.
- *Pattern evaluation—the interestingness problem:* A data mining system can uncover thousands of patterns. Many of the patterns discovered may be uninteresting to the given user, representing common knowledge or lacking novelty. Several challenges remain regarding the development of techniques to assess the interestingness of discovered patterns, particularly with regard to subjective measures that estimate the value of patterns with respect to a given user class, based on user beliefs or expectations. The use of interestingness measures or user-specified constraints to guide the discovery process and reduce the search space is another active area of research.

**Performance issues:** These include efficiency, scalability, and parallelization of data mining algorithms.

- *Efficiency and scalability of data mining algorithms:* To effectively extract information from a huge amount of data in databases, data mining algorithms must be efficient and scalable. In other words,

the running time of a data mining algorithm must be predictable and acceptable in large databases. From a database perspective on knowledge discovery, efficiency and scalability are key issues in the implementation of data mining systems. Many of the issues discussed above under *mining methodology and user interaction* must also consider efficiency and scalability.

- *Parallel, distributed, and incremental mining algorithms:* The huge size of many databases, the wide distribution of data, and the computational complexity of some data mining methods are factors motivating the development of **parallel and distributed data mining algorithms**. Such algorithms divide the data into partitions, which are processed in parallel. The results from the partitions are then merged. Moreover, the high cost of some data mining processes promotes the need for **incremental** data mining algorithms that incorporate database updates without having to mine the entire data again “from scratch.” Such algorithms perform knowledge modification incrementally to amend and strengthen what was previously discovered.

#### Issues relating to the diversity of database types:

- *Handling of relational and complex types of data:* Since relational databases and data warehouses are widely used, the development of efficient and effective data mining systems for such data is important. However, other databases may contain complex data objects, hypertext and multimedia data, spatial data, temporal data, or transaction data. It is unrealistic to expect one system to mine all kinds of data, given the diversity of data types and different goals of data mining. Specific data mining systems should be constructed for mining specific kinds of data. Therefore, one may expect to have different data mining systems for different kinds of data.
- *Mining information from heterogeneous databases and global information systems:* Local- and wide-area computer networks (such as the Internet) connect many sources of data, forming huge, distributed, and heterogeneous databases. The discovery of knowledge from different sources of structured, semistructured, or unstructured data with diverse data semantics poses great challenges to data mining. Data mining may help disclose high-level data regularities in multiple heterogeneous databases that are unlikely to be discovered by simple query systems and may improve information exchange and interoperability in heterogeneous databases. Web mining, which uncovers interesting knowledge about Web contents, Web structures, Web usage, and Web dynamics, becomes a very challenging and highly dynamic field in data mining.

The above issues are considered major requirements and challenges for the further evolution of data mining technology. Some of the challenges have been addressed in recent data mining research and development, *to a certain extent*, and are now considered *requirements*, while others are still at the research stage. The issues, however, continue to stimulate further investigation and improvement. Additional issues relating to applications, privacy, and the social impacts of data mining are discussed in Chapter 9, the final chapter of this book.

## 1.10 Summary

- **Database technology** has evolved from primitive file processing to the development of database management systems with query and transaction processing. Further progress has led to the increasing demand for efficient and effective advanced data analysis tools. This need is a result of the explosive growth in data collected from applications including business and management, government administration, science and engineering, and environmental control.
- **Data mining** is the task of discovering interesting patterns from large amounts of data where the data can be stored in databases, data warehouses, or other information repositories. It is a young interdisciplinary field, drawing from areas such as database systems, data warehousing, statistics, machine learning, data visualization, information retrieval, and high-performance computing. Other contributing areas include neural networks, pattern recognition, spatial data analysis, image databases, signal processing, and many application fields, such as business, economics, and bioinformatics.



- A **knowledge discovery process** includes data cleaning, data integration, data selection, data transformation, data mining, pattern evaluation, and knowledge presentation.
- The **architecture** of a typical data mining system includes a database and/or data warehouse and their appropriate servers, a data mining engine and pattern evaluation module (both of which interact with a knowledge base), and a graphical user interface. **Integration** of the data mining components, as a whole, with a database or data warehouse system can involve either no coupling, loose coupling, semitight coupling, or tight coupling. A well-designed data mining system should offer tight or semitight coupling with a database and/or data warehouse system.
- Data patterns can be mined from many different kinds of **databases**, such as relational databases, data warehouses, and transactional, and object-relational databases. Interesting data patterns can also be extracted from other kinds of **information repositories**, including spatial, time-related, text, multimedia, and legacy databases, data streams, and the World Wide Web.
- A **data warehouse** is a repository for long-term storage of data from multiple sources, organized so as to facilitate management decision making. The data are stored under a unified schema and are typically summarized. Data warehouse systems provide some data analysis capabilities, collectively referred to as **OLAP (On-Line Analytical Processing)**.
- **Data mining functionalities** include the discovery of concept/class descriptions, associations and correlations, classification, prediction, clustering, trend analysis, outlier and deviation analysis, and similarity analysis. Characterization and discrimination are forms of data summarization.
- A pattern represents **knowledge** if it is easily understood by humans; valid on test data with some degree of certainty; and potentially useful, novel, or validates a hunch about which the user was curious. Measures of **pattern interestingness**, either *objective* or *subjective*, can be used to guide the discovery process.
- **Data mining systems** can be **classified** according to the kinds of databases mined, the kinds of knowledge mined, the techniques used, or the applications adapted.
- We have studied five **primitives** for specifying a data mining task in the form of a **data mining query**. These primitives are the specification of task-relevant data (i.e., the data set to be mined), the kind of knowledge to be mined, background knowledge (typically in the form of concept hierarchies), interestingness measures, and knowledge presentation and visualization techniques to be used for displaying the discovered patterns.
- **Data mining query languages** can be designed to support ad hoc and interactive data mining. A data mining query language, such as DMQL, should provide commands for specifying each of the data mining primitives. Such query languages are SQL-based and may eventually form a standard on which graphical user interfaces for data mining can be based.
- Efficient and effective data mining in large databases poses numerous requirements and great challenges to researchers and developers. The issues involved include data mining methodology, user interaction, performance and scalability, and the processing of a large variety of data types. Other issues include the exploration of data mining applications and their social impacts.

## 1.11 Exercises

1. What is *data mining*? In your answer, address the following:
  - (a) Is it another hype?
  - (b) Is it a simple transformation of technology developed from databases, statistics, and machine learning?
  - (c) Explain how the evolution of database technology led to data mining.
  - (d) Describe the steps involved in data mining when viewed as a process of knowledge discovery.

2. Present an example where data mining is crucial to the success of a business. What *data mining functions* does this business need? Can they be performed alternatively by data query processing or simple statistical analysis?
3. Suppose your task as a software engineer at *Big-University* is to design a data mining system to examine their university course database, which contains the following information: the name, address, and status (e.g., undergraduate or graduate) of each student, the courses taken, and their cumulative grade point average (GPA). Describe the *architecture* you would choose. What is the purpose of each component of this architecture?
4. How is a *data warehouse* different from a database? How are they similar?
5. Briefly describe the following *advanced database systems* and applications: object-relational databases, spatial databases, text databases, multimedia databases, the World Wide Web.
6. Define each of the following *data mining functionalities*: characterization, discrimination, association and correlation analysis, classification, prediction, clustering, and evolution analysis. Give examples of each data mining functionality, using a real-life database that you are familiar with.
7. What is the difference between discrimination and classification? Between characterization and clustering? Between classification and prediction? For each of these pairs of tasks, how are they similar?
8. Based on your observation, describe another possible kind of knowledge that needs to be discovered by data mining methods but has not been listed in this chapter. Does it require a mining methodology that is quite different from those outlined in this chapter?
9. List and describe the five *primitives* for specifying a data mining task.
10. Describe why *concept hierarchies* are useful in data mining.
11. *Outliers* are often discarded as noise. However, one person's garbage could be another's treasure. For example, exceptions in credit card transactions can help us detect the fraudulent use of credit cards. Taking fraudulence detection as an example, propose two methods that can be used to detect outliers and discuss which one is more reliable.
12. Recent applications pay special attention to spatiotemporal data streams. A *spatiotemporal data stream* contains spatial information that changes over time, and is in the form of stream data, i.e., the data flow in-and-out like possibly infinite streams.
  - (a) Present three application examples of spatiotemporal data streams.
  - (b) Discuss what kind of interesting knowledge can be mined from such data streams, with limited time and resources.
  - (c) Identify and discuss the major challenges in spatiotemporal data mining.
  - (d) Using one application example, sketch a method to mine one kind of knowledge from such stream data efficiently.
13. Describe the differences between the following approaches for the integration of a data mining system with a database or data warehouse system: *no coupling*, *loose coupling*, *semitight coupling*, and *tight coupling*. State which approach you think is the most popular, and why.
14. Describe three challenges to data mining regarding *data mining methodology* and *user interaction issues*.
15. What are the major challenges of mining a huge amount of data (such as billions of tuples) in comparison with mining a small amount of data (such as a few hundred tuple data set)?
16. Outline the major research challenges of data mining in one specific application domain, such as stream/sensor data analysis, spatiotemporal data analysis, or bioinformatics.

## 1.12 Bibliographic Notes

The book *Knowledge Discovery in Databases*, edited by Piatetsky-Shapiro and Frawley [PSF91], is an early collection of research papers on knowledge discovery from data. The book *Advances in Knowledge Discovery and Data Mining*, edited by Fayyad, Piatetsky-Shapiro, Smyth, and Uthurusamy [FPSSe96], is a collection of later research results on knowledge discovery and data mining. There have been many data mining books published in recent years, including *Predictive Data Mining* by Weiss and Indurkha [WI98], *Data Mining Solutions: Methods and Tools for Solving Real-World Problems* by Westphal and Blaxton [WB98], *Mastering Data Mining: The Art and Science of Customer Relationship Management* by Berry and Linoff [BL99], *Building Data Mining Applications for CRM* by Berson, Smith, and Thearling [BST99], *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations* by Witten and Frank [WF00a], *Principles of Data Mining (Adaptive Computation and Machine Learning)* by Hand, Mannila, and Smyth [HMS01], *The Elements of Statistical Learning* by Hastie, Tibshirani, and Friedman [HTF01], *Data Mining: Introductory and Advanced Topics* by Dunham, *Data Mining: Multimedia, Soft Computing, and Bioinformatics* by Mitra and Acharya [MA03], and *Introduction to Data Mining* by Tan, Steinbach and Kumar [TSK05]. There are also books containing collections of papers on particular aspects of knowledge discovery, such as *Machine Learning and Data Mining: Methods and Applications* edited by Michalski, Brakto, and Kubat [MBK98], and *Relational Data Mining* edited by Dzeroski and Lavrac [De01], as well as many tutorial notes on data mining in major database, data mining and machine learning conferences.

*KDD Nuggets* is a regular, free electronic newsletter containing information relevant to knowledge discovery and data mining, moderated by Piatetsky-Shapiro since 1991. The Internet site *KDNuggets*, located at <http://www.kdnuggets.com>, contains a good collection of KDD-related information.

The data mining community started its first international conference on knowledge discovery and data mining in 1995 [Fe95]. The conference evolved from the four international workshops on knowledge discovery in databases, held from 1989 to 1994 [PS89, PS91a, FUE93, Fe94]. ACM-SIGKDD, a Special Interest Group on Knowledge Discovery in Databases was set up under ACM in 1998. In 1999, ACM-SIGKDD organized the fifth international conference on knowledge discovery and data mining (KDD'99). IEEE Computer Science Society has organized its annual data mining conference, International Conference on Data Mining (ICDM), since 2001. SIAM (Society on Industrial and Applied Mathematics) has organized its annual data mining conference, SIAM Data Mining conference (SDM), since 2002. A dedicated journal, *Data Mining and Knowledge Discovery*, published by Kluwer Publishers, has been available since 1997. ACM-SIGKDD also publishes a biannual newsletter, *SIGKDD Explorations*. There are a few other international or regional conferences on data mining, such as the Pacific Asian Conference on Knowledge Discovery and Data Mining (PAKDD), the European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD), and the International Conference on Data Warehousing and Knowledge Discovery (DaWaK).

Research in data mining has also been published in books, conferences, and journals on databases, statistics, machine learning, and data visualization. References to such sources are listed below.

Popular textbooks on database systems include *Database Systems: The Complete Book* by Garcia-Molina, Ullman, and Widom [GMUW02], *Database Management Systems* by Ramakrishnan and Gehrke [RG00], *Database System Concepts* by Silberschatz, Korth, and Sudarshan [SKS02], and *Fundamentals of Database Systems* by Elmasri and Navathe [EN03]. For an edited collection of seminal articles on database systems, see *Readings in Database Systems* by Stonebraker and Hellerstein [SH98]. Many books on data warehouse technology, systems, and applications have been published in the last several years, such as *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling* by Kimball and M. Ross [KR02], *The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing, and Deploying Data Warehouses* by Kimball, Reeves, Ross et al. [KRRT98], *Mastering Data Warehouse Design: Relational and Dimensional Techniques* by Imhoff, Gallemmo and Geiger [IGG03], *Building the Data Warehouse* by Inmon [Inm96], and *OLAP Solutions: Building Multidimensional Information Systems* by Thomsen [Tho97]. A set of research papers on materialized views and data warehouse implementations were collected in *Materialized Views: Techniques, Implementations, and Applications* by Gupta and Mumick [GM99]. Chaudhuri and Dayal [CD97] present a comprehensive overview of data warehouse technology.

Research results relating to data mining and data warehousing have been published in the proceedings of many international database conferences, including the *ACM-SIGMOD International Conference on Management of*

*Data (SIGMOD)*, the *International Conference on Very Large Data Bases (VLDB)*, the *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, the *International Conference on Data Engineering (ICDE)*, the *International Conference on Extending Database Technology (EDBT)*, the *International Conference on Database Theory (ICDT)*, the *International Conference on Information and Knowledge Management (CIKM)*, the *International Conference on Database and Expert Systems Applications (DEXA)*, and the *International Symposium on Database Systems for Advanced Applications (DASFAA)*. Research in data mining is also published in major database journals, such as *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, *ACM Transactions on Database Systems (TODS)*, *Journal of ACM (JACM)*, *Information Systems*, *The VLDB Journal*, *Data and Knowledge Engineering*, *International Journal of Intelligent Information Systems (JIIS)*, and *Knowledge and Information Systems (KAIS)*.

Many effective data mining methods have been developed by statisticians and pattern recognition researchers, and introduced in a rich set of textbooks. An overview of classification from a statistical pattern recognition perspective can be found in *Pattern Classification* by Duda, Hart, Stork [DHS00]. There are also many textbooks covering different topics in statistical analysis, such as *Mathematical Statistics: Basic Ideas and Selected Topics* by Bickel and Doksum [BD01], *The Statistical Sleuth: A Course in Methods of Data Analysis* by Ramsey and Schafer [RS01], *Applied Linear Statistical Models* by Neter, Kutner, Nachtsheim, and Wasserman [NKNW96], *An Introduction to Generalized Linear Models* by Dobson [Dob90], *Applied Statistical Time Series Analysis* by Shumway [Shu88], and *Applied Multivariate Statistical Analysis* by Johnson and Wichern [JW92].

Research in statistics is published in the proceedings of several major statistical conferences, including *Joint Statistical Meetings*, *International Conference of the Royal Statistical Society*, and *Symposium on the Interface: Computing Science and Statistics*. Other sources of publication include the *Journal of the Royal Statistical Society*, *The Annals of Statistics*, *Journal of American Statistical Association*, *Technometrics*, and *Biometrika*.

Textbooks and reference books on machine learning include *Machine Learning, An Artificial Intelligence Approach*, Vols. 1–4, edited by Michalski et al. [MCM83, MCM86, KM90, MT94], *C4.5: Programs for Machine Learning* by Quinlan [Qui93], *Elements of Machine Learning* by Langley [Lan96], and *Machine Learning* by Mitchell [Mit97]. The book *Computer Systems That Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems* by Weiss and Kulikowski [WK91] compares classification and prediction methods from several different fields. For an edited collection of seminal articles on machine learning, see *Readings in Machine Learning* by Shavlik and Dietterich [SD90].

Machine learning research is published in the proceedings of several large machine learning and artificial intelligence conferences, including the *International Conference on Machine Learning (ML)*, the *ACM Conference on Computational Learning Theory (COLT)*, the *International Joint Conference on Artificial Intelligence (IJCAI)*, and the *American Association of Artificial Intelligence Conference (AAAI)*. Other sources of publication include major machine learning, artificial intelligence, pattern recognition, and knowledge system journals, some of which have been mentioned above. Others include *Machine Learning (ML)*, *Artificial Intelligence Journal (AI)*, *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, and *Cognitive Science*.

Pioneering work on data visualization techniques is described in *The Visual Display of Quantitative Information* [Tuf83], *Envisioning Information* [Tuf90], and *Visual Explanations : Images and Quantities, Evidence and Narrative* [Tuf97], all by Tufte, in addition to *Graphics and Graphic Information Processing* by Bertin [Ber81], *Visualizing Data* by Cleveland [Cle93], and *Information Visualization in Data Mining and Knowledge Discovery* edited by Fayyad, Grinstein, and Wierse [FGW01]. Major conferences and symposiums on visualization include *ACM Human Factors in Computing Systems (CHI)*, *Visualization*, and the *International Symposium on Information Visualization*. Research on visualization is also published in *Transactions on Visualization and Computer Graphics*, *Journal of Computational and Graphical Statistics*, and *IEEE Computer Graphics and Applications*.

The DMQL data mining query language was proposed by Han, Fu, Wang, et al. [HFW<sup>+</sup>96] for the *DBMiner* data mining system. Other examples include *Discovery Board* (formerly *Data Mine*) by Imielinski, Virmani, and Abdulghani [IVA96], and *MSQL* by Imielinski and Virmani [IV99]. *MINE RULE*, an SQL-like operator for mining single-dimensional association rules, was proposed by Meo, Psaila, and Ceri [MPC96] and extended by Baralis and Psaila [BP97]. Microsoft Corporation has made a major data mining standardization effort by proposing *OLE DB for Data Mining (DM)* [Cor00] and *DMX* language (<http://research.microsoft.com/dmx/DataMining/>).

An introduction to the data mining language primitives of *DMX* can be found in Appendix A of this book. An important and major data mining standardization effort organized by the DMG group ([www.dmg.org](http://www.dmg.org)) has proposed a mark-up language called PMML (Programming data Model Markup Language) [Ras04]. A description of PMML is given in Appendix B.

Architectures of data mining systems have been discussed by many researchers in conference panels and meetings. The recent design of data mining languages, such as [BP97, IV99, Cor00, Ras04], the proposal of on-line analytical mining, such as [Han98], and the study of optimization of data mining queries, such as [NLHP98, STA98, LNHP99], can be viewed as steps toward tight integration of data mining systems with database systems and data warehouse systems. Some data mining primitives, proposed by Sarawagi, Thomas, and Agrawal [STA98], can also be used as building blocks in relational or object-relational systems, for efficient implementation of data mining in such database systems.



## Chapter 2

# Data Preprocessing

Today's real-world databases are highly susceptible to noisy, missing, and inconsistent data due to their typically huge size (often several gigabytes or more), and their likely origin from multiple, heterogeneous sources. Low quality data will lead to low quality mining results. *"How can the data be preprocessed in order to help improve the quality of the data and, consequently, of the mining results? How can the data be preprocessed so as to improve the efficiency and ease of the mining process?"*

There are a number of data preprocessing techniques. *Data cleaning* can be applied to remove noise and correct inconsistencies in the data. *Data integration* merges data from multiple sources into a coherent data store, such as a data warehouse. *Data transformations*, such as normalization, may be applied. For example, normalization may improve the accuracy and efficiency of mining algorithms involving distance measurements. *Data reduction* can reduce the data size by aggregating, eliminating redundant features, or clustering, for instance. These techniques are not mutually exclusive; they may work together. For example, data cleaning can involve transformations to correct wrong data, such as by transforming all entries for a *date* field to a common format. Data processing techniques, when applied prior to mining, can substantially improve the overall quality of the patterns mined and/or the time required for the actual mining.

In this chapter, we introduce the basic concepts of data preprocessing in Section 2.1. Section 2.2 presents *descriptive data summarization*, which serves as a foundation for data preprocessing. Descriptive data summarization helps us study the general characteristics of the data and identify the presence of noise or outliers, which is useful for successful data cleaning and data integration. The methods for data preprocessing are organized into the following categories: *data cleaning* (Section 2.3), *data integration and transformation* (Section 2.4), and *data reduction* (Section 2.5). Concept hierarchies can be used in an alternative form of data reduction where we replace low-level data (such as raw values for *age*) by higher level concepts (such as *youth*, *middle-aged*, or *senior*). This form of data reduction is the topic of Section 2.6, wherein we discuss the automatic generation of concept hierarchies from numerical data using data discretization techniques. The automatic generation of concept hierarchies from categorical data is also described.

### 2.1 Why Preprocess the Data?

Imagine that you are a manager at *AllElectronics* and have been charged with analyzing the company's data with respect to the sales at your branch. You immediately set out to perform this task. You carefully inspect the company's database and data warehouse, identifying and selecting the attributes or dimensions to be included in your analysis, such as *item*, *price*, and *units\_sold*. Alas! You notice that several of the attributes for various tuples have no recorded value. For your analysis, you would like to include information as to whether each item purchased was advertised as on sale, yet you discover that this information has not been recorded. Furthermore, users of your database system have reported errors, unusual values, and inconsistencies in the data recorded for some transactions. In other words, the data you wish to analyze by data mining techniques are **incomplete**

(lacking attribute values or certain attributes of interest, or containing only aggregate data), **noisy** (containing errors, or *outlier* values that deviate from the expected), and **inconsistent** (e.g., containing discrepancies in the department codes used to categorize items). Welcome to the real world!

Incomplete, noisy, and inconsistent data are commonplace properties of large real-world databases and data warehouses. Incomplete data can occur for a number of reasons. Attributes of interest may not always be available, such as customer information for sales transaction data. Other data may not be included simply because it was not considered important at the time of entry. Relevant data may not be recorded due to a misunderstanding, or because of equipment malfunctions. Data that were inconsistent with other recorded data may have been deleted. Furthermore, the recording of the history or modifications to the data may have been overlooked. Missing data, particularly for tuples with missing values for some attributes, may need to be inferred.

There are many possible reasons for noisy data (having incorrect attribute values). The data collection instruments used may be faulty. There may have been human or computer errors occurring at data entry. Errors in data transmission can also occur. There may be technology limitations, such as limited buffer size for coordinating synchronized data transfer and consumption. Incorrect data may also result from inconsistencies in naming conventions or data codes used, or inconsistent formats for input fields, such as *date*. Duplicate tuples also require data cleaning.

**Data cleaning** routines work to “clean” the data by filling in missing values, smoothing noisy data, identifying or removing outliers, and resolving inconsistencies. If users believe the data are dirty, they are unlikely to trust the results of any data mining that has been applied to it. Furthermore, dirty data can cause confusion for the mining procedure, resulting in unreliable output. Although most mining routines have some procedures for dealing with incomplete or noisy data, they are not always robust. Instead, they may concentrate on avoiding overfitting the data to the function being modeled. Therefore, a useful preprocessing step is to run your data through some data cleaning routines. Section 2.3 discusses methods for cleaning up your data.

Getting back to your task at *AllElectronics*, suppose that you would like to include data from multiple sources in your analysis. This would involve integrating multiple databases, data cubes, or files, that is, **data integration**. Yet some attributes representing a given concept may have different names in different databases, causing inconsistencies and redundancies. For example, the attribute for customer identification may be referred to as *customer\_id* in one data store, and *cust\_id* in another. Naming inconsistencies may also occur for attribute values. For example, the same first name could be registered as “Bill” in one database, but “William” in another, and “B.” in the third. Furthermore, you suspect that some attributes may be inferred from others (e.g., annual revenue). Having a large amount of redundant data may slow down or confuse the knowledge discovery process. Clearly, in addition to data cleaning, steps must be taken to help avoid redundancies during data integration. Typically, data cleaning and data integration are performed as a preprocessing step when preparing the data for a data warehouse. Additional data cleaning may be performed to detect and remove redundancies that may have resulted from data integration.

Getting back to your data, you have decided, say, that you would like to use a distance-based mining algorithm for your analysis, such as neural networks, nearest-neighbor classifiers, or clustering.<sup>1</sup> Such methods provide better results if the data to be analyzed have been *normalized*, that is, scaled to a specific range such as [0.0, 1.0]. Your customer data, for example, contain the attributes *age* and *annual salary*. The *annual salary* attribute usually takes much larger values than *age*. Therefore, if the attributes are left unnormalized, the distance measurements taken on *annual salary* will generally outweigh distance measurements taken on *age*. Furthermore, it would be useful for your analysis to obtain aggregate information as to the sales per customer region—something that is not part of any precomputed data cube in your data warehouse. You soon realize that **data transformation** operations, such as normalization and aggregation, are additional data preprocessing procedures that would contribute towards the success of the mining process. Data integration and data transformation are discussed in Section 2.4.

“Hmmm,” you wonder, as you consider your data even further. “The data set I have selected for analysis is *HUGE*, which is sure to slow down the mining process. Is there any way I can reduce the size of my data set, without jeopardizing the data mining results?” **Data reduction** obtains a reduced representation of the data set that is much smaller in volume, yet produces the same (or almost the same) analytical results. There are a number

<sup>1</sup>Neural network and nearest-neighbor classifiers are described in Chapter 6, while clustering is discussed in Chapter 7.



of strategies for data reduction. These include *data aggregation* (e.g., building a data cube), *attribute subset selection* (e.g., removing irrelevant attributes through correlation analysis), *dimensionality reduction* (e.g., using encoding schemes such as minimum length encoding or wavelets), and *numerosity reduction* (e.g., “replacing” the data by alternative, smaller representations such as clusters or parametric models). Data reduction is the topic of Section 2.5. Data can also be “reduced” by *generalization* with the use of concept hierarchies, where low-level concepts, such as *city* for customer location, are replaced with higher-level concepts, such as *region* or *province\_or\_state*. A concept hierarchy organizes the concepts into varying levels of abstraction. *Data discretization* is a form of data reduction that is very useful for the automatic generation of concept hierarchies from numerical data. This is described in Section 2.6, along with the automatic generation of concept hierarchies for categorical data.

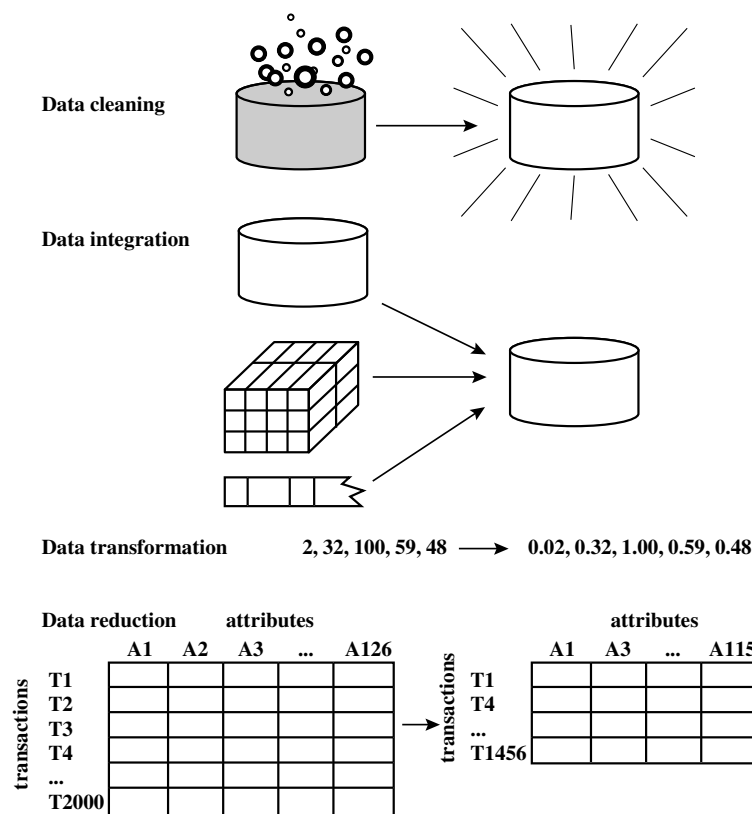


Figure 2.1: Forms of data preprocessing.

Figure 2.1 summarizes the data preprocessing steps described here. Note that the above categorization is not mutually exclusive. For example, the removal of redundant data may be seen as a form of data cleaning, as well as data reduction.

In summary, real-world data tend to be dirty, incomplete, and inconsistent. Data preprocessing techniques can improve the quality of the data, thereby helping to improve the accuracy and efficiency of the subsequent mining process. Data preprocessing is an important step in the knowledge discovery process, since quality decisions must be based on quality data. Detecting data anomalies, rectifying them early, and reducing the data to be analyzed can lead to huge payoffs for decision making.

## 2.2 Descriptive Data Summarization

For data preprocessing to be successful, it is essential to have an overall picture of your data. Descriptive data summarization techniques can be used to identify the typical properties of your data and highlight which data values should be treated as noise or outliers. Thus, we first introduce the basic concepts of descriptive data summarization before getting into the concrete workings of data preprocessing techniques.

For many data preprocessing tasks, users would like to learn about data characteristics regarding both central tendency and dispersion of the data. Measures of central tendency include *mean*, *median*, *mode*, and *midrange*, while measures of data dispersion include *quartiles*, *interquartile range (IQR)*, and *variance*. These descriptive statistics are of great help in understanding the distribution of the data. Such measures have been studied extensively in the statistical literature. From the data mining point of view, we need to examine how they can be computed efficiently in large databases. In particular, it is necessary to introduce the notions of *distributive measure*, *algebraic measure*, and *holistic measure*. Knowing what kind of measure we are dealing with can help us choose an efficient implementation for it.

### 2.2.1 Measuring the Central Tendency

In this section, we look at various ways to measure the central tendency of data. The most common and most effective numerical measure of the “center” of a set of data is the (*arithmetic*) *mean*. Let  $x_1, x_2, \dots, x_N$  be a set of  $N$  values or observations, such as for some attribute, like *salary*. The **mean** of this set of values is

$$\bar{x} = \frac{\sum_{i=1}^N x_i}{N} = \frac{x_1 + x_2 + \dots + x_N}{N}. \quad (2.1)$$

This corresponds to the built-in aggregate function, *average* (**avg()** in SQL), provided in relational database systems.

A **distributive measure** is a measure (i.e., function) that can be computed for a given data set by partitioning the data into smaller subsets, computing the measure for each subset, and then merging the results in order to arrive at the measure’s value for the original (entire) data set. Both **sum()** and **count()** are distributive measures since they can be computed in this manner. Other examples include **max()** and **min()**. An **algebraic measure** is a measure that can be computed by applying an algebraic function to one or more distributive measures. Hence, *average* (or **mean()**) is an algebraic measure since it can be computed by **sum()/count()**. When computing data cubes<sup>2</sup>, **sum()** and **count()** are typically saved in precomputation. Thus, the derivation of *average* for data cubes is straightforward.

Sometimes, each value  $x_i$  in a set may be associated with a weight  $w_i$ , for  $i = 1, \dots, N$ . The weights reflect the significance, importance, or occurrence frequency attached to their respective values. In this case, we can compute

$$\bar{x} = \frac{\sum_{i=1}^N w_i x_i}{\sum_{i=1}^N w_i} = \frac{w_1 x_1 + w_2 x_2 + \dots + w_N x_N}{w_1 + w_2 + \dots + w_N}. \quad (2.2)$$

This is called the **weighted arithmetic mean** or the **weighted average**. Note that the weighted average is another example of an algebraic measure.

Although the mean is the single most useful quantity for describing a data set, it is not always the best way of measuring the center of the data. A major problem with the *mean* is its sensitivity to extreme (e.g., outlier) values. Even a small number of extreme values can corrupt the mean. For example, the mean salary at a company may be substantially pushed up by that of a few highly paid managers. Similarly, the average score of a class in

<sup>2</sup>Data cube computation is described in detail in Chapters 3 and 4.

an exam could be pulled down quite a bit by a few very low scores. To offset the effect caused by a small number of extreme values, we can instead use the **trimmed mean**, which is the mean obtained after chopping off values at the high and low extremes. For example, we can sort the values observed for *salary* and remove the top and bottom 2% before computing the mean. We should avoid trimming too large a portion (such as 20%) at both ends as this can result in the loss of valuable information.

For skewed (asymmetric) data, a better measure of the center of data is the *median*. Suppose that a given data set of  $N$  distinct values is sorted in numerical order. If  $N$  is odd, then the **median** is the *middle value* of the ordered set; otherwise (i.e., if  $N$  is even), the median is the average of the middle two values.

A **holistic measure** is a measure that must be computed on the entire data set as a whole. It cannot be computed by partitioning the given data into subsets and merging the values obtained for the measure in each subset. The median is an example of a holistic measure. Holistic measures are much more expensive to compute than distributive measures such as those listed above.

Assume that data are grouped in intervals according to their  $x_i$  data values, and that the frequency (i.e., number of data values) of each interval is known. For example, people may be grouped according to their annual salary in intervals such as 10-20K, 20-30K, and so on. Let the interval that contains the median frequency be the *median interval*. We can then approximate the median of the entire data set (e.g., the median salary) by interpolation using the formula:

$$median = L_1 + \left( \frac{N/2 - (\sum freq)_l}{freq_{median}} \right) width, \quad (2.3)$$

where  $L_1$  is the lower boundary of the median interval,  $N$  is the number of values in the entire data set,  $(\sum freq)_l$  is the sum of the frequencies of all of the intervals that are lower than the median interval,  $freq_{median}$  is the frequency of the median interval, and *width* is the width of the median interval.

Another measure of central tendency is the *mode*. The **mode** for a set of data is the value that occurs most frequently in the set. It is possible for the greatest frequency to correspond to several different values, which results in more than one mode. Data sets with one, two, or three modes are respectively called **unimodal**, **bimodal**, and **trimodal**. In general, a data set with two or more modes is **multimodal**. At the other extreme, if each data value occurs only once, then there is no mode.

For unimodal frequency curves that are moderately skewed (asymmetrical), we have the following empirical relation:

$$mean - mode = 3 \times (mean - median). \quad (2.4)$$

This implies that the mode for unimodal frequency curves that are moderately skewed can easily be computed if the mean and median values are known.

In a unimodal frequency curve with perfect symmetric data distribution, the mean, median, and the mode are all at the same center value, as shown in Figure 2.2 (a). However, data in most real applications are not symmetric. They may instead be either positively skewed, where the mode occurs at a value that is smaller than the median (Figure 2.2 (b)), or negatively skewed, where the mode occurs at a value greater than the median (Figure 2.2 (c)).

The **midrange** can also be used to assess the central tendency of a data set. It is the average of the largest and smallest values in the set. This algebraic measure is easy to compute using the SQL aggregate functions, `max()` and `min()`.

### 2.2.2 Measuring the Dispersion of Data

The degree to which numerical data tend to spread is called the **dispersion**, or **variance** of the data. The most common measures of data dispersion are *range*, the *five-number summary* (based on *quartiles*), the *interquartile range*, and the *standard deviation*. Boxplots can be plotted based on the five-number summary and are a useful tool for identifying outliers.

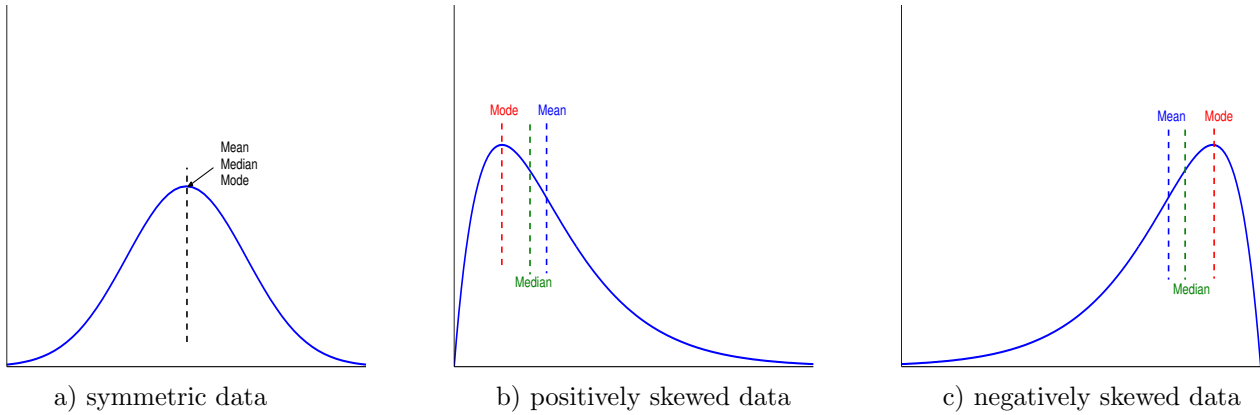


Figure 2.2: Median, mean, and mode of symmetric versus positively and negatively skewed data.

### Range, Quartiles, Outliers, and Boxplots

Let  $x_1, x_2, \dots, x_N$  be a set of observations for some attribute. The **range** of the set is the difference between the largest ( $\max()$ ) and smallest ( $\min()$ ) values. For the remainder for this section, let's assume that the data are sorted in increasing numerical order.

The  **$k$ th percentile** of a set of data in numerical order is the value  $x_i$  having the property that  $k$  percent of the data entries lie at or below  $x_i$ . Values at or below the *median* (discussed in the previous subsection) correspond to the 50th percentile.

The most commonly used percentiles other than the median are **quartiles**. The **first quartile**, denoted by  $Q_1$ , is the 25th percentile; the **third quartile**, denoted by  $Q_3$ , is the 75th percentile. The quartiles, including the median, give some indication of the center, spread, and shape of a distribution. The distance between the first and third quartiles is a simple measure of spread that gives the range covered by the middle half of the data. This distance is called the **interquartile range (IQR)** and is defined as

$$IQR = Q_3 - Q_1. \quad (2.5)$$

Based on reasoning similar to that in our analysis of the median in Section 2.2.1, we can conclude that  $Q_1$  and  $Q_3$  are holistic measures, as is  $IQR$ .

No single numerical measure of spread, such as  $IQR$ , is very useful for describing skewed distributions. The spreads of two sides of a skewed distribution are unequal (Figure 2.2). Therefore, it is more informative to also provide the two quartiles  $Q_1$  and  $Q_3$ , along with the median. A common rule of thumb for identifying suspected **outliers** is to single out values falling at least  $1.5 \times IQR$  above the third quartile or below the first quartile.

Because  $Q_1$ , the median, and  $Q_3$  together contain no information about the endpoints (e.g., tails) of the data, a fuller summary of the shape of a distribution can be obtained by providing the lowest and highest data values as well. This is known as the *five-number summary*. The **five-number summary** of a distribution consists of the median, the quartiles  $Q_1$  and  $Q_3$ , and the smallest and largest individual observations, written in the order *Minimum*,  $Q_1$ , *Median*,  $Q_3$ , *Maximum*.

**Boxplots** are a popular way of visualizing a distribution. A boxplot incorporates the five-number summary as follows:

- Typically, the ends of the box are at the quartiles, so that the box length is the interquartile range,  $IQR$ .
- The median is marked by a line within the box.
- Two lines (called *whiskers*) outside the box extend to the smallest (*Minimum*) and largest (*Maximum*) observations.

When dealing with a moderate number of observations, it is worthwhile to plot potential outliers individually. To do this in a boxplot, the whiskers are extended to the extreme low and high observations *only if* these values are less than  $1.5 \times IQR$  beyond the quartiles. Otherwise, the whiskers terminate at the most extreme observations occurring within  $1.5 \times IQR$  of the quartiles. The remaining cases are plotted individually. Boxplots can be used in the comparisons of several sets of compatible data. Figure 2.3 shows boxplots for unit price data for items sold at four branches of *AllElectronics* during a given time period. For branch 1, we see that the median price of items sold is \$80,  $Q_1$  is \$60,  $Q_3$  is \$100. [NEW Notice that two outlying observations for this branch were plotted individually as their values of 75 and 202 are more than 1.5 times the IQR here of 40.] The efficient computation of boxplots, or even *approximate boxplots* (based on approximates of the five-number summary) remains a challenging issue for the mining of large data sets.

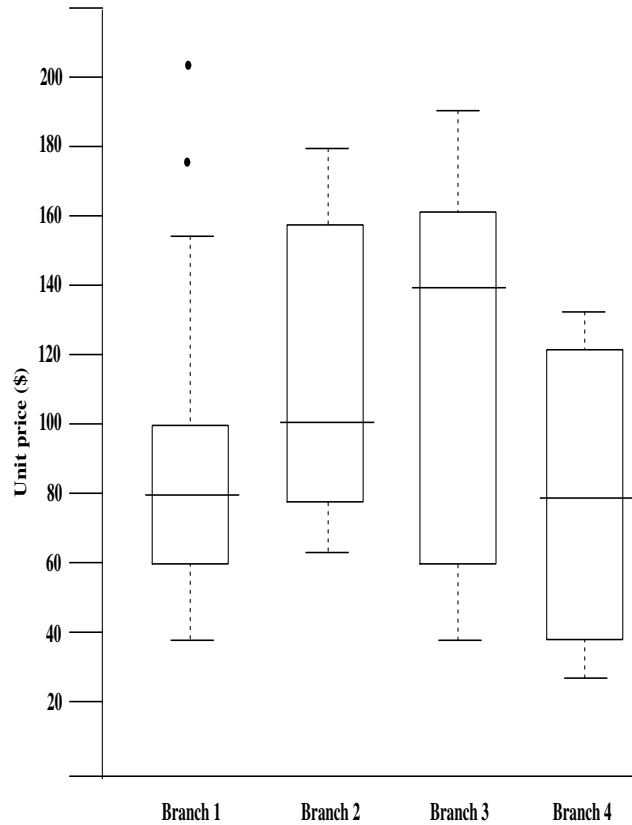


Figure 2.3: Boxplot for the unit price data for items sold at four branches of *AllElectronics* during a given time period.

### Variance and Standard Deviation

The **variance** of  $N$  observations,  $x_1, x_2, \dots, x_N$ , is

$$\sigma^2 = \frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2 = \frac{1}{N} \left[ \sum x_i^2 - \frac{1}{N} (\sum x_i)^2 \right], \quad (2.6)$$

where  $\bar{x}$  is the mean value of the observations, as defined in Equation (2.1). The **standard deviation**,  $\sigma$ , of the observations is the square root of the variance,  $\sigma^2$ .

The basic properties of the standard deviation,  $\sigma$ , as a measure of spread are

Table 2.1: A set of unit price data for items sold at a branch of *AllElectronics*.

Unit price (\$)	Count of items sold
40	275
43	300
47	250
..	..
74	360
75	515
78	540
..	..
115	320
117	270
120	350

- $\sigma$  measures spread about the mean and should be used only when the mean is chosen as the measure of center.
- $\sigma = 0$  only when there is no spread, that is, when all observations have the same value. Otherwise  $\sigma > 0$ .

The variance and standard deviation are algebraic measures since they can be computed from distributive measures. That is,  $N$  (which is `count()` in SQL),  $\sum x_i$  (which is the `sum()` of  $x_i$ ), and  $\sum x_i^2$  (which is the `sum()` of  $x_i^2$ ) can be computed in any partition and then merged to feed into the algebraic Equation (2.6). Thus the computation of the variance and standard deviation is scalable in large databases.

### 2.2.3 Graphic Displays of Basic Descriptive Data Summaries

Aside from the bar charts, pie charts, and line graphs used in most statistical or graphical data presentation software packages, there are other popular types of graphs for the display of data summaries and distributions. These include *histograms*, *quantile plots*, *q-q plots*, *scatter plots*, and *loess curves*. Such graphs are very helpful for the visual inspection of your data.

Plotting **histograms**, or **frequency histograms**, is a graphical method for summarizing the distribution of a given attribute. A histogram for an attribute  $A$  partitions the data distribution of  $A$  into disjoint subsets, or *buckets*. Typically, the width of each bucket is uniform. Each bucket is represented by a rectangle whose height is equal to the count or relative frequency of the values at the bucket. If  $A$  is categoric, such as *automobile\_model* or *item\_type*, then one rectangle is drawn for each known value of  $A$  and the resulting graph is more commonly referred to as a **bar chart**. If  $A$  is numeric, the term *histogram* is preferred. Partitioning rules for constructing histograms for numerical attributes are discussed in Section 2.5.4. In an equal-width histogram, for example, each bucket represents an equal-width range of numerical attribute  $A$ .

Figure 2.4 shows a histogram for the data set of Table 2.1, where buckets are defined by equal-width ranges representing \$20 increments and the frequency is the count of items sold. Histograms are at least a century old and are a widely used univariate graphical method. However, they may not be as effective as the quantile plot, q-q plot, and boxplot methods for comparing groups of univariate observations.

A **quantile plot** is a simple and effective way to have a first look at a univariate data distribution. First, it displays all of the data for the given attribute (allowing the user to assess both the overall behavior and unusual occurrences). Second, it plots quantile information. The mechanism used in this step is slightly different from the percentile computation discussed in Section 2.2.2. Let  $x_i$ , for  $i = 1$  to  $N$ , be the data sorted in increasing order so that  $x_1$  is the smallest observation and  $x_N$  is the largest. Each observation,  $x_i$ , is paired with a percentage,  $f_i$ , which indicates that approximately  $100f_i\%$  of the data are below or equal to the value,  $x_i$ . We say “approximately”



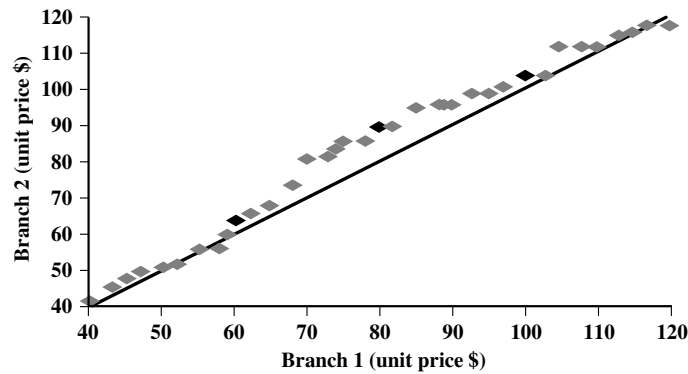


Figure 2.6: A quantile-quantile plot for unit price data from two different branches.

tronics during a given time period. Each point corresponds to the same quantile for each data set and shows the unit price of items sold at branch 1 versus branch 2 for that quantile. For example, here the lowest point in the left corner corresponds to the 0.03 quantile. (To aid in comparison, we also show a straight line that represents the case of when, for each given quantile, the unit price at each branch is the same. In addition, the darker points correspond to the data for  $Q_1$ , the median, and  $Q_3$ , respectively.) We see that at this quantile, the unit price of items sold at branch 1 was slightly less than that at branch 2. In other words, 3% of items sold at branch 1 were less than or equal to \$40, while 3% of items at branch 2 were less than or equal to \$42. At the highest quantile, we see that the unit price of items at branch 2 was slightly less than that at branch 1. In general, we note that there is a shift in the distribution of branch 1 with respect to branch 2 in that the unit prices of items sold at branch 1 tend to be lower than those at branch 2.

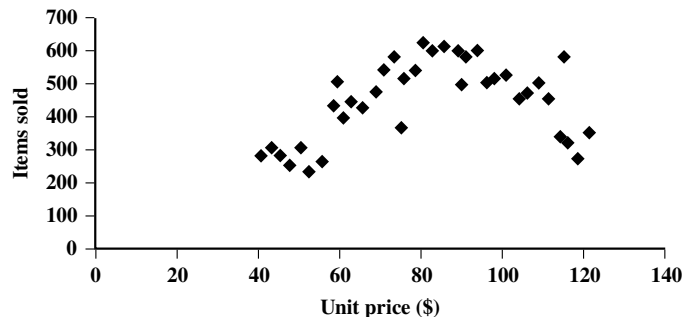


Figure 2.7: A scatter plot for the data set of Table 2.1.

A **scatter plot** is one of the most effective graphical methods for determining if there appears to be a relationship, pattern, or trend between two numerical attributes. To construct a scatter plot, each pair of values is treated as a pair of coordinates in an algebraic sense and plotted as points in the plane. Figure 2.7 shows a scatter plot for the set of data in Table 2.1. The scatter plot is a useful method for providing a first look at bivariate data to see clusters of points and outliers, or to explore the possibility of correlation relationships<sup>3</sup>. In Figure 2.8, we see examples of positive and negative correlations between two attributes in two different data sets. Figure 2.9 shows three cases that there is no correlation relationship between the two attributes in each of the given data sets.

When dealing with several attributes, the **scatter-plot matrix** is a useful extension to the scatter plot. Given  $n$  attributes, a scatter-plot matrix is an  $n \times n$  grid of scatter plots that provides a visualization of each attribute (or dimension) with every other attribute. The scatter-plot matrix becomes less effective as the number of attributes under study grows. In this case, user interactions such as zooming and panning become necessary to help interpret

<sup>3</sup>A statistical test for correlation is given in Section 2.4.1 on data integration (Equation (2.8)).



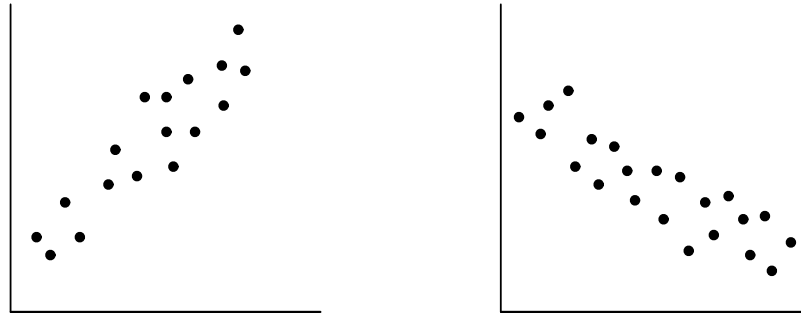


Figure 2.8: Scatter plots can be used to find (a) positive or (b) negative correlations between attributes.

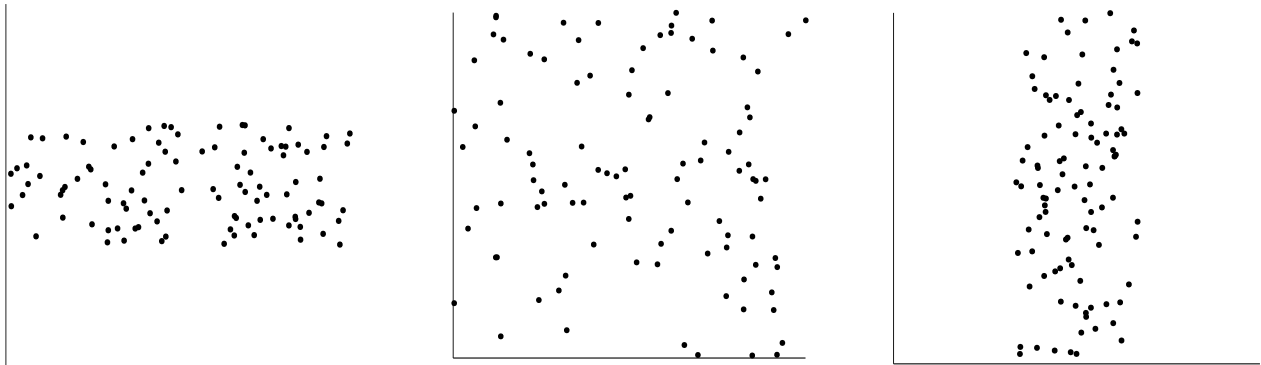


Figure 2.9: Three cases where there is no observed correlation between the two plotted attributes in each of the data sets.

the individual scatter plots effectively.

A **loess curve** is another important exploratory graphic aid that adds a smooth curve to a scatter plot in order to provide better perception of the pattern of dependence. The word *loess* is short for “local regression.” Figure 2.10 shows a loess curve for the set of data in Table 2.1.

To fit a loess curve, values need to be set for two parameters— $\alpha$ , a smoothing parameter, and  $\lambda$ , the degree of the polynomials that are fitted by the regression. While  $\alpha$  can be any positive number (typical values are between  $1/4$  and  $1$ ),  $\lambda$  can be  $1$  or  $2$ . The goal in choosing  $\alpha$  is to produce a fit that is as smooth as possible without unduly distorting the underlying pattern in the data. The curve becomes smoother as  $\alpha$  increases. There may be some lack of fit, however, indicating possible “missing” data patterns. If  $\alpha$  is very small, the underlying pattern is tracked, yet overfitting of the data may occur, where local “wiggles” in the curve may not be supported by the data. If the underlying pattern of the data has a “gentle” curvature with no local maxima and minima, then local linear fitting is usually sufficient ( $\lambda = 1$ ). However, if there are local maxima or minima, then local quadratic fitting ( $\lambda = 2$ ) typically does a better job of following the pattern of the data and maintaining local smoothness.

In conclusion, descriptive data summaries provide valuable insight into the overall behavior of your data. By helping to identify noise and outliers, they are especially useful for data cleaning.

## 2.3 Data Cleaning

Real-world data tend to be incomplete, noisy, and inconsistent. *Data cleaning* (or *data cleansing*) routines attempt to fill in missing values, smooth out noise while identifying outliers, and correct inconsistencies in the data. In this

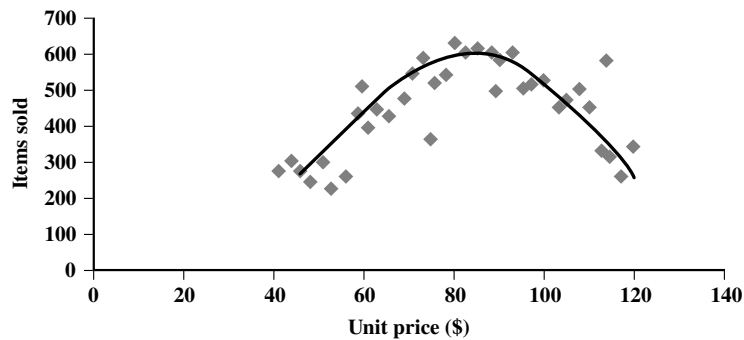


Figure 2.10: A loess curve for the data set of Table 2.1.

section, you will study basic methods for data cleaning. Section 2.3.1 looks at ways of handling missing values. Section 2.3.2 explains data smoothing techniques. Section 2.3.3 discusses approaches to data cleaning as a process.

### 2.3.1 Missing Values

Imagine that you need to analyze *Allelectronics* sales and customer data. You note that many tuples have no recorded value for several attributes, such as customer *income*. How can you go about filling in the missing values for this attribute? Let's look at the following methods:

1. **Ignore the tuple:** This is usually done when the class label is missing (assuming the mining task involves classification). This method is not very effective, unless the tuple contains several attributes with missing values. It is especially poor when the percentage of missing values per attribute varies considerably.
2. **Fill in the missing value manually:** In general, this approach is time-consuming and may not be feasible given a large data set with many missing values.
3. **Use a global constant to fill in the missing value:** Replace all missing attribute values by the same constant, such as a label like "*Unknown*" or  $-\infty$ . If missing values are replaced by, say, "*Unknown*," then the mining program may mistakenly think that they form an interesting concept, since they all have a value in common—that of "*Unknown*." Hence, although this method is simple, it is not foolproof.
4. **Use the attribute mean to fill in the missing value:** For example, suppose that the average income of *Allelectronics* customers is \$56,000. Use this value to replace the missing value for *income*.
5. **Use the attribute mean for all samples belonging to the same class as the given tuple:** For example, if classifying customers according to *credit\_risk*, replace the missing value with the average *income* value for customers in the same credit risk category as that of the given tuple.
6. **Use the most probable value to fill in the missing value:** This may be determined with regression, inference-based tools using a Bayesian formalism, or decision tree induction. For example, using the other customer attributes in your data set, you may construct a decision tree to predict the missing values for *income*. Decision trees, regression, and Bayesian inference are described in detail in Chapter 6.

Methods 3 to 6 bias the data. The filled-in value may not be correct. Method 6, however, is a popular strategy. In comparison to the other methods, it uses the most information from the present data to predict missing values. By considering the values of the other attributes in its estimation of the missing value for *income*, there is a greater chance that the relationships between *income* and the other attributes are preserved.

It is important to note that, in some cases, a missing value may not imply an error in the data! For example, when applying for a credit card, candidates may be asked to supply their driver's license number. Candidates who

do not have a driver's license may naturally leave this field blank. Forms should allow respondents to specify values such as "not applicable". Software routines may also be used to uncover other "null" values, such as "don't know", "?", or "none". Ideally, each attribute should have one or more rules regarding the *null* condition. The rules may specify whether or not nulls are allowed, and/or how such values should be handled or transformed. Fields may also be intentionally left blank if they are to be provided in a later step of the business process. Hence, although we can try our best to clean the data after it is seized, good design of databases and of data entry procedures should help minimize the number of missing values or errors in the first place.

### 2.3.2 Noisy Data

"What is noise?" **Noise** is a random error or variance in a measured variable. Given a numerical attribute such as, say, *price*, how can we "smooth" out the data to remove the noise? Let's look at the following data smoothing techniques:

**Sorted data for *price* (in dollars):** 4, 8, 15, 21, 21, 24, 25, 28, 34

**Partition into (equal-frequency) bins:**

Bin 1: 4, 8, 15  
Bin 2: 21, 21, 24  
Bin 3: 25, 28, 34

**Smoothing by bin means:**

Bin 1: 9, 9, 9  
Bin 2: 22, 22, 22  
Bin 3: 29, 29, 29

**Smoothing by bin boundaries:**

Bin 1: 4, 4, 15  
Bin 2: 21, 21, 24  
Bin 3: 25, 25, 34

Figure 2.11: Binning methods for data smoothing.

1. **Binning:** Binning methods smooth a sorted data value by consulting its "neighborhood," that is, the values around it. The sorted values are distributed into a number of "buckets," or *bins*. Because binning methods consult the neighborhood of values, they perform *local* smoothing. Figure 2.11 illustrates some binning techniques. In this example, the data for *price* are first sorted and then partitioned into *equal-frequency* bins of size 3 (i.e., each bin contains three values). In **smoothing by bin means**, each value in a bin is replaced by the mean value of the bin. For example, the mean of the values 4, 8, and 15 in Bin 1 is 9. Therefore, each original value in this bin is replaced by the value 9. Similarly, **smoothing by bin medians** can be employed, in which each bin value is replaced by the bin median. In **smoothing by bin boundaries**, the minimum and maximum values in a given bin are identified as the *bin boundaries*. Each bin value is then replaced by the closest boundary value. In general, the larger the width, the greater the effect of the smoothing. Alternatively, bins may be *equal-width*, where the interval range of values in each bin is constant. Binning is also used as a discretization technique and is further discussed in Section 2.6.
2. **Regression:** Data can be smoothed by fitting the data to a function, such as with regression. *Linear regression* involves finding the "best" line to fit two attributes (or variables), so that one attribute can be used to predict the other. *Multiple linear regression* is an extension of linear regression, where more than two attributes are involved and the data are fit to a multidimensional surface. Regression is further described in Section 2.5.4, as well as in Chapter 6.

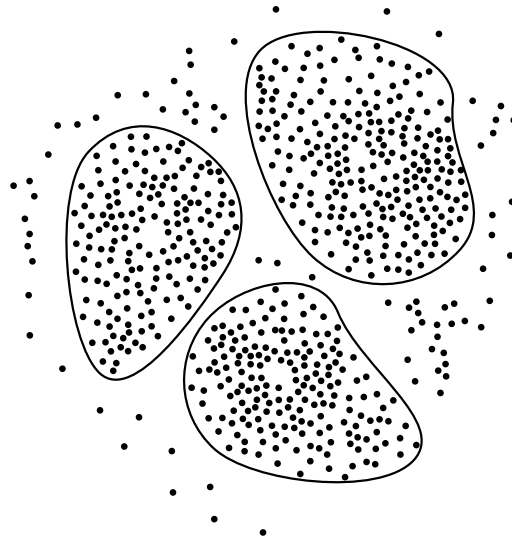


Figure 2.12: A 2-D plot of customer data with respect to customer locations in a city, showing three data clusters. Each cluster centroid is marked with a “+”, representing the average point in space for that cluster. Outliers may be detected as values that fall outside of the sets of clusters.

3. **Clustering:** Outliers may be detected by clustering, where similar values are organized into groups, or “clusters.” Intuitively, values that fall outside of the set of clusters may be considered outliers (Figure 2.12). Chapter 7 is dedicated to the topic of clustering and outlier analysis.

Many methods for data smoothing are also methods for data reduction involving discretization. For example, the binning techniques described above reduce the number of distinct values per attribute. This acts as a form of data reduction for logic-based data mining methods, such as decision tree induction, which repeatedly make value comparisons on sorted data. Concept hierarchies are a form of data discretization that can also be used for data smoothing. A concept hierarchy for *price*, for example, may map real *price* values into *inexpensive*, *moderately-priced*, and *expensive*, thereby reducing the number of data values to be handled by the mining process. Data discretization is discussed in Section 2.6. Some methods of classification, such as neural networks, have built-in data smoothing mechanisms. Classification is the topic of Chapter 6.

### 2.3.3 Data Cleaning as a Process

Missing values, noise, and inconsistencies contribute to inaccurate data. So far, we have looked at techniques for handling missing data and for smoothing data. “*But data cleaning is a big job. What about data cleaning as a process? How exactly does one proceed in tackling this task? Are there any tools out there to help?*”

The first step in data cleaning as a process is *discrepancy detection*. Discrepancies can be due to a number of causes, ranging from poorly designed data entry forms having many optional fields, human error in data entry, deliberate errors (e.g., respondents not wanting to divulge information about themselves), and data decay (e.g., outdated addresses). Discrepancies may also arise from inconsistent data representations and the inconsistent use of codes. Errors in instrumentation devices that record data, and system errors, are another source of discrepancies. Errors can also occur when the data are (inadequately) used for purposes other than originally intended. There may also be inconsistencies due to data integration, e.g., where a given attribute can have different names in different databases.<sup>4</sup>

“*So, how can we proceed with discrepancy detection?*” As a starting point, use any knowledge you may already

<sup>4</sup>Data integration and the removal of redundant data that can result from such integration are further described in Section 2.4.1.

have regarding properties of the data. Such knowledge or “data about data” is referred to as **metadata**. For example, what are the domain and data type of each attribute? What are the acceptable values for each attribute? What is the range of the length of values? Do all values fall within the expected range? Are there any known dependencies between attributes? The descriptive data summaries presented in Section 2.2 are useful here for grasping data trends and identifying anomalies. For example, values that are more than two standard deviations away from the mean for a given attribute may be flagged as potential outliers. In this step, you may write your own scripts and/or use some of the tools that we discuss further below. From this, you may find noise, outliers, and unusual values that need investigation.

As a data analyst, you should be on the lookout for the inconsistent use of codes and any inconsistent data representations (such as “2004/12/25” and “25/12/2004” for *date*). **Field overloading** is another source of errors that typically results when developers squeeze new attribute definitions into unused (bit) portions of already defined attributes (e.g., using a unused bit of an attribute whose value range uses only, say, 31 out of 32 bits).

The data should also be examined regarding uniqueness rules, consecutive rules, and null rules. A **unique rule** says that each value of the given attribute must be different from all other values for that attribute. A **consecutive rule** says that there can be no missing values between the lowest and highest values for the attribute, and that all values must also be unique (e.g., as in check numbers). A **null rule** specifies the use of blanks, questions marks, special characters, or other strings that may indicate the null condition (e.g., where a value for a given attribute is not available), and how such values should be handled. As mentioned in Section 2.3.1, reasons for missing values may include (1) the person originally asked to provide a value for the attribute refuses and/or finds that the information requested is not applicable (e.g., a *license-number* attribute left blank by non-drivers); (2) the data entry person does not know the correct value, or (3) the value is to be provided by a later step of the process. The null rule should specify how to record the null condition, e.g., such as to store zero for numerical attributes, a blank for character attributes, or any other conventions that may be in use (such as that entries like “don’t know” or “?” should be transformed to blank).

There are a number of different commercial tools that can aid in the step of discrepancy detection. **Data scrubbing tools** use simple domain knowledge (e.g., knowledge of postal addresses, and spell-checking) to detect errors and make corrections in the data. These tools rely on parsing and fuzzy matching techniques when cleaning data from multiple sources. **Data auditing tools** find discrepancies by analyzing the data to discover rules and relationships, and detecting data that violate such conditions. They are variants of data mining tools. For example, they may employ statistical analysis to find correlations, or clustering to identify outliers. They may also make use of the descriptive data summaries that were described in Section 2.2.

Some data inconsistencies may be corrected manually using external references. For example, errors made at data entry may be corrected by performing a paper trace. Most errors, however, will require *data transformations*. This is the second step in data cleaning as a process. That is, once we find discrepancies, we typically need to define and apply (a series of) transformations to correct them.

Commercial tools can assist in the data transformation step. **Data migration tools** allow simple transformations to be specified, such as to replace the string “*gender*” by “*sex*”. **ETL (Extraction/Transformation/Loading) tools** allow users to specify transforms through a graphical user interface (GUI). These tools typically support only a restricted set of transforms so that, often, we may also choose to write custom scripts for this step of the data cleaning process.

The two-step process of discrepancy detection and data transformation (to correct discrepancies) iterates. This process, however, is error-prone and time-consuming. Some transformations may introduce more discrepancies. Some *nested discrepancies* may only be detected after others have been fixed. For example, a typo such as “20004” in a year field may only surface once all date values have been converted to a uniform format. Transformations are often done as a batch process while the user waits without feedback. Only after the transformation is complete can the user go back and check that no new anomalies have been created by mistake. Typically, numerous iterations are required before the user is satisfied. Any tuples that cannot be automatically handled by a given transformation are typically written to a file without any explanation regarding the reasoning behind their failure. As a result, the entire data cleaning process also suffers from a lack of interactivity.

New approaches to data cleaning emphasize increased interactivity. Potter’s Wheel, for example, is a publicly

available data cleaning tool that integrates discrepancy detection and transformation. Users gradually build a series of transformations by composing and debugging individual transformations, one step at a time, on a spreadsheet-like interface. The transformations can be specified graphically or by providing examples. Results are shown immediately on the records that are visible on the screen. The user can choose to undo the transformations, so that transformations that introduced additional errors can be “erased”. The tool performs discrepancy checking automatically in the background on the latest transformed view of the data. Users can gradually develop and refine transformations as discrepancies are found, leading to more effective and efficient data cleaning.

Another approach to increased interactivity in data cleaning is the development of declarative languages for the specification of data transformation operators. Such work focusses on defining powerful extensions to SQL and algorithms that enable users to express data cleaning specifications efficiently.

As we discover more about the data, it is important to keep updating the metadata to reflect this knowledge. This will help speed up data cleaning on future versions of the same data store.

## 2.4 Data Integration and Transformation

Data mining often requires data integration—the merging of data from multiple data stores. The data may also need to be transformed into forms appropriate for mining. This section describes both data integration and data transformation.

### 2.4.1 Data Integration

It is likely that your data analysis task will involve *data integration*, which combines data from multiple sources into a coherent data store, as in data warehousing. These sources may include multiple databases, data cubes, or flat files.

There are a number of issues to consider during data integration. *Schema integration* and *object matching* can be tricky. How can equivalent real-world entities from multiple data sources be matched up? This is referred to as the **entity identification problem**. For example, how can the data analyst or the computer be sure that *customer\_id* in one database and *cust\_number* in another refer to the same attribute? Examples of metadata for each attribute include the name, meaning, data type, and range of values permitted for the attribute, and NULL rules for handling blank, zero, or NULL values (Section 2.3). Such metadata can be used to help avoid errors in schema integration. The metadata may also be used to help transform the data (e.g., where data codes for *pay\_type* in one database may be “H” and “S”, and 1 and 2 in another). Hence, this step also relates to data cleaning, as described earlier.

*Redundancy* is another important issue. An attribute (such as *annual revenue*, for instance) may be redundant if it can be “derived” from another attribute or set of attributes. Inconsistencies in attribute or dimension naming can also cause redundancies in the resulting data set.

Some redundancies can be detected by **correlation analysis**. Given two attributes, such analysis can measure how strongly one attribute implies the other, based on the available data. For numerical attributes, we can evaluate the correlation between two attributes, *A* and *B*, by computing the **correlation coefficient** (also known as *Pearson’s product moment coefficient*, named after its inventor, Karl Pearson). This is

$$r_{A,B} = \frac{\sum_{i=1}^N (a_i - \bar{A})(b_i - \bar{B})}{N\sigma_A\sigma_B} = \frac{\sum_{i=1}^N (a_i b_i) - N\bar{A}\bar{B}}{N\sigma_A\sigma_B} \quad (2.8)$$

where  $N$  is the number of tuples,  $a_i$  and  $b_i$  are the respective values of *A* and *B* in tuple  $i$ ,  $\bar{A}$  and  $\bar{B}$  are the respective mean values of *A* and *B*,  $\sigma_A$  and  $\sigma_B$  are the respective standard deviations of *A* and *B* (as defined in Section 2.2.2), and  $\sum(a_i b_i)$  is the sum of the *AB* cross-product (that is, for each tuple, the value for *A* is multiplied by the value for *B* in that tuple). Note that  $-1 \leq r_{A,B} \leq +1$ . If  $r_{A,B}$  is greater than 0, then *A* and *B* are positively

correlated, meaning that the values of  $A$  increase as the values of  $B$  increase. The higher the value, the stronger the correlation, i.e., the more each attribute implies the other. Hence, a higher value may indicate that  $A$  (or  $B$ ) may be removed as a redundancy. If the resulting value is equal to 0, then  $A$  and  $B$  are independent and there is no correlation between them. If the resulting value is less than 0, then  $A$  and  $B$  are negatively correlated, where the values of one attribute increase as the values of the other attribute decrease. This means that each attribute discourages the other. Scatter plots can also be used to view correlations between attributes (Section 2.2.3).

Note that correlation does not imply causality. That is, if  $A$  and  $B$  are correlated, this does not necessarily imply that  $A$  causes  $B$  or that  $B$  causes  $A$ . For example, in analyzing a demographic database, we may find that attributes representing the number of hospitals and the number of car-thefts in a region are correlated. This does not mean that one causes the other. Both are actually causally linked to a third attribute, namely, *population*.

For categorical (discrete) data, a correlation relationship between two attributes,  $A$  and  $B$ , can be discovered by a  $\chi^2$  (**chi-square**) test. Suppose  $A$  has  $c$  distinct values, namely  $a_1, a_2, \dots, a_c$ .  $B$  has  $r$  distinct values, namely  $b_1, b_2, \dots, b_r$ . The data tuples described by  $A$  and  $B$  can be shown as a **contingency table**, with the  $c$  values of  $A$  making up the columns and the  $r$  values of  $B$  making up the rows. Let  $(A_i, B_j)$  denote the event that attribute  $A$  takes on value  $a_i$  and attribute  $B$  takes on value  $b_j$ , i.e., where  $(A = a_i, B = b_j)$ . Each and every possible  $(A_i, B_j)$  joint event has its own cell (or slot) in the table. The  $\chi^2$  value (also known as the *Pearson  $\chi^2$  statistic*) is computed as:

$$\chi^2 = \sum_{i=1}^c \sum_{j=1}^r \frac{(o_{ij} - e_{ij})^2}{e_{ij}}, \quad (2.9)$$

where  $o_{ij}$  is the *observed frequency* (i.e., actual count) of the joint event  $(A_i, B_j)$ ; and  $e_{ij}$  is the *expected frequency* of  $(A_i, B_j)$ , which can be computed as

$$e_{ij} = \frac{\text{count}(A = a_i) \times \text{count}(B = b_j)}{N}, \quad (2.10)$$

where  $N$  is the number of data tuples,  $\text{count}(A = a_i)$  is the number of tuples having value  $a_i$  for  $A$ , and  $\text{count}(B = b_j)$  is the number of tuples having value  $b_j$  for  $B$ . The sum in Equation (2.9) is computed over all of the  $r \times c$  cells. Note that the cells that contribute the most to the  $\chi^2$  value are those whose actual count is very different from that expected.

The  $\chi^2$  statistic tests the hypothesis that  $A$  and  $B$  are independent. The test is based on a significance level, with  $(r - 1) \times (c - 1)$  degrees of freedom. We will illustrate the use of this statistic in an example below. If the hypothesis can be rejected, then we say that  $A$  and  $B$  are statistically related or associated.

Let's have a look at a concrete example.

**Example 2.1 Correlation analysis of categorical attributes using  $\chi^2$ .** Suppose that a group of 1,500 people was surveyed. The gender of each person was noted. Each person was polled as to whether their preferred type of reading material was fiction or nonfiction. Thus, we have two attributes, *gender* and *preferred\_reading*. The observed frequency (or count) of each possible joint event is summarized in the contingency table shown in Table 2.2, where the numbers in parentheses are the expected frequencies (calculated based on the data distribution for both attributes using Equation (2.10)).

Table 2.2: A  $2 \times 2$  contingency table for the data of Example 2.1. Are *gender* and *preferred\_Reading* correlated?

	<i>male</i>	<i>female</i>	Total
<i>fiction</i>	250 (90)	200 (360)	450
<i>non-fiction</i>	50 (210)	1000 (840)	1050
Total	300	1200	1500

Using Equation (2.10), we can verify the expected frequencies for each cell. For example, the expected frequency for the cell (male, fiction) is

$$e_{11} = \frac{\text{count}(\text{male}) \times \text{count}(\text{fiction})}{N} = \frac{300 \times 450}{1500} = 90,$$

and so on. Notice that in any row, the sum of the expected frequencies must equal the total observed frequency for that row, and the sum of the expected frequencies in any column must also equal the total observed frequency for that column. Using Equation (2.9) for  $\chi^2$  computation, we get

$$\chi^2 = \frac{(250 - 90)^2}{90} + \frac{(50 - 210)^2}{210} + \frac{(200 - 360)^2}{360} + \frac{(1000 - 840)^2}{840} = 284.44 + 121.90 + 71.11 + 30.48 = 507.93$$

For this  $2 \times 2$  table, the degrees of freedom are  $(2 - 1)(2 - 1) = 1$ . For 1 degree of freedom, the  $\chi^2$  value needed to reject the hypothesis at the 0.001 significance level is 10.828 (taken from the table of upper percentage points of the  $\chi^2$  distribution, typically available from any textbook on statistics). Since our computed value is above this, we can reject the hypothesis that *gender* and *preferred\_reading* are independent and conclude that the two attributes are (strongly) correlated for the given group of people. ■

In addition to detecting redundancies between attributes, duplication should also be detected at the tuple level (e.g., where there are two or more identical tuples for a given unique data entry case). The use of denormalized tables (often done to improve performance by avoiding joins) is another source of data redundancy. Inconsistencies often arise between various duplicates, due to inaccurate data entry or updating some but not all of the occurrences of the data. For example, if a purchase order database contains attributes for the purchaser's name and address instead of a key to this information in a purchaser database, discrepancies can occur, such as the same purchaser's name appearing with different addresses within the purchase order database.

A third important issue in data integration is the *detection and resolution of data value conflicts*. For example, for the same real-world entity, attribute values from different sources may differ. This may be due to differences in representation, scaling, or encoding. For instance, a *weight* attribute may be stored in metric units in one system and British imperial units in another. For a hotel chain, the *price* of rooms in different cities may involve not only different currencies but also different services (such as free breakfast) and taxes. An attribute in one system may be recorded at a lower level of abstraction than the "same" attribute in another. For example, the *total\_sales* in one database may refer to one branch of *All\_Electronics*, while an attribute of the same name in another database may refer to the total sales for *All\_Electronics* stores in a given region.

When matching attributes from one database to another during integration, special attention must be paid to the *structure* of the data. This is to ensure that any attribute functional dependencies and referential constraints in the source system match those in the target system. For example, in one system, a *discount* may be applied to the order, while in another system it is applied to each individual line item within the order. If this is not caught prior to integration, items in the target system may be improperly discounted.

The semantic heterogeneity and structure of data pose great challenges in data integration. Careful integration of the data from multiple sources can help reduce and avoid redundancies and inconsistencies in the resulting data set. This can help improve the accuracy and speed of the subsequent mining process.

## 2.4.2 Data Transformation

In *data transformation*, the data are transformed or consolidated into forms appropriate for mining. Data transformation can involve the following:

- **Smoothing**, which works to remove noise from the data. Such techniques include binning, regression, and clustering.
- **Aggregation**, where summary or aggregation operations are applied to the data. For example, the daily sales data may be aggregated so as to compute monthly and annual total amounts. This step is typically used in constructing a data cube for analysis of the data at multiple granularities.



- **Generalization** of the data, where low-level or “primitive” (raw) data are replaced by higher-level concepts through the use of concept hierarchies. For example, categorical attributes, like *street*, can be generalized to higher-level concepts, like *city* or *country*. Similarly, values for numerical attributes, like *age*, may be mapped to higher-level concepts, like *youth*, *middle-aged*, and *senior*.
- **Normalization**, where the attribute data are scaled so as to fall within a small specified range, such as  $-1.0$  to  $1.0$ , or  $0.0$  to  $1.0$ .
- **Attribute construction** (or *feature construction*), where new attributes are constructed and added from the given set of attributes to help the mining process.

Smoothing is a form of data cleaning and was addressed in Section 2.3.2. Section 2.3.3 on the data cleaning process also discussed ETL tools, where users specify transformations to correct data inconsistencies. Aggregation and generalization serve as forms of data reduction and are discussed in Sections 2.5 and 2.6, respectively. In this section, we therefore discuss normalization and attribute construction.

An attribute is normalized by scaling its values so that they fall within a small specified range, such as  $0.0$  to  $1.0$ . Normalization is particularly useful for classification algorithms involving neural networks, or distance measurements such as nearest neighbor classification and clustering. If using the neural network backpropagation algorithm for classification mining (Chapter 6), normalizing the input values for each attribute measured in the training tuples will help speed up the learning phase. For distance-based methods, normalization helps prevent attributes with initially large ranges (e.g., *income*) from outweighing attributes with initially smaller ranges (e.g., binary attributes). There are many methods for data normalization. We study three: *min-max normalization*, *z-score normalization*, and *normalization by decimal scaling*.

**Min-max normalization** performs a linear transformation on the original data. Suppose that  $\min_A$  and  $\max_A$  are the minimum and maximum values of an attribute,  $A$ . Min-max normalization maps a value,  $v$ , of  $A$  to  $v'$  in the range  $[\text{new\_min}_A, \text{new\_max}_A]$  by computing

$$v' = \frac{v - \min_A}{\max_A - \min_A}(\text{new\_max}_A - \text{new\_min}_A) + \text{new\_min}_A. \quad (2.11)$$

Min-max normalization preserves the relationships among the original data values. It will encounter an “out of bounds” error if a future input case for normalization falls outside of the original data range for  $A$ .

**Example 2.2 Min-max normalization.** Suppose that the minimum and maximum values for the attribute *income* are \$12,000 and \$98,000, respectively. We would like to map *income* to the range  $[0.0, 1.0]$ . By min-max normalization, a value of \$73,600 for *income* is transformed to  $\frac{73,600 - 12,000}{98,000 - 12,000}(1.0 - 0) + 0 = 0.716$ . ■

In **z-score normalization** (or *zero-mean normalization*), the values for an attribute,  $A$ , are normalized based on the mean and standard deviation of  $A$ . A value,  $v$ , of  $A$  is normalized to  $v'$  by computing

$$v' = \frac{v - \bar{A}}{\sigma_A} \quad (2.12)$$

where  $\bar{A}$  and  $\sigma_A$  are the mean and standard deviation, respectively, of attribute  $A$ . This method of normalization is useful when the actual minimum and maximum of attribute  $A$  are unknown, or when there are outliers that dominate the min-max normalization.

**Example 2.3 z-score normalization** Suppose that the mean and standard deviation of the values for the attribute *income* are \$54,000 and \$16,000, respectively. With z-score normalization, a value of \$73,600 for *income* is transformed to  $\frac{73,600 - 54,000}{16,000} = 1.225$ . ■

**Normalization by decimal scaling** normalizes by moving the decimal point of values of attribute  $A$ . The number of decimal points moved depends on the maximum absolute value of  $A$ . A value,  $v$ , of  $A$  is normalized to



2.5.1 Data Cube Aggregation

Imagine that you have collected the data for your analysis. These data consist of the *AllElectronics* sales per quarter, for the years 2002 to 2004. You are, however, interested in the annual sales (total per year), rather than the total per quarter. Thus the data can be *aggregated* so that the resulting data summarize the total sales per year instead of per quarter. This aggregation is illustrated in Figure 2.13. The resulting data set is smaller in volume, without loss of information necessary for the analysis task.

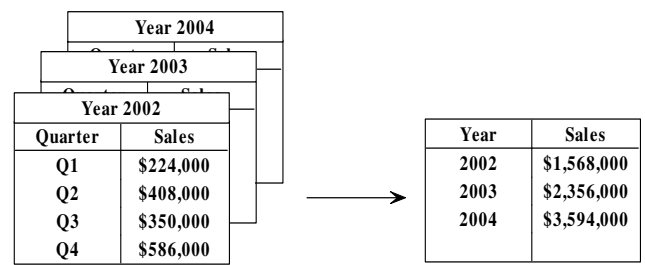


Figure 2.13: Sales data for a given branch of *AllElectronics* for the years 2002 to 2004. On the left, the sales are shown per quarter. On the right, the data are aggregated to provide the annual sales.

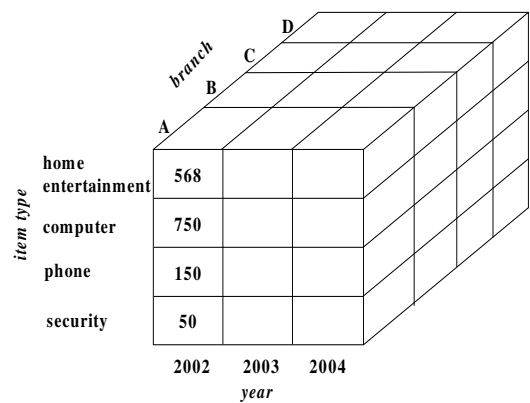


Figure 2.14: A data cube for sales at *AllElectronics*.

Data cubes are discussed in detail in Chapter 3 on data warehousing. We briefly introduce some concepts here. Data cubes store multidimensional aggregated information. For example, Figure 2.14 shows a data cube for multidimensional analysis of sales data with respect to annual sales per item type for each *AllElectronics* branch. Each cell holds an aggregate data value, corresponding to the data point in multidimensional space. Concept hierarchies may exist for each attribute, allowing the analysis of data at multiple levels of abstraction. For example, a hierarchy for *branch* could allow branches to be grouped into regions, based on their address. Data cubes provide fast access to precomputed, summarized data, thereby benefiting on-line analytical processing as well as data mining.

The cube created at the lowest level of abstraction is referred to as the *base cuboid*. The base cuboid should correspond to an individual entity of interest, such as *sales* or *customer*. In other words, the lowest level should be usable, or useful for the analysis. A cube at the highest level of abstraction is the *apex cuboid*. For the sales data of Figure 2.14, the apex cuboid would give one total—the total *sales* for all three years, for all item types, and for all branches. Data cubes created for varying levels of abstraction are often referred to as *cuboids*, so that a data cube may instead refer to a *lattice of cuboids*. Each higher level of abstraction further reduces the resulting data size. When replying to data mining requests, the *smallest* available cuboid relevant to the given task should

be used. This issue is also addressed in Chapter 3.

### 2.5.2 Attribute Subset Selection

Data sets for analysis may contain hundreds of attributes, many of which may be irrelevant to the mining task, or redundant. For example, if the task is to classify customers as to whether or not they are likely to purchase a popular new CD at *Allelectronics* when notified of a sale, attributes such as the customer's telephone number are likely to be irrelevant, unlike attributes such as *age* or *music\_taste*. Although it may be possible for a domain expert to pick out some of the useful attributes, this can be a difficult and time-consuming task, especially when the behavior of the data is not well known (hence, a reason behind its analysis!). Leaving out relevant attributes or keeping irrelevant attributes may be detrimental, causing confusion for the mining algorithm employed. This can result in discovered patterns of poor quality. In addition, the added volume of irrelevant or redundant attributes can slow down the mining process.

**Attribute subset selection**<sup>6</sup> reduces the data set size by removing irrelevant or redundant attributes (or dimensions). The goal of attribute subset selection is to find a minimum set of attributes such that the resulting probability distribution of the data classes is as close as possible to the original distribution obtained using all attributes. Mining on a reduced set of attributes has an additional benefit. It reduces the number of attributes appearing in the discovered patterns, helping to make the patterns easier to understand.

*“How can we find a ‘good’ subset of the original attributes?”* For  $n$  attributes, there are  $2^n$  possible subsets. An exhaustive search for the optimal subset of attributes can be prohibitively expensive, especially as  $n$  and the number of data classes increase. Therefore, heuristic methods that explore a reduced search space are commonly used for attribute subset selection. These methods are typically **greedy** in that, while searching through attribute space, they always make what looks to be the best choice at the time. Their strategy is to make a locally optimal choice in the hope that this will lead to a globally optimal solution. Such greedy methods are effective in practice and may come close to estimating an optimal solution.

The “best” (and “worst”) attributes are typically determined using tests of statistical significance, which assume that the attributes are independent of one another. Many other attribute evaluation measures can be used, such as the *information gain* measure used in building decision trees for classification.<sup>7</sup>

Basic heuristic methods of attribute subset selection include the following techniques, some of which are illustrated in Figure 2.15.

1. **Stepwise forward selection:** The procedure starts with an empty set of attributes as the reduced set. The best of the original attributes is determined and added to the reduced set. At each subsequent iteration or step, the best of the remaining original attributes is added to the set.
2. **Stepwise backward elimination:** The procedure starts with the full set of attributes. At each step, it removes the worst attribute remaining in the set.
3. **Combination of forward selection and backward elimination:** The stepwise forward selection and backward elimination methods can be combined so that, at each step, the procedure selects the best attribute and removes the worst from among the remaining attributes.
4. **Decision tree induction:** Decision tree algorithms, such as ID3, C4.5, and CART, were originally intended for classification. Decision tree induction constructs a flow-chart-like structure where each internal (nonleaf) node denotes a test on an attribute, each branch corresponds to an outcome of the test, and each external (leaf) node denotes a class prediction. At each node, the algorithm chooses the “best” attribute to partition the data into individual classes.

<sup>6</sup>In machine learning, attribute subset selection is known as *feature subset selection*.

<sup>7</sup>The information gain measure is described in detail in Chapter 6. It is briefly described in Section 2.6.1 with respect to attribute discretization.

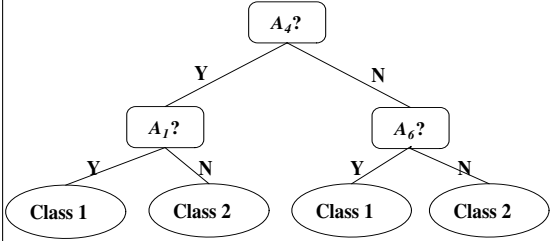
Forward selection	Backward elimination	Decision tree induction
<p><b>Initial attribute set:</b>  <math>\{A_1, A_2, A_3, A_4, A_5, A_6\}</math></p> <p><b>Initial reduced set:</b>  <math>\{\}</math>  <math>\Rightarrow \{A_1\}</math>  <math>\Rightarrow \{A_1, A_4\}</math>  <math>\Rightarrow</math> <b>Reduced attribute set:</b>  <math>\{A_1, A_4, A_6\}</math></p>	<p><b>Initial attribute set:</b>  <math>\{A_1, A_2, A_3, A_4, A_5, A_6\}</math></p> <p><math>\Rightarrow \{A_1, A_3, A_4, A_5, A_6\}</math>  <math>\Rightarrow \{A_1, A_4, A_5, A_6\}</math>  <math>\Rightarrow</math> <b>Reduced attribute set:</b>  <math>\{A_1, A_4, A_6\}</math></p>	<p><b>Initial attribute set:</b>  <math>\{A_1, A_2, A_3, A_4, A_5, A_6\}</math></p>  <p><math>\Rightarrow</math> <b>Reduced attribute set:</b>  <math>\{A_1, A_4, A_6\}</math></p>

Figure 2.15: Greedy (heuristic) methods for attribute subset selection.

When decision tree induction is used for attribute subset selection, a tree is constructed from the given data. All attributes that do not appear in the tree are assumed to be irrelevant. The set of attributes appearing in the tree form the reduced subset of attributes.

The stopping criteria for the methods may vary. The procedure may employ a threshold on the measure used to determine when to stop the attribute selection process.

### 2.5.3 Dimensionality Reduction

In *dimensionality reduction*, data encoding or transformations are applied so as to obtain a reduced or “compressed” representation of the original data. If the original data can be *reconstructed* from the compressed data without any loss of information, the data reduction is called **lossless**. If, instead, we can reconstruct only an approximation of the original data, then the data reduction is called **lossy**. There are several well-tuned algorithms for string compression. Although they are typically lossless, they allow only limited manipulation of the data. In this section, we instead focus on two popular and effective methods of lossy [NEW data compression]dimensionality reduction: *wavelet transforms* and *principal components analysis*.

#### Wavelet Transforms

The **discrete wavelet transform (DWT)** is a linear signal processing technique that, when applied to a data vector  $\mathbf{X}$ , transforms it to a numerically different vector,  $\mathbf{X}'$ , of **wavelet coefficients**. The two vectors are of the same length. When applying this technique to data reduction, we consider each tuple as an  $n$ -dimensional data vector, i.e.,  $\mathbf{X} = (x_1, x_2, \dots, x_n)$ , depicting  $n$  measurements made on the tuple from  $n$  database attributes.<sup>8</sup>

“How can this technique be useful for data reduction if the wavelet transformed data are of the same length as the original data?” The usefulness lies in the fact that the wavelet transformed data can be truncated. A compressed approximation of the data can be retained by storing only a small fraction of the strongest of the wavelet coefficients. For example, all wavelet coefficients larger than some user-specified threshold can be retained. All other coefficients are set to 0. The resulting data representation is therefore very sparse, so that operations that can take advantage of data sparsity are computationally very fast if performed in wavelet space. The technique also

<sup>8</sup>In our notation, any variable representing a vector is shown in bold italic font; measurements depicting the vector are shown in italic font.

works to remove noise without smoothing out the main features of the data, making it effective for data cleaning as well. Given a set of coefficients, an approximation of the original data can be constructed by applying the *inverse* of the DWT used.

The DWT is closely related to the *discrete Fourier transform (DFT)*, a signal processing technique involving sines and cosines. In general, however, the DWT achieves better lossy compression. That is, if the same number of coefficients is retained for a DWT and a DFT of a given data vector, the DWT version will provide a more accurate approximation of the original data. Hence, for an equivalent approximation, the DWT requires less space than the DFT. Unlike the DFT, wavelets are quite localized in space, contributing to the conservation of local detail.

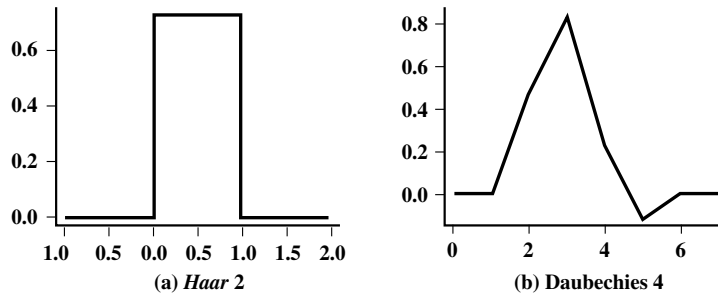


Figure 2.16: Examples of wavelet families. The number next to a wavelet name is the number of *vanishing moments* of the wavelet. This is a set of mathematical relationships that the coefficients must satisfy and is related to the number of coefficients.

There is only one DFT, yet there are several families of DWTs. Figure 2.16 shows some wavelet families. Popular wavelet transforms include the Haar\_2, Daubechies\_4, and Daubechies\_6 transforms. The general procedure for applying a discrete wavelet transform uses a hierarchical *pyramid algorithm* that halves the data at each iteration, resulting in fast computational speed. The method is as follows:

1. The length,  $L$ , of the input data vector must be an integer power of 2. This condition can be met by padding the data vector with zeros, as necessary ( $L \geq n$ ).
2. Each transform involves applying two functions. The first applies some data smoothing, such as a sum or weighted average. The second performs a weighted difference, which acts to bring out the detailed features of the data.
3. The two functions are applied to pairs of data points in  $\mathbf{X}$ , that is, to all pairs of measurements  $(x_{2i}, x_{2i+1})$ . This results in two sets of data of length  $L/2$ . In general, these represent a smoothed or low frequency version of the input data, and the high frequency content of it, respectively.
4. The two functions are recursively applied to the sets of data obtained in the previous loop, until the resulting data sets obtained are of length 2.
5. A selection of values from the data sets obtained in the above iterations are designated the wavelet coefficients of the transformed data.

Equivalently, a matrix multiplication can be applied to the input data in order to obtain the wavelet coefficients, where the matrix used depends on the given DWT. The matrix must be **orthonormal**, meaning that the columns are unit vectors and are mutually orthogonal, so that the matrix inverse is just its transpose. Although we do not have room to discuss it here, this property allows the reconstruction of the data from the smooth and smooth-difference data sets. By factoring the matrix used into a product of a few sparse matrices, the resulting “*fast DWT*” algorithm has a complexity of  $O(n)$  for an input vector of length  $n$ .

Wavelet transforms can be applied to multidimensional data, such as a data cube. This is done by first applying the transform to the first dimension, then to the second, and so on. The computational complexity involved is linear

with respect to the number of cells in the cube. Wavelet transforms give good results on sparse or skewed data and on data with ordered attributes. Lossy compression by wavelets is reportedly better than JPEG compression, the current commercial standard. Wavelet transforms have many real-world applications, including the compression of fingerprint images, computer vision, analysis of time-series data, and data cleaning.

### Principal Components Analysis

In this subsection we provide an intuitive introduction to principal components analysis as a method of dimensionality reduction. A detailed theoretical explanation is beyond the scope of this book.

Suppose that the data to be reduced consist of tuples or data vectors described by  $n$  attributes or dimensions. **Principal components analysis**, or **PCA** (also called the Karhunen-Loeve, or K-L method), searches for  $k$   $n$ -dimensional orthogonal vectors that can best be used to represent the data, where  $k \leq n$ . The original data are thus projected onto a much smaller space, resulting in dimensionality reduction. Unlike attribute subset selection, which reduces the attribute set size by retaining a subset of the initial set of attributes, PCA “combines” the essence of attributes by creating an alternative, smaller set of variables. The initial data can then be projected onto this smaller set. PCA often reveals relationships that were not previously suspected and thereby allows interpretations that would not ordinarily result.

The basic procedure is as follows:

1. The input data are normalized, so that each attribute falls within the same range. This step helps ensure that attributes with large domains will not dominate attributes with smaller domains.
2. PCA computes  $k$  orthonormal vectors that provide a basis for the normalized input data. These are unit vectors that each point in a direction perpendicular to the others. These vectors are referred to as the *principal components*. The input data are a linear combination of the principal components.
3. The principal components are sorted in order of decreasing “significance” or strength. The principal components essentially serve as a new set of axes for the data, providing important information about variance. That is, the sorted axes are such that the first axis shows the most variance among the data, the second axis shows the next highest variance, and so on. For example, Figure 2.17 shows the first two principal components,  $Y_1$  and  $Y_2$ , for the given set of data originally mapped to the axes,  $X_1$  and  $X_2$ . This information helps identify groups or patterns within the data.

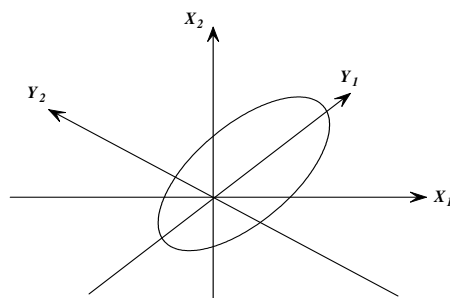


Figure 2.17: Principal components analysis.  $Y_1$  and  $Y_2$  are the first two principal components for the given data.

4. Since the components are sorted according to decreasing order of “significance,” the size of the data can be reduced by eliminating the weaker components, that is, those with low variance. Using the strongest principal components, it should be possible to reconstruct a good approximation of the original data.

PCA is computationally inexpensive, can be applied to ordered and unordered attributes, and can handle sparse data and skewed data. Multidimensional data of more than two dimensions can be handled by reducing

the problem to two dimensions. Principal components may be used as inputs to multiple regression and cluster analysis. In comparison with wavelet transforms, PCA tends to be better at handling sparse data, while wavelet transforms are more suitable for data of high dimensionality.

### 2.5.4 Numerosity Reduction

“Can we reduce the data volume by choosing alternative, ‘smaller’ forms of data representation?” Techniques of *numerosity reduction* can indeed be applied for this purpose. These techniques may be parametric or nonparametric. For *parametric methods*, a model is used to estimate the data, so that typically only the data parameters need be stored, instead of the actual data. (Outliers may also be stored.) Log-linear models, which estimate discrete multidimensional probability distributions, are an example. *Nonparametric methods* for storing reduced representations of the data include histograms, clustering, and sampling.

Let’s have a look at each of the numerosity reduction techniques mentioned above.

#### Regression and Log-Linear Models

Regression and log-linear models can be used to approximate the given data. In (simple) **linear regression**, the data are modeled to fit a straight line. For example, a random variable,  $y$  (called a *response variable*), can be modeled as a linear function of another random variable,  $x$  (called a *predictor variable*), with the equation

$$y = wx + b, \quad (2.14)$$

where the variance of  $y$  is assumed to be constant. In the context of data mining,  $x$  and  $y$  are numerical database attributes. The coefficients,  $w$  and  $b$  (called *regression coefficients*), specify the slope of the line and the  $Y$ -intercept, respectively. These coefficients can be solved for by the *method of least squares*, which minimizes the error between the actual line separating the data and the estimate of the line. **Multiple linear regression** is an extension of (simple) linear regression, which allows a response variable,  $y$ , to be modeled as a linear function of two or more predictor variables.

**Log-linear models** approximate discrete multidimensional probability distributions. Given a set of tuples in  $n$  dimensions (e.g., described by  $n$  attributes), we can consider each tuple as a point in an  $n$ -dimensional space. Log-linear models can be used to estimate the probability of each point in a multidimensional space for a set of discretized attributes, based on a smaller subset of dimensional combinations. This allows a higher-dimensional data space to be constructed from lower-dimensional spaces. Log-linear models are therefore also useful for [OLD: data compression][NEW: dimensionality reduction] (since the lower-dimensional points together typically occupy less space than the original data points) and data smoothing (since aggregate estimates in the lower-dimensional space are less subject to sampling variations than the estimates in the higher dimensional space).

Regression and log-linear models can both be used on sparse data although their application may be limited. While both methods can handle skewed data, regression does exceptionally well. Regression can be computationally intensive when applied to high-dimensional data, while log-linear models show good scalability for up to 10 or so dimensions. Regression and log-linear models are further discussed in Section 6.10.

#### Histograms

Histograms use binning to approximate data distributions and are a popular form of data reduction. Histograms were introduced in Section 2.2.3. A **histogram** for an attribute,  $A$ , partitions the data distribution of  $A$  into disjoint subsets, or *buckets*. If each bucket represents only a single attribute-value/frequency pair, the buckets are called *singleton buckets*. Often, buckets instead represent continuous ranges for the given attribute.

**Example 2.5 Histograms.** The following data are a list of prices of commonly sold items at *AllElectronics* (rounded to the nearest dollar). The numbers have been sorted: 1, 1, 5, 5, 5, 5, 5, 8, 8, 10, 10, 10, 10, 12, 14, 14,



14, 15, 15, 15, 15, 15, 18, 18, 18, 18, 18, 18, 18, 18, 20, 20, 20, 20, 20, 20, 20, 21, 21, 21, 21, 25, 25, 25, 25, 25, 28, 28, 30, 30, 30.

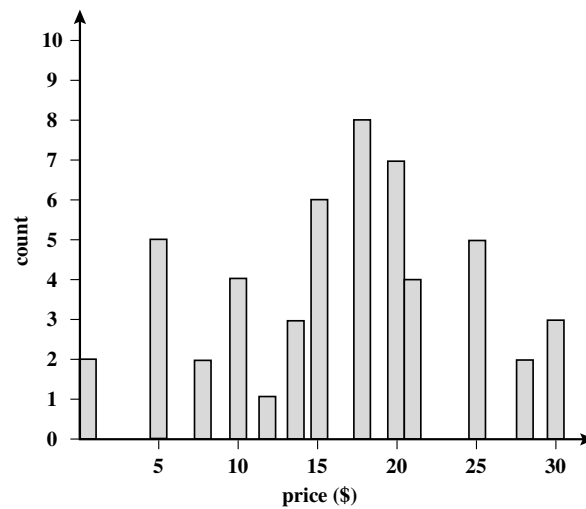


Figure 2.18: A histogram for *price* using singleton buckets—each bucket represents one price-value/frequency pair.

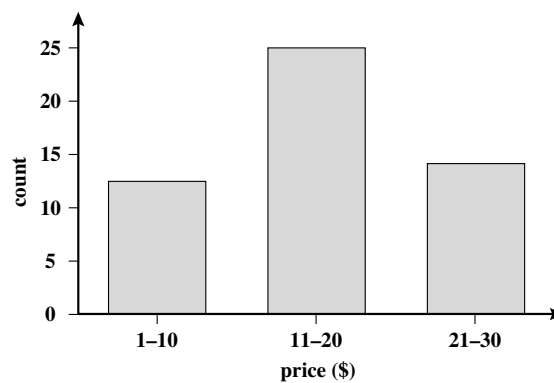


Figure 2.19: An equal-width histogram for *price*, where values are aggregated so that each bucket has a uniform width of \$10.

Figure 2.18 shows a histogram for the data using singleton buckets. To further reduce the data, it is common to have each bucket denote a continuous range of values for the given attribute. In Figure 2.19, each bucket represents a different \$10 range for *price*. ■

“How are the buckets determined and the attribute values partitioned?” There are several partitioning rules, including the following:

- **Equal-width:** In an equal-width histogram, the width of each bucket range is uniform (such as the width of \$10 for the buckets in Figure 2.19).
- **Equal-frequency** (or equidepth): In an equal-frequency histogram, the buckets are created so that, roughly, the frequency of each bucket is constant (that is, each bucket contains roughly the same number of contiguous data samples).

- **V-Optimal:** If we consider all of the possible histograms for a given number of buckets, the V-Optimal histogram is the one with the least variance. Histogram variance is a weighted sum of the original values that each bucket represents, where bucket weight is equal to the number of values in the bucket.
- **MaxDiff:** In a MaxDiff histogram, we consider the difference between each pair of adjacent values. A bucket boundary is established between each pair for pairs having the  $\beta - 1$  largest differences, where  $\beta$  is the user-specified number of buckets.

V-Optimal and MaxDiff histograms tend to be the most accurate and practical. Histograms are highly effective at approximating both sparse and dense data, as well as highly skewed and uniform data. The histograms described above for single attributes can be extended for multiple attributes. *Multidimensional histograms* can capture dependencies between attributes. Such histograms have been found effective in approximating data with up to five attributes. More studies are needed regarding the effectiveness of multidimensional histograms for very high dimensions. Singleton buckets are useful for storing outliers with high frequency.

## Clustering

Clustering techniques consider data tuples as objects. They partition the objects into groups or *clusters*, so that objects within a cluster are “similar” to one another and “dissimilar” to objects in other clusters. Similarity is commonly defined in terms of how “close” the objects are in space, based on a distance function. The “quality” of a cluster may be represented by its *diameter*, the maximum distance between any two objects in the cluster. *Centroid distance* is an alternative measure of cluster quality and is defined as the average distance of each cluster object from the cluster centroid (denoting the “average object,” or average point in space for the cluster). Figure 2.12 of Section 2.3.2 shows a 2-D plot of customer data with respect to customer locations in a city, where the centroid of each cluster is shown with a “+”. Three data clusters are visible.

In data reduction, the cluster representations of the data are used to replace the actual data. The effectiveness of this technique depends on the nature of the data. It is much more effective for data that can be organized into distinct clusters than for smeared data.

In database systems, **multidimensional index trees** are primarily used for providing fast data access. They can also be used for hierarchical data reduction, providing a multiresolution clustering of the data. This can be used to provide approximate answers to queries. An index tree recursively partitions the multidimensional space for a given set of data objects, with the root node representing the entire space. Such trees are typically balanced, consisting of internal and leaf nodes. Each parent node contains keys and pointers to child nodes that, collectively, represent the space represented by the parent node. Each leaf node contains pointers to the data tuples they represent (or to the actual tuples).

An index tree can therefore store aggregate and detail data at varying levels of resolution or abstraction. It provides a hierarchy of clusterings of the data set, where each cluster has a label that holds for the data contained in the cluster. If we consider each child of a parent node as a bucket, then an index tree can be considered as a *hierarchical histogram*. For example, consider the root of a B+-tree as shown in Figure 2.20, with pointers to the data keys 986, 3396, 5411, 8392, and 9544. Suppose that the tree contains 10,000 tuples with keys ranging from 1 to 9999. The data in the tree can be approximated by an equal-frequency histogram of six buckets for the key ranges 1 to 985, 986 to 3395, 3396 to 5410, 5411 to 8391, 8392 to 9543, and 9544 to 9999. Each bucket contains roughly 10,000/6 items. Similarly, each bucket is subdivided into smaller buckets, allowing for aggregate data at a finer-detailed level. The use of multidimensional index trees as a form of data reduction relies on an ordering of the attribute values in each dimension. Two-dimensional or multidimensional index trees include R-trees, quad-trees, and their variations. They are well suited for handling both sparse and skewed data.

There are many measures for defining clusters and cluster quality. Clustering methods are further described in Chapter 7.

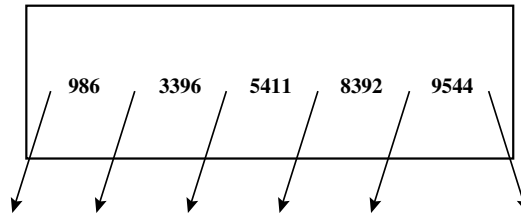


Figure 2.20: The root of a B+-tree for a given set of data.

### Sampling

Sampling can be used as a data reduction technique since it allows a large data set to be represented by a much smaller random sample (or subset) of the data. Suppose that a large data set,  $D$ , contains  $N$  tuples. Let's have a look at the most common ways that we could sample  $D$  for data reduction, as illustrated in Figure 2.21.

- **Simple random sample without replacement (SRSWOR) of size  $s$ :** This is created by drawing  $s$  of the  $N$  tuples from  $D$  ( $s < N$ ), where the probability of drawing any tuple in  $D$  is  $1/N$ , that is, all tuples are equally likely to be sampled.
- **Simple random sample with replacement (SRSWR) of size  $s$ :** This is similar to SRSWOR, except that each time a tuple is drawn from  $D$ , it is recorded and then *replaced*. That is, after a tuple is drawn, it is placed back in  $D$  so that it may be drawn again.
- **Cluster sample:** If the tuples in  $D$  are grouped into  $M$  mutually disjoint “clusters,” then an SRS of  $s$  clusters can be obtained, where  $s < M$ . For example, tuples in a database are usually retrieved a page at a time, so that each page can be considered a cluster. A reduced data representation can be obtained by applying, say, SRSWOR to the pages, resulting in a cluster sample of the tuples. Other clustering criteria conveying rich semantics can also be explored. For example, in a spatial database, we may choose to define clusters geographically based on how closely different areas are located.
- **Stratified sample:** If  $D$  is divided into mutually disjoint parts called *strata*, a stratified sample of  $D$  is generated by obtaining an SRS at each stratum. This helps to ensure a representative sample, especially when the data are skewed. For example, a stratified sample may be obtained from customer data, where a stratum is created for each customer age group. In this way, the age group having the smallest number of customers will be sure to be represented.

An advantage of sampling for data reduction is that the cost of obtaining a sample is *proportional to the size of the sample*,  $s$ , as opposed to  $N$ , the data set size. Hence, sampling complexity is potentially *sublinear* to the size of the data. Other data reduction techniques can require at least one complete pass through  $D$ . For a fixed sample size, sampling complexity increases only linearly as the number of data dimensions,  $n$ , increases, while techniques using histograms, for example, increase exponentially in  $n$ .

When applied to data reduction, sampling is most commonly used to estimate the answer to an aggregate query. It is possible (using the central limit theorem) to determine a sufficient sample size for estimating a given function within a specified degree of error. This sample size,  $s$ , may be extremely small in comparison to  $N$ . Sampling is a natural choice for the progressive refinement of a reduced data set. Such a set can be further refined by simply increasing the sample size.

## 2.6 Data Discretization and Concept Hierarchy Generation

**Data discretization techniques** can be used to reduce the number of values for a given continuous attribute by dividing the range of the attribute into intervals. Interval labels can then be used to replace actual data values.

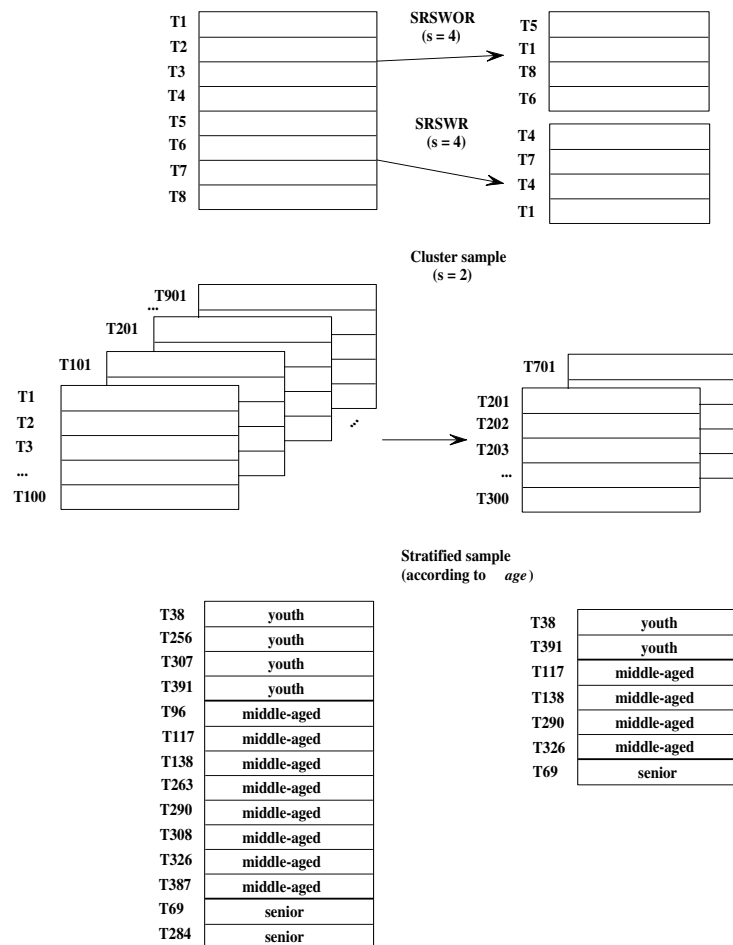


Figure 2.21: Sampling can be used for data reduction.

Replacing numerous values of a continuous attribute by a small number of interval labels thereby reduces and simplifies the original data. This leads to a concise, easy-to-use, knowledge-level representation of mining results.

Discretization techniques can be categorized based on how the discretization is performed, such as whether it uses class information, or which direction it proceeds (i.e., top-down vs. bottom-up). If the discretization process makes use of class information, then we say it is *supervised discretization*. Otherwise, it is *unsupervised*. If the process starts by first finding one or a few points (called *split points* or *cut points*) to split the entire attribute range, and then repeats this recursively on the resulting intervals, it is called *top-down discretization* or *splitting*. This contrasts with *bottom-up discretization* or *merging*, which starts by considering all of the continuous values as potential split-points, removes some by merging neighborhood values to form intervals, and then recursively applies this process to the resulting intervals. Discretization can be performed recursively on an attribute to provide a hierarchical or multiresolution partitioning of the attribute values, known as a concept hierarchy. Concept hierarchies are useful for mining at multiple levels of abstraction.

A concept hierarchy for a given numerical attribute defines a discretization of the attribute. Concept hierarchies can be used to reduce the data by collecting and replacing low-level concepts (such as numerical values for the attribute *age*) by higher-level concepts (such as *youth*, *middle-aged*, or *senior*). Although detail is lost by such data generalization, the generalized data may be more meaningful and easier to interpret. This contributes to a consistent representation of data mining results among multiple mining tasks, which is a common requirement. In addition, mining on a reduced data set requires fewer input/output operations and is more efficient than mining

on a larger, ungeneralized data set. Because of these benefits, discretization techniques and concept hierarchies are typically applied prior to data mining as a preprocessing step, rather than during mining. An example of a concept hierarchy for the attribute *price* is given in Figure 2.22. More than one concept hierarchy can be defined for the same attribute in order to accommodate the needs of various users.

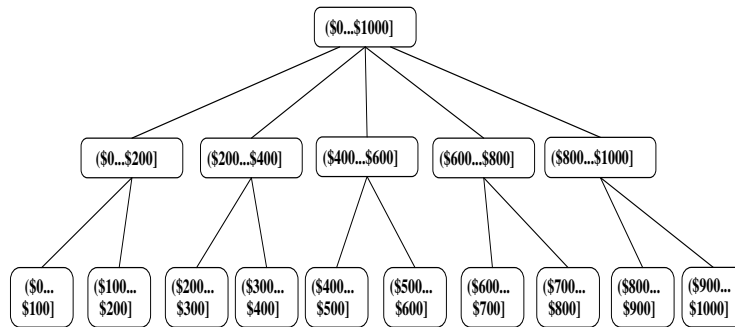


Figure 2.22: A concept hierarchy for the attribute *price*, where an interval  $(\$X \dots \$Y]$  denotes the range from  $\$X$  (exclusive) to  $\$Y$  (inclusive).

Manual definition of concept hierarchies can be a tedious and time-consuming task for a user or a domain expert. Fortunately, several discretization methods can be used to automatically generate or dynamically refine concept hierarchies for numerical attributes. Furthermore, many hierarchies for categorical attributes are implicit within the database schema and can be automatically defined at the schema definition level.

Let's look at the generation of concept hierarchies for numerical and categorical data.

### 2.6.1 Discretization and Concept Hierarchy Generation for Numerical Data

It is difficult and laborious to specify concept hierarchies for numerical attributes due to the wide diversity of possible data ranges and the frequent updates of data values. Such manual specification can also be quite arbitrary.

Concept hierarchies for numerical attributes can be constructed automatically based on data discretization. We examine the following methods: *binning*, *histogram analysis*, *entropy-based discretization*,  $\chi^2$ -*merging*, *cluster analysis*, and *discretization by intuitive partitioning*. In general, each method assumes that the values to be discretized are sorted in ascending order.

#### Binning

Binning is a top-down splitting technique based on a specified number of bins. Section 2.3.2 discussed binning methods for data smoothing. These methods are also used as discretization methods for numerosity reduction and concept hierarchy generation. For example, attribute values can be discretized by applying equal-width or equal-frequency binning, and then replacing each bin value by the bin mean or median, as in *smoothing by bin means* or *smoothing by bin medians*, respectively. These techniques can be applied recursively to the resulting partitions in order to generate concept hierarchies. Binning does not make use of class information and is therefore an unsupervised discretization technique. It is sensitive to the user-specified number of bins, as well as the presence of outliers.

#### Histogram Analysis

Like binning, histograms analysis is an unsupervised discretization technique since it does not make use of class information. Histograms partition the values for an attribute,  $A$ , into disjoint ranges called *buckets*. Histograms

were introduced in Section 2.2.3. Partitioning rules for defining histograms were described in Section 2.5.4. In an *equal-width* histogram, for example, the values are partitioned into equal-sized partitions or ranges (such as in Figure 2.19 for *price*, where each bucket has a width of \$10). With an *equal-frequency* histogram, the values are partitioned so that, ideally, each partition contains the same number of data tuples. The histogram analysis algorithm can be applied recursively to each partition in order to automatically generate a multilevel concept hierarchy, with the procedure terminating once a prespecified number of concept levels has been reached. A *minimum interval size* can also be used per level to control the recursive procedure. This specifies the minimum width of a partition, or the minimum number of values for each partition at each level. Histograms can also be partitioned based on cluster analysis of the data distribution, as described below.

### Entropy-Based Discretization

*Entropy* is one of the most commonly used discretization measures. It was first introduced by Claude Shannon in pioneering work on information theory and the concept of information gain. Entropy-based discretization is a supervised, top-down splitting technique. It explores class distribution information in its calculation and determination of split points (data values for partitioning an attribute range). To discretize a numerical attribute,  $A$ , the method selects the value of  $A$  that has the minimum entropy as a split point, and recursively partitions the resulting intervals to arrive at a hierarchical discretization. Such discretization forms a concept hierarchy for  $A$ .

Let  $D$  consist of data tuples defined by a set of attributes and a class-label attribute. The class-label attribute provides the class information per tuple. The basic method for entropy-based discretization of an attribute  $A$  within the set is as follows:

1. Each value of  $A$  can be considered as a potential interval boundary or split point (denoted *split\_point*) to partition the range of  $A$ . That is, a split point for  $A$  can partition the tuples in  $D$  into two subsets satisfying the conditions  $A \leq \text{split\_point}$  and  $A > \text{split\_point}$ , respectively, thereby creating a binary discretization.
2. Entropy-based discretization, as mentioned above, uses information regarding the class label of tuples. To explain the intuition behind entropy-based discretization, we must take a glimpse at classification. Suppose we want to classify the tuples in  $D$  by partitioning on attribute  $A$  and some split point. Ideally, we would like this partitioning to result in an exact classification of the tuples. For example, if we had two classes, we would hope that all of the tuples of, say, class  $C_1$  will fall into one partition, and all of the tuples of class  $C_2$  will fall into the other partition. However, this is unlikely. For example, the first partition may contain many tuples of  $C_1$ , but also some of  $C_2$ . How much more information would we still need for a perfect classification, after this partitioning? This amount is called the *expected information requirement* for classifying a tuple in  $D$  based on partitioning by  $A$ . It is given by

$$\text{Info}_A(D) = \frac{|D_1|}{|D|} \text{Entropy}(D_1) + \frac{|D_2|}{|D|} \text{Entropy}(D_2), \quad (2.15)$$

where  $D_1$  and  $D_2$  correspond to the tuples in  $D$  satisfying the conditions  $A \leq \text{split\_point}$  and  $A > \text{split\_point}$ , respectively;  $|D|$  is the number of tuples in  $D$ , and so on. The entropy function, *Entropy*, for a given set is calculated based on the class distribution of the tuples in the set. For example, given  $m$  classes,  $C_1, C_2, \dots, C_m$ , the entropy of  $D_1$  is

$$\text{Entropy}(D_1) = - \sum_{i=1}^m p_i \log_2(p_i), \quad (2.16)$$

where  $p_i$  is the probability of class  $C_i$  in  $D_1$ , determined by dividing the number of tuples of class  $C_i$  in  $D_1$  by  $|D_1|$ , the total number of tuples in  $D_1$ . Therefore, when selecting a split point for attribute  $A$ , we want to pick the attribute value that gives the minimum expected information requirement (i.e.,  $\min(\text{Info}_A(D))$ ). This would result in the minimum amount of expected information (still) required to perfectly classify the tuples after partitioning by  $A \leq \text{split\_point}$  and  $A > \text{split\_point}$ . This is equivalent to the attribute-value pair with the maximum information gain (the further details of which are given in Chapter 6 on classification.) Note that the value of  $\text{Entropy}(D_2)$  can be computed similarly as in Equation (2.16).

“But our task is discretization, not classification!”, you may exclaim. This is true. We use the split point to partition the range of  $A$  into two intervals, corresponding to  $A \leq \textit{split\_point}$  and  $A > \textit{split\_point}$ .

3. The process of determining a split point is recursively applied to each partition obtained, until some stopping criterion is met, such as when the minimum information requirement on all candidate split points is less than a small threshold,  $\epsilon$ , or when the number of intervals is greater than a threshold,  $\textit{max\_interval}$ .

Entropy-based discretization can reduce data size. Unlike the other methods mentioned here so far, entropy-based discretization uses class information. This makes it more likely that the interval boundaries (split points) are defined to occur in places that may help improve classification accuracy. The entropy and information gain measures described here are also used for decision tree induction. These measures are revisited in greater detail in Section 6.3.2.

### Interval Merging by $\chi^2$ Analysis

*ChiMerge* is a  $\chi^2$ -based discretization method. The discretization methods that we have studied up to this point have all employed a top-down, splitting strategy. This contrasts with *ChiMerge*, which employs a bottom-up approach by finding the best neighboring intervals and then merging these to form larger intervals, recursively. The method is supervised in that it makes use of class information. The basic notion is that for accurate discretization, the relative class frequencies should be fairly consistent within an interval. Therefore, if two adjacent intervals have a very similar distribution of classes, then the intervals can be merged. Otherwise, they should remain separate.

*ChiMerge* proceeds as follows. Initially, each distinct value of a numerical attribute  $A$  is considered to be one interval.  $\chi^2$  tests are performed for every pair of adjacent intervals. Adjacent intervals with the least  $\chi^2$  values are merged together since low  $\chi^2$  values for a pair indicate similar class distributions. This merging process proceeds recursively until a predefined stopping criterion is met.

The  $\chi^2$  statistic was introduced in Section 2.4.1 on data integration, where we explained its use to detect a correlation relationship between two categorical attributes (Equation (2.9)). Since *ChiMerge* treats intervals as discrete categories, Equation (2.9) can be applied. The  $\chi^2$  statistic tests the hypothesis that two adjacent intervals for a given attribute are independent of the class. Following the method in Example 2.1, we can construct a contingency table for our data. The contingency table has two columns (representing the two adjacent intervals) and  $m$  rows, where  $m$  is the number of distinct classes. Applying Equation (2.9) here, the cell value  $o_{ij}$  is the count of tuples in the  $i^{\text{th}}$  interval and  $j^{\text{th}}$  class. Similarly, the expected frequency of  $o_{ij}$  is  $e_{ij} = (\text{number of tuples in interval } i) \times (\text{number of tuples in class } j) / N$ , where  $N$  is the total number of data tuples. Low  $\chi^2$  values for an interval pair indicate that the intervals are independent of the class and can, therefore, be merged.

The stopping criterion is typically determined by three conditions. First, merging stops when  $\chi^2$  values of all pairs of adjacent intervals are below some threshold, which is determined by a specified significance level. A high value of significance level for the  $\chi^2$  test may cause over discretization, while a lower value may lead to under discretization. Typically, the significance level is set between 0.90 to 0.99. Second, the number of intervals cannot be over a prespecified  $\textit{max\_interval}$ , such as 10 to 15. Finally, recall that the premise behind *ChiMerge* is that the relative class frequencies should be fairly consistent within an interval. In practice, some inconsistency is allowed, although this should be no more than a prespecified threshold, such as 3%. This last condition can be used to remove irrelevant attributes from the data set.

### Cluster Analysis

Cluster analysis is a popular data discretization method. A clustering algorithm can be applied to discretize a numerical attribute,  $A$ , by partitioning the values of  $A$  into clusters or groups. Clustering takes the distribution of  $A$  into consideration, as well as the closeness of data points, and therefore is able to produce high quality discretization results. Clustering can be used to generate a concept hierarchy for  $A$  by following either a top-down splitting strategy or a bottom-up merging strategy, where each cluster forms a node of the concept hierarchy. In the former, each initial cluster or partition may be further decomposed into several subclusters, forming a lower

level of the hierarchy. In the latter, clusters are formed by repeatedly grouping neighboring clusters in order to form higher level concepts. Clustering methods for data mining are studied in Chapter 7.

### Discretization by Intuitive Partitioning

Although the above discretization methods are useful in the generation of numerical hierarchies, many users would like to see numerical ranges partitioned into relatively uniform, easy-to-read intervals that appear intuitive or “natural.” For example, annual salaries broken into ranges like  $(\$50,000, \$60,000]$  are often more desirable than ranges like  $(\$51,263.98, \$60,872.34]$ , obtained by, say, some sophisticated clustering analysis.

The **3-4-5 rule** can be used to segment numerical data into relatively uniform, natural-seeming intervals. In general, the rule partitions a given range of data into 3, 4, or 5 relatively equal-width intervals, recursively and level by level, based on the value range at the most significant digit. We will illustrate the use of the rule with an example further below. The rule is as follows:

- If an interval covers 3, 6, 7, or 9 distinct values at the most significant digit, then partition the range into 3 intervals (3 equal-width intervals for 3, 6, and 9; and 3 intervals in the grouping of 2-3-2 for 7).
- If it covers 2, 4, or 8 distinct values at the most significant digit, then partition the range into 4 equal-width intervals.
- If it covers 1, 5, or 10 distinct values at the most significant digit, then partition the range into 5 equal-width intervals.

The rule can be recursively applied to each interval, creating a concept hierarchy for the given numerical attribute. Real world data often contain extremely large positive and/or negative outlier values, which could distort any top-down discretization method based on minimum and maximum data values. For example, the assets of a few people could be several orders of magnitude higher than those of others in the same data set. Discretization based on the maximal asset values may lead to a highly biased hierarchy. Thus the top-level discretization can be performed based on the range of data values representing the majority (e.g., 5th percentile to 95th percentile) of the given data. The extremely high or low values beyond the top-level discretization will form distinct interval(s) that can be handled separately, but in a similar manner.

The following example illustrates the use of the 3-4-5 rule for the automatic construction of a numerical hierarchy.

**Example 2.6 Numeric concept hierarchy generation by intuitive partitioning.** Suppose that profits at different branches of *AllElectronics* for the year 2004 cover a wide range, from  $-\$351,976.00$  to  $\$4,700,896.50$ . A user desires the automatic generation of a concept hierarchy for *profit*. For improved readability, we use the notation  $(l..r]$  to represent the interval  $[l, r]$ . For example,  $(-\$1,000,000..\$0]$  denotes the range from  $-\$1,000,000$  (exclusive) to  $\$0$  (inclusive).

Suppose that the data within the 5th percentile and 95th percentile are between  $-\$159,876$  and  $\$1,838,761$ . The results of applying the 3-4-5 rule are shown in Figure 2.23.

1. Based on the above information, the minimum and maximum values are  $MIN = -\$351,976.00$ , and  $MAX = \$4,700,896.50$ . The low (5th percentile) and high (95th percentile) values to be considered for the top or first level of discretization are  $LOW = -\$159,876$ , and  $HIGH = \$1,838,761$ .
2. Given  $LOW$  and  $HIGH$ , the most significant digit (*msd*) is at the million dollar digit position (i.e.,  $msd = 1,000,000$ ). Rounding  $LOW$  down to the million dollar digit, we get  $LOW' = -\$1,000,000$ ; rounding  $HIGH$  up to the million dollar digit, we get  $HIGH' = +\$2,000,000$ .
3. Since this interval ranges over three distinct values at the most significant digit, that is,  $(2,000,000 - (-1,000,000))/1,000,000 = 3$ , the segment is partitioned into three equal-width subsegments according to the 3-4-5 rule:  $(-\$1,000,000 \dots \$0]$ ,  $(\$0 \dots \$1,000,000]$ , and  $(\$1,000,000 \dots \$2,000,000]$ . This represents the top tier of the hierarchy.



To Editor: Fig. 3.16 p. 137 (of Edition 1) should be printed here. However, since including this in .dvi file will not generate correct .ps file, we do not print the graph here.

---

Figure 2.23: Automatic generation of a concept hierarchy for *profit* based on the 3-4-5 rule.  
 [TO EDITOR The graph enclosed here is old and contains errors! Fig. 3.16 (pg. 137) of the latest printing of the book is the correct one to use.]

---

4. We now examine the MIN and MAX values to see how they “fit” into the first-level partitions. Since the first interval  $(-\$1,000,000 \dots \$0]$  covers the *MIN* value, that is,  $LOW' < MIN$ , we can adjust the left boundary of this interval to make the interval smaller. The most significant digit of *MIN* is the hundred thousand digit position. Rounding *MIN* down to this position, we get  $MIN' = -\$400,000$ . Therefore, the first interval is redefined as  $(-\$400,000 \dots \$0]$ .

Since the last interval,  $(\$1,000,000 \dots \$2,000,000]$ , does not cover the *MAX* value, that is,  $MAX > HIGH'$ , we need to create a new interval to cover it. Rounding up *MAX* at its most significant digit position, the new interval is  $(\$2,000,000 \dots \$5,000,000]$ . Hence, the topmost level of the hierarchy contains four partitions,  $(-\$400,000 \dots \$0]$ ,  $(\$0 \dots \$1,000,000]$ ,  $(\$1,000,000 \dots \$2,000,000]$ , and  $(\$2,000,000 \dots \$5,000,000]$ .

5. Recursively, each interval can be further partitioned according to the 3-4-5 rule to form the next lower level of the hierarchy:
  - The first interval,  $(-\$400,000 \dots \$0]$ , is partitioned into 4 subintervals:  $(-\$400,000 \dots -\$300,000]$ ,  $(-\$300,000 \dots -\$200,000]$ ,  $(-\$200,000 \dots -\$100,000]$ , and  $(-\$100,000 \dots \$0]$ .
  - The second interval,  $(\$0 \dots \$1,000,000]$ , is partitioned into 5 subintervals:  $(\$0 \dots \$200,000]$ ,  $(\$200,000 \dots \$400,000]$ ,  $(\$400,000 \dots \$600,000]$ ,  $(\$600,000 \dots \$800,000]$ , and  $(\$800,000 \dots \$1,000,000]$ .
  - The third interval,  $(\$1,000,000 \dots \$2,000,000]$ , is partitioned into 5 subintervals:  $(\$1,000,000 \dots \$1,200,000]$ ,  $(\$1,200,000 \dots \$1,400,000]$ ,  $(\$1,400,000 \dots \$1,600,000]$ ,  $(\$1,600,000 \dots \$1,800,000]$ , and  $(\$1,800,000 \dots \$2,000,000]$ .
  - The last interval,  $(\$2,000,000 \dots \$5,000,000]$ , is partitioned into 3 subintervals:  $(\$2,000,000 \dots \$3,000,000]$ ,  $(\$3,000,000 \dots \$4,000,000]$ , and  $(\$4,000,000 \dots \$5,000,000]$ .

Similarly, the 3-4-5 rule can be carried on iteratively at deeper levels, as necessary. ■

### 2.6.2 Concept Hierarchy Generation for Categorical Data

Categorical data are discrete data. Categorical attributes have a finite (but possibly large) number of distinct values, with no ordering among the values. Examples include *geographic location*, *job category*, and *item type*. There are several methods for the generation of concept hierarchies for categorical data.

#### Specification of a partial ordering of attributes explicitly at the schema level by users or experts:

Concept hierarchies for categorical attributes or dimensions typically involve a group of attributes. A user

or expert can easily define a concept hierarchy by specifying a partial or total ordering of the attributes at the schema level. For example, a relational database or a dimension *location* of a data warehouse may contain the following group of attributes: *street*, *city*, *province\_or\_state*, and *country*. A hierarchy can be defined by specifying the total ordering among these attributes at the schema level, such as  $street < city < province\_or\_state < country$ .

**Specification of a portion of a hierarchy by explicit data grouping:** This is essentially the manual definition of a portion of a concept hierarchy. In a large database, it is unrealistic to define an entire concept hierarchy by explicit value enumeration. On the contrary, we can easily specify explicit groupings for a small portion of intermediate-level data. For example, after specifying that *province* and *country* form a hierarchy at the schema level, a user could define some intermediate levels manually, such as “{*Alberta*, *Saskatchewan*, *Manitoba*}  $\subset$  *prairies\_Canada*” and “{*British Columbia*, *prairies\_Canada*}  $\subset$  *Western\_Canada*”.

**Specification of a set of attributes, but not of their partial ordering:** A user may specify a set of attributes forming a concept hierarchy, but omit to explicitly state their partial ordering. The system can then try to automatically generate the attribute ordering so as to construct a meaningful concept hierarchy.

“Without knowledge of data semantics, how can a hierarchical ordering for an arbitrary set of categorical attributes be found?” Consider the following observation that since higher-level concepts generally cover several subordinate lower-level concepts, an attribute defining a high concept level (e.g., *country*) will usually contain a smaller number of distinct values than an attribute defining a lower concept level (e.g., *street*). Based on this observation, a concept hierarchy can be automatically generated based on the number of distinct values per attribute in the given attribute set. The attribute with the most distinct values is placed at the lowest level of the hierarchy. The lower the number of distinct values an attribute has, the higher it is in the generated concept hierarchy. This heuristic rule works well in many cases. Some local-level swapping or adjustments may be applied by users or experts, when necessary, after examination of the generated hierarchy.

Let’s examine an example of this method.

**Example 2.7 Concept hierarchy generation based on the number of distinct values per attribute.** Suppose a user selects a set of attributes, *street*, *country*, *province\_or\_state*, and *city*, for a dimension *location* from the *AllElectronics* database, but does not specify the hierarchical ordering among the attributes.

The concept hierarchy for *location* can be generated automatically, as illustrated in Figure 2.24. First, sort the attributes in ascending order based on the number of distinct values in each attribute. This results in the following (where the number of distinct values per attribute is shown in parentheses): *country* (15), *province\_or\_state* (365), *city* (3567), and *street* (674,339). Second, generate the hierarchy from the top down according to the sorted order, with the first attribute at the top level and the last attribute at the bottom level. Finally, the user can examine the generated hierarchy, and when necessary, modify it to reflect desired semantic relationships among the attributes. In this example, it is obvious that there is no need to modify the generated hierarchy. ■

Note that this heuristic rule is not foolproof. For example, a time dimension in a database may contain 20 distinct years, 12 distinct months, and 7 distinct days of the week. However, this does not suggest that the time hierarchy should be “*year* < *month* < *days\_of\_the\_week*”, with *days\_of\_the\_week* at the top of the hierarchy.

**Specification of only a partial set of attributes:** Sometimes a user can be sloppy when defining a hierarchy, or have only a vague idea about what should be included in a hierarchy. Consequently, the user may have included only a small subset of the relevant attributes in the hierarchy specification. For example, instead of including all the hierarchically relevant attributes for *location*, the user may have specified only *street* and *city*. To handle such partially specified hierarchies, it is important to embed data semantics in the database schema so that attributes with tight semantic connections can be pinned together. In this way, the specification of one attribute may trigger a whole group of semantically tightly linked attributes to be

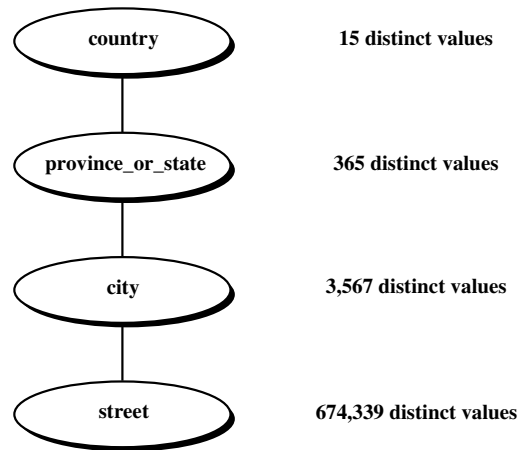


Figure 2.24: Automatic generation of a schema concept hierarchy based on the number of distinct attribute values.

“dragged in” to form a complete hierarchy. Users, however, should have the option to override this feature, as necessary.

**Example 2.8 Concept hierarchy generation using prespecified semantic connections.** Suppose that a data mining expert (serving as an administrator) has pinned together the five attributes, *number*, *street*, *city*, *province\_or\_state*, and *country*, because they are closely linked semantically, regarding the notion of *location*. If a user were to specify only the attribute *city* for a hierarchy defining *location*, the system can automatically drag in all of the above five semantically related attributes to form a hierarchy. The user may choose to drop any of these attributes, such as *number* and *street*, from the hierarchy, keeping *city* as the lowest conceptual level in the hierarchy. ■

## 2.7 Summary

- **Data preprocessing** is an important issue for both data warehousing and data mining, as real-world data tend to be incomplete, noisy, and inconsistent. Data preprocessing includes data cleaning, data integration, data transformation, and data reduction.
- **Descriptive data summarization** provides the analytical foundation for data preprocessing. The basic statistical measures for data summarization include *mean*, *weighted mean*, *median*, and *mode* for measuring the central tendency of data, and *range*, *quartiles*, *interquartile range*, *variance* and *standard deviation* for measuring the dispersion of data. Graphical representations, such as *histograms*, *boxplots*, *quantile plots*, *quantile-quantile plots*, *scatter plots*, and *scatter-plot matrices* facilitate visual inspection of the data and are thus useful for data preprocessing and mining.
- **Data cleaning** routines attempt to fill in missing values, smooth out noise while identifying outliers, and correct inconsistencies in the data. Data cleaning is usually performed as an iterative two-step process consisting of discrepancy detection and data transformation.
- **Data integration** combines data from multiple sources to form a coherent data store. Metadata, correlation analysis, data conflict detection, and the resolution of semantic heterogeneity contribute towards smooth data integration.
- **Data transformation** routines convert the data into appropriate forms for mining. For example, attribute data may be **normalized** so as to fall between a small range, such as 0.0 to 1.0.

- **Data reduction** techniques such as data cube aggregation, attribute subset selection, dimensionality reduction, numerosity reduction, and discretization can be used to obtain a reduced representation of the data, while minimizing the loss of information content.
- **Data discretization and automatic generation of concept hierarchies** for numerical data can involve techniques such as binning, histogram analysis, entropy-based discretization,  $\chi^2$  analysis, cluster analysis, and discretization by intuitive partitioning. For categorical data, concept hierarchies may be generated based on the number of distinct values of the attributes defining the hierarchy.
- Although numerous methods of data preprocessing have been developed, data preprocessing remains an active area of research, due to the huge amount of inconsistent or dirty data and the complexity of the problem.

## 2.8 Exercises

1. *Data quality* can be assessed in terms of accuracy, completeness, and consistency. Propose two other dimensions of data quality.
2. Suppose that the values for a given set of data are grouped into intervals. The intervals and corresponding frequencies are as follows.

<i>age</i>	<i>frequency</i>
1-5	200
5-15	450
15-20	300
20-50	1500
50-80	700
80-110	44

Compute an *approximate median* value for the data.

3. Give three additional commonly used statistical measures (i.e., not illustrated in this chapter) for the characterization of *data dispersion*, and discuss how they can be computed efficiently in large databases.
4. Suppose that the data for analysis includes the attribute *age*. The *age* values for the data tuples are (in increasing order) 13, 15, 16, 16, 19, 20, 20, 21, 22, 22, 25, 25, 25, 25, 30, 33, 33, 35, 35, 35, 35, 36, 40, 45, 46, 52, 70.
  - (a) What is the *mean* of the data? What is the *median*?
  - (b) What is the *mode* of the data? Comment on the data's modality (i.e., bimodal, trimodal, etc.).
  - (c) What is the *midrange* of the data?
  - (d) Can you find (roughly) the first quartile ( $Q_1$ ) and the third quartile ( $Q_3$ ) of the data?
  - (e) Give the *five-number summary* of the data.
  - (f) Show a *boxplot* of the data.
  - (g) How is a *quantile-quantile plot* different from a *quantile plot*?
5. In many applications, new data sets are incrementally added to the existing large data sets. Thus an important consideration for computing descriptive data summary is whether a measure can be computed efficiently in incremental manner. Use *count*, *standard deviation*, and *median* as examples to show that a distributive or algebraic measure facilitates efficient incremental computation, whereas a holistic measure does not.
6. In real-world data, tuples with *missing values* for some attributes are a common occurrence. Describe various methods for handling this problem.

7. Suppose that the data for analysis include the attribute *age*. The *age* values for the data tuples are (in increasing order): 13, 15, 16, 16, 19, 20, 20, 21, 22, 22, 25, 25, 25, 25, 30, 33, 33, 35, 35, 35, 35, 36, 40, 45, 46, 52, 70.
- Use *smoothing by bin means* to smooth the above data, using a bin depth of 3. Illustrate your steps. Comment on the effect of this technique for the given data.
  - How might you determine *outliers* in the data?
  - What other methods are there for *data smoothing*?
8. Discuss issues to consider during *data integration*.
9. Suppose a hospital tested the age and body fat data for 18 randomly selected adults with the following result

<i>age</i>	23	23	27	27	39	41	47	49	50
<i>%fat</i>	9.5	26.5	7.8	17.8	31.4	25.9	27.4	27.2	31.2
<i>age</i>	52	54	54	56	57	58	58	60	61
<i>%fat</i>	34.6	42.5	28.8	33.4	30.2	34.1	32.9	41.2	35.7

- Calculate the mean, median and standard deviation of *age* and *%fat*.
  - Draw the boxplots for *age* and *%fat*.
  - Draw a *scatter plot* and a *q-q plot* based on these two variables.
  - Normalize the two variables based on *z-score normalization*.
  - Calculate the *Pearson correlation coefficient*. Are these two variables positively or negatively correlated?
10. What are the value ranges of the following *normalization methods*?
- min-max normalization
  - z-score normalization
  - normalization by decimal scaling
11. Use the two methods below to *normalize* the following group of data:
- 200, 300, 400, 600, 1000
- min-max normalization by setting  $min = 0$  and  $max = 1$
  - z-score normalization
12. Using the data for *age* given in Exercise 2.4, answer the following:
- Use min-max normalization to transform the value 35 for *age* onto the range  $[0.0, 1.0]$ .
  - Use z-score normalization to transform the value 35 for *age*, where the standard deviation of *age* is 12.94 years.
  - Use normalization by decimal scaling to transform the value 35 for *age*.
  - Comment on which method you would prefer to use for the given data, giving reasons as to why.
13. Use a flow chart to summarize the following procedures for *attribute subset selection*:
- stepwise forward selection
  - stepwise backward elimination
  - a combination of forward selection and backward elimination

14. Suppose a group of 12 *sales price* records has been sorted as follows:

5, 10, 11, 13, 15, 35, 50, 55, 72, 92, 204, 215.

Partition them into three bins by each of the following methods.

- (a) equal-frequency (equidepth) partitioning
  - (b) equal-width partitioning
  - (c) clustering
15. Using the data for *age* given in Exercise 2.4,
- (a) Plot an equal-width histogram of width 10.
  - (b) Sketch examples of each of the following sampling techniques: SRSWOR, SRSWR, cluster sampling, stratified sampling. Use samples of size 5 and the strata “youth”, “middle-aged”, and “senior”.
16. [Contributed by Chen Chen] The *median* is one of the most important holistic measures in data analysis. Propose several methods for median approximation. Analyze their respective complexity under different parameter settings and decide to what extent the real value can be approximated. Moreover, suggest a heuristic strategy to balance between accuracy and complexity and then apply it to all methods you have given.
17. [Contributed by Deng Cai] It is important to define or select similarity measures in data analysis. However, there is no commonly-accepted subjective similarity measure. Using different similarity measures may deduce different results. Nonetheless, some apparently different similarity measures may be equivalent after some transformation.

Suppose we have the following two-dimensional data set:

	$A_1$	$A_2$
$\mathbf{x}_1$	1.5	1.7
$\mathbf{x}_2$	2	1.9
$\mathbf{x}_3$	1.6	1.8
$\mathbf{x}_4$	1.2	1.5
$\mathbf{x}_5$	1.5	1.0

- (a) Consider the data as two-dimensional data points. Given a new data point,  $\mathbf{x} = (1.4, 1.6)$  as a query, rank the database points based on similarity with the query using (1) Euclidean distance, and (2) cosine similarity.
  - (b) Normalize the data set to make the norm of each data point equal to 1. Use Euclidean distance on the transformed data to rank the data points.
18. ChiMerge [Ker92] is a supervised, bottom-up (i.e., merge-based) *data discretization* method. It relies on  $\chi^2$  analysis: adjacent intervals with the least  $\chi^2$  values are merged together till the chosen stopping criterion satisfies.
- (a) Briefly describe how ChiMerge works.
  - (b) Take the IRIS data set, obtained from <http://www.ics.uci.edu/~mlearn/MLRepository.html> (UC-Irvine Machine Learning Data Repository), as a data set to be discretized. Perform data discretization for each of the four numerical attributes using the ChiMerge method. (Let the stopping criteria be: *max-interval* = 6). You need to write a small program to do this to avoid clumsy numerical computation. Submit your simple analysis and your test results: split points, final intervals, and your documented source program.
19. Propose an algorithm, in pseudocode or in your favorite programming language, for the following:

- (a) The automatic generation of a concept hierarchy for categorical data based on the number of distinct values of attributes in the given schema
  - (b) The automatic generation of a concept hierarchy for numerical data based on the *equal-width* partitioning rule
  - (c) The automatic generation of a concept hierarchy for numerical data based on the *equal-frequency* partitioning rule
20. Robust data loading poses a challenge in database systems because the input data are often dirty. In many cases, an input record may miss multiple values, some records could be *contaminated*, with some data values out of range or of a different data type than expected. Work out an automated *data cleaning and loading* algorithm so that the erroneous data will be marked, and contaminated data will not be mistakenly inserted into the database during data loading.

## 2.9 Bibliographic Notes

Data preprocessing is discussed in a number of textbooks, including English [Eng99], Pyle [Pyl99], Loshin [Los01], Redman [Red01], and Dasu and Johnson [DJ03]. More specific references to individual preprocessing techniques are given below.

Methods for descriptive data summarization have been studied in the statistics literature long before the onset of computers. Good summaries of statistical descriptive data mining methods include Freedman, Pisani and Purves [FPP97], and Devore [Dev95]. For statistics-based visualization of data using boxplots, quantile plots, quantile-quantile plots, scatter plots, and loess curves, see Cleveland [Cle93].

For discussion regarding data quality, see Redman [Red92], Wang, Storey, and Firth [WSF95], Wand and Wang [WW96], Ballou and Tayi [BT99], and Olson [Ols03]. Potter's Wheel (*control.cx.berkeley.edu/abc*), the interactive data cleaning tool described in Section 2.3.3, is presented in Raman and Hellerstein [RH01]. An example of the development of declarative languages for the specification of data transformation operators is given in Galhardas et al. [GFS<sup>+</sup>01]. The handling of missing attribute values is discussed in Friedman [Fri77], Breiman, Friedman, Olshen, and Stone [BFOS84], and Quinlan [Qui89]. A method for the detection of outlier or "garbage" patterns in a handwritten character database is given in Guyon, Matic, and Vapnik [GMV96]. Binning and data normalization are treated in many texts, including [KLV<sup>+</sup>98], [WI98], [Pyl99]. Systems that include attribute (or feature) construction include BACON by Langley, Simon, Bradshaw, and Zytkow [LSBZ87], Stagger by Schlimmer [Sch86], FRINGE by Pagallo [Pag89], and AQ17-DCI by Bloedorn and Michalski [BM98]. Attribute construction is also described in Liu and Motoda [LM98, Le98]. Dasu, et al. built a BELLMAN system and proposed a set of interesting methods for building a data quality browser by mining database structures [DJMS02].

A good survey of data reduction techniques can be found in Barbará et al. [BDF<sup>+</sup>97]. For algorithms on data cubes and their precomputation, see [SS94, AAD<sup>+</sup>96, HRU96, RS97, ZDN97]. Attribute subset selection (or *feature subset selection*) is described in many texts, such as Neter, Kutner, Nachtsheim, and Wasserman [NKNW96], Dash and Liu [DL97], and Liu and Motoda [LM98, LM98b]. A combination forward selection and backward elimination method was proposed in Siedlecki and Sklansky [SS88]. A wrapper approach to attribute selection is described in Kohavi and John [KJ97]. Unsupervised attribute subset selection is described in Dash, Liu, and Yao [DLY97]. For a description of wavelets for [OLD: data compression][NEW: dimensionality reduction], see Press, Teukolsky, Vetterling, and Flannery [PTVF96]. A general account of wavelets can be found in Hubbard [Hub96]. For a list of wavelet software packages, see Bruce, Donoho, and Gao [BDG96]. Daubechies transforms are described in Daubechies [Dau92]. The book by Press, et al. [PTVF96] includes an introduction to singular value decomposition for principal components analysis. Routines for PCA are included in most statistical software packages, such as SAS (<http://www.sas.com/SASHome.html>).

An introduction to regression and log-linear models can be found in several textbooks, such as [Jam85, Dob90, JW92, Dev95, NKNW96]. For log-linear models (known as *multiplicative models* in the computer science literature), see Pearl [Pea88]. For a general introduction to histograms, see Barbará et al. [BDF<sup>+</sup>97] and Devore and Peck [DP97]. For extensions of single attribute histograms to multiple attributes, see Muralikrishna and DeWitt [MD88]

and Poosala and Ioannidis [PI97]. Several references to clustering algorithms are given in Chapter 7 of this book, which is devoted to the topic. A survey of multidimensional indexing structures is given in Gaede and Günther [GG98]. The use of multidimensional index trees for data aggregation is discussed in Aoki [Aok98]. Index trees include R-trees (Guttman [Gut84]), quad-trees (Finkel and Bentley [FB74]), and their variations. For discussion on sampling and data mining, see Kivinen and Mannila [KM94] and John and Langley [JL96].

There are many methods for assessing attribute relevance. Each has its own bias. The information gain measure is biased towards attributes with many values. Many alternatives have been proposed, such as gain ratio (Quinlan [Qui93]), which considers the probability of each attribute value. Other relevance measures include the gini index (Breiman, Friedman, Olshen, and Stone [BFOS84]), the  $\chi^2$  contingency table statistic, and the uncertainty coefficient (Johnson and Wichern [JW92]). For a comparison of attribute selection measures for decision tree induction, see Buntine and Niblett [BN92]. For additional methods, see Liu and Motoda [LM98]b, Dash and Liu [DL97], and Almuallim and Dietterich [AD91].

Liu et al. [LHTD02] performed a comprehensive survey of data discretization methods. Entropy-based discretization with the C4.5 algorithm is described in Quinlan [Qui93]. In Catlett [Cat91], the D-2 system binarizes a numerical feature recursively. ChiMerge by Kerber [Ker92] and Chi2 by Liu and Setiono [LS95] are methods for the automatic discretization of numerical attributes that both employ the  $\chi^2$  statistic. Fayyad and Irani [FI93] apply the minimum description length principle to determine the number of intervals for numerical discretization. Concept hierarchies and their automatic generation from categorical data are described in Han and Fu [HF94].



## Chapter 3

# Data Warehouse and OLAP Technology: An Overview

Data warehouses generalize and consolidate data in multidimensional space. The construction of data warehouses, which involves data cleaning, data transformation, and data integration, can be viewed as an important preprocessing step for data mining. Moreover, data warehouses provide *on-line analytical processing (OLAP)* tools for the interactive analysis of multidimensional data of varied granularities, which facilitates effective data generalization and data mining. Many other data mining functions, such as classification, prediction, association, and clustering, can be integrated with OLAP operations to enhance interactive mining of knowledge at multiple levels of abstraction. Hence, the data warehouse has become an increasingly important platform for data analysis and on-line analytical processing and will provide an effective platform for data mining. Therefore, data warehousing and OLAP form an essential step in the knowledge discovery process. This chapter presents an overview of data warehouse and OLAP technology. Such an overview is essential for understanding the overall data mining and knowledge discovery process.

In this chapter, we study a well-accepted definition of the data warehouse and see why more and more organizations are building data warehouses for the analysis of their data. In particular, we study the *data cube*, a multidimensional data model for data warehouses and OLAP, as well as OLAP operations, such as roll-up, drill-down, slicing and dicing. We also look at data warehouse architecture, including steps on data warehouse design and construction. An overview of data warehouse implementation is presented, which examines general strategies for efficient data cube computation, OLAP data indexing, and OLAP query processing. Finally, we look at *online-analytical mining*, a powerful paradigm that integrates data warehouse and OLAP technology with that of data mining.

### 3.1 What Is a Data Warehouse?

Data warehousing provides architectures and tools for business executives to systematically organize, understand, and use their data to make strategic decisions. Data warehouse systems are valuable tools in today's competitive, fast-evolving world. In the last several years, many firms have spent millions of dollars in building enterprise-wide data warehouses. Many people feel that with competition mounting in every industry, data warehousing is the latest must-have marketing weapon—a way to keep customers by learning more about their needs.

“Then, what exactly is a data warehouse?” Data warehouses have been defined in many ways, making it difficult to formulate a rigorous definition. Loosely speaking, a data warehouse refers to a database that is maintained separately from an organization's operational databases. Data warehouse systems allow for the integration of a variety of application systems. They support information processing by providing a solid platform of consolidated historical data for analysis.

According to William H. Inmon, a leading architect in the construction of data warehouse systems, “A data warehouse is a subject-oriented, integrated, time-variant, and nonvolatile collection of data in support of management’s decision making process” [Inm96]. This short, but comprehensive definition presents the major features of a data warehouse. The four keywords, *subject-oriented*, *integrated*, *time-variant*, and *nonvolatile*, distinguish data warehouses from other data repository systems, such as relational database systems, transaction processing systems, and file systems. Let’s take a closer look at each of these key features.

- **Subject-oriented:** A data warehouse is organized around major subjects, such as customer, supplier, product, and sales. Rather than concentrating on the day-to-day operations and transaction processing of an organization, a data warehouse focuses on the modeling and analysis of data for decision makers. Hence, data warehouses typically provide a simple and concise view around particular subject issues by excluding data that are not useful in the decision support process.
- **Integrated:** A data warehouse is usually constructed by integrating multiple heterogeneous sources, such as relational databases, flat files, and on-line transaction records. Data cleaning and data integration techniques are applied to ensure consistency in naming conventions, encoding structures, attribute measures, and so on.
- **Time-variant:** Data are stored to provide information from a historical perspective (e.g., the past 5–10 years). Every key structure in the data warehouse contains, either implicitly or explicitly, an element of time.
- **Nonvolatile:** A data warehouse is always a physically separate store of data transformed from the application data found in the operational environment. Due to this separation, a data warehouse does not require transaction processing, recovery, and concurrency control mechanisms. It usually requires only two operations in data accessing: *initial loading of data* and *access of data*.

In sum, a data warehouse is a semantically consistent data store that serves as a physical implementation of a decision support data model and stores the information on which an enterprise needs to make strategic decisions. A data warehouse is also often viewed as an architecture, constructed by integrating data from multiple heterogeneous sources to support structured and/or ad hoc queries, analytical reporting, and decision making.

Based on the above, we view *data warehousing* as the *process of constructing and using data warehouses*. The construction of a data warehouse requires data cleaning, data integration, and data consolidation. The utilization of a data warehouse often necessitates a collection of *decision support* technologies. This allows “knowledge workers” (e.g., managers, analysts, and executives) to use the warehouse to quickly and conveniently obtain an overview of the data, and to make sound decisions based on information in the warehouse. Some authors use the term “data warehousing” to refer only to the process of data warehouse *construction*, while the term “warehouse DBMS” is used to refer to the *management and utilization* of data warehouses. We will not make this distinction here.

“*How are organizations using the information from data warehouses?*” Many organizations use this information to support business decision making activities, including (1) increasing customer focus, which includes the analysis of customer buying patterns (such as buying preference, buying time, budget cycles, and appetites for spending); (2) repositioning products and managing product portfolios by comparing the performance of sales by quarter, by year, and by geographic regions, in order to fine-tune production strategies; (3) analyzing operations and looking for sources of profit; and (4) managing the customer relationships, making environmental corrections, and managing the cost of corporate assets.

Data warehousing is also very useful from the point of view of *heterogeneous database integration*. Many organizations typically collect diverse kinds of data and maintain large databases from multiple, heterogeneous, autonomous, and distributed information sources. To integrate such data, and provide easy and efficient access to it, is highly desirable, yet challenging. Much effort has been spent in the database industry and research community towards achieving this goal.

The traditional database approach to heterogeneous database integration is to build **wrappers** and **integrators** (or **mediators**), on top of multiple, heterogeneous databases. When a query is posed to a client site, a metadata dictionary is used to translate the query into queries appropriate for the individual heterogeneous sites involved. These queries are then mapped and sent to local query processors. The results returned from the different sites

are integrated into a global answer set. This **query-driven approach** requires complex information filtering and integration processes, and competes for resources with processing at local sources. It is inefficient and potentially expensive for frequent queries, especially for queries requiring aggregations.

Data warehousing provides an interesting alternative to the traditional approach of heterogeneous database integration described above. Rather than using a query-driven approach, data warehousing employs an **update-driven** approach in which information from multiple, heterogeneous sources is integrated in advance and stored in a warehouse for direct querying and analysis. Unlike on-line transaction processing databases, data warehouses do not contain the most current information. However, a data warehouse brings high performance to the integrated heterogeneous database system since data are copied, preprocessed, integrated, annotated, summarized, and restructured into one semantic data store. Furthermore, query processing in data warehouses does not interfere with the processing at local sources. Moreover, data warehouses can store and integrate historical information and support complex multidimensional queries. As a result, data warehousing has become popular in industry.

### 3.1.1 Differences between Operational Database Systems and Data Warehouses

Since most people are familiar with commercial relational database systems, it is easy to understand what a data warehouse is by comparing these two kinds of systems.

The major task of on-line operational database systems is to perform on-line transaction and query processing. These systems are called **on-line transaction processing (OLTP)** systems. They cover most of the day-to-day operations of an organization, such as purchasing, inventory, manufacturing, banking, payroll, registration, and accounting. Data warehouse systems, on the other hand, serve users or knowledge workers in the role of data analysis and decision making. Such systems can organize and present data in various formats in order to accommodate the diverse needs of the different users. These systems are known as **on-line analytical processing (OLAP)** systems.

The major distinguishing features between OLTP and OLAP are summarized as follows.

- **Users and system orientation:** An OLTP system is *customer-oriented* and is used for transaction and query processing by clerks, clients, and information technology professionals. An OLAP system is *market-oriented* and is used for data analysis by knowledge workers, including managers, executives, and analysts.
- **Data contents:** An OLTP system manages current data that, typically, are too detailed to be easily used for decision making. An OLAP system manages large amounts of historical data, provides facilities for summarization and aggregation, and stores and manages information at different levels of granularity. These features make the data easier to use in informed decision making.
- **Database design:** An OLTP system usually adopts an entity-relationship (ER) data model and an application-oriented database design. An OLAP system typically adopts either a *star* or *snowflake* model (to be discussed in Section 3.2.2) and a subject-oriented database design.
- **View:** An OLTP system focuses mainly on the current data within an enterprise or department, without referring to historical data or data in different organizations. In contrast, an OLAP system often spans multiple versions of a database schema, due to the evolutionary process of an organization. OLAP systems also deal with information that originates from different organizations, integrating information from many data stores. Because of their huge volume, OLAP data are stored on multiple storage media.
- **Access patterns:** The access patterns of an OLTP system consist mainly of short, atomic transactions. Such a system requires concurrency control and recovery mechanisms. However, accesses to OLAP systems are mostly read-only operations (since most data warehouses store historical rather than up-to-date information), although many could be complex queries.

Other features that distinguish between OLTP and OLAP systems include database size, frequency of operations, and performance metrics. These are summarized in Table 3.1.

Table 3.1: Comparison between OLTP and OLAP systems.

Feature	OLTP	OLAP
Characteristic	operational processing	informational processing
Orientation	transaction	analysis
User	clerk, DBA, database professional	knowledge worker (e.g., manager, executive, analyst)
Function	day-to-day operations	long-term informational requirements, decision support
DB design	ER based, application-oriented	star/snowflake, subject-oriented
Data	current; guaranteed up-to-date	historical; accuracy maintained over time
Summarization	primitive, highly detailed	summarized, consolidated
View	detailed, flat relational	summarized, multidimensional
Unit of work	short, simple transaction	complex query
Access	read/write	mostly read
Focus	data in	information out
Operations	index/hash on primary key	lots of scans
Number of records accessed	tens	millions
Number of users	thousands	hundreds
DB size	100 MB to GB	100 GB to TB
Priority	high performance, high availability	high flexibility, end-user autonomy
Metric	transaction throughput	query throughput, response time

NOTE: Table is partially based on [CD97].

### 3.1.2 But, Why Have a Separate Data Warehouse?

Since operational databases store huge amounts of data, you may wonder, “*why not perform on-line analytical processing directly on such databases instead of spending additional time and resources to construct a separate data warehouse?*” A major reason for such a separation is to help promote the *high performance of both systems*. An operational database is designed and tuned from known tasks and workloads, such as indexing and hashing using primary keys, searching for particular records, and optimizing “canned” queries. On the other hand, data warehouse queries are often complex. They involve the computation of large groups of data at summarized levels, and may require the use of special data organization, access, and implementation methods based on multidimensional views. Processing OLAP queries in operational databases would substantially degrade the performance of operational tasks.

Moreover, an operational database supports the concurrent processing of multiple transactions. Concurrency control and recovery mechanisms, such as locking and logging, are required to ensure the consistency and robustness of transactions. An OLAP query often needs read-only access of data records for summarization and aggregation. Concurrency control and recovery mechanisms, if applied for such OLAP operations, may jeopardize the execution of concurrent transactions and thus substantially reduce the throughput of an OLTP system.

Finally, the separation of operational databases from data warehouses is based on the different structures, contents, and uses of the data in these two systems. Decision support requires historical data, whereas operational databases do not typically maintain historical data. In this context, the data in operational databases, though abundant, is usually far from complete for decision making. Decision support requires consolidation (such as aggregation and summarization) of data from heterogeneous sources, resulting in high-quality, clean, and integrated data. In contrast, operational databases contain only detailed raw data, such as transactions, which need to be consolidated before analysis. Since the two systems provide quite different functionalities and require different kinds of data, it is presently necessary to maintain separate databases. However, many vendors of operational relational database management systems are beginning to optimize such systems so as to support OLAP queries. As this trend continues, the separation between OLTP and OLAP systems is expected to decrease.

## 3.2 A Multidimensional Data Model

Data warehouses and OLAP tools are based on a **multidimensional data model**. This model views data in the form of a *data cube*. In this section, you will learn how data cubes model  $n$ -dimensional data. You will also learn about concept hierarchies and how they can be used in basic OLAP operations to allow interactive mining at multiple levels of abstraction.

### 3.2.1 From Tables and Spreadsheets to Data Cubes

“What is a *data cube*?” A **data cube** allows data to be modeled and viewed in multiple dimensions. It is defined by dimensions and facts.

In general terms, **dimensions** are the perspectives or entities with respect to which an organization wants to keep records. For example, *Allelectronics* may create a *sales* data warehouse in order to keep records of the store’s sales with respect to the dimensions *time*, *item*, *branch*, and *location*. These dimensions allow the store to keep track of things like monthly sales of items, and the branches and locations at which the items were sold. Each dimension may have a table associated with it, called a **dimension table**, which further describes the dimension. For example, a dimension table for *item* may contain the attributes *item\_name*, *brand*, and *type*. Dimension tables can be specified by users or experts, or automatically generated and adjusted based on data distributions.

A multidimensional data model is typically organized around a central theme, like *sales*, for instance. This theme is represented by a fact table. **Facts** are numerical measures. Think of them as the quantities by which we want to analyze relationships between dimensions. Examples of facts for a sales data warehouse include *dollars\_sold* (sales amount in dollars), *units\_sold* (number of units sold), and *amount\_budgeted*. The **fact table** contains the names of the *facts*, or measures, as well as keys to each of the related dimension tables. You will soon get a clearer picture of how this works when we look at multidimensional schemas.

Table 3.2: A 2-D view of sales data for *Allelectronics* according to the dimensions *time* and *item*, where the sales are from branches located in the city of Vancouver. The measure displayed is *dollars\_sold* (in thousands).

<i>time</i> (quarter)	<i>location</i> = “Vancouver”			
	<i>item</i> (type)			
	home entertainment	computer	phone	security
Q1	605	825	14	400
Q2	680	952	31	512
Q3	812	1023	30	501
Q4	927	1038	38	580

Although we usually think of cubes as 3-D geometric structures, in data warehousing the data cube is  $n$ -dimensional. To gain a better understanding of data cubes and the multidimensional data model, let’s start by looking at a simple 2-D data cube that is, in fact, a table or spreadsheet for sales data from *Allelectronics*. In particular, we will look at the *Allelectronics* sales data for items sold per quarter in the city of Vancouver. These data are shown in Table 3.2. In this 2-D representation, the sales for Vancouver are shown with respect to the *time* dimension (organized in quarters) and the *item* dimension (organized according to the types of items sold). The fact or measure displayed is *dollars\_sold* (in thousands).

Now, suppose that we would like to view the sales data with a third dimension. For instance, suppose we would like to view the data according to *time*, *item*, as well as *location* for the cities Chicago, New York, Toronto, and Vancouver. These 3-D data are shown in Table 3.3. The 3-D data of Table 3.3 are represented as a series of 2-D tables. Conceptually, we may also represent the same data in the form of a 3-D data cube, as in Figure 3.1.

Suppose that we would now like to view our sales data with an additional fourth dimension, such as *supplier*.

Table 3.3: A 3-D view of sales data for *AllElectronics*, according to the dimensions *time*, *item*, and *location*. The measure displayed is *dollars\_sold* (in thousands).

	<i>location</i> = "Chicago"				<i>location</i> = "New York"				<i>location</i> = "Toronto"				<i>location</i> = "Vancouver"			
<i>t</i> <i>i</i> <i>m</i> <i>e</i>	<i>item</i>				<i>item</i>				<i>item</i>				<i>item</i>			
	home ent.	comp.	phone	sec.	home ent.	comp.	phone	sec.	home ent.	comp.	phone	sec.	home ent.	comp.	phone	sec.
Q1	854	882	89	623	1087	968	38	872	818	746	43	591	605	825	14	400
Q2	943	890	64	698	1130	1024	41	925	894	769	52	682	680	952	31	512
Q3	1032	924	59	789	1034	1048	45	1002	940	795	58	728	812	1023	30	501
Q4	1129	992	63	870	1142	1091	54	984	978	864	59	784	927	1038	38	580

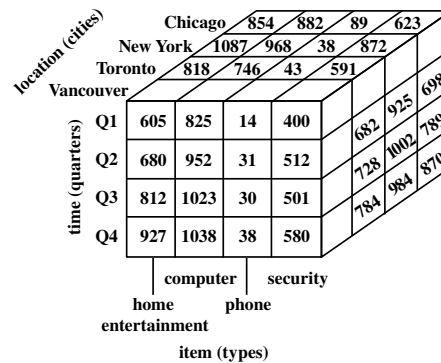


Figure 3.1: A 3-D data cube representation of the data in Table 3.3, according to the dimensions *time*, *item*, and *location*. The measure displayed is *dollars\_sold* (in thousands). [TO EDITOR For consistency with text, please kindly show *time*, *item*, and *location* in italics since they are dimension names.]

Viewing things in 4-D becomes tricky. However, we can think of a 4-D cube as being a series of 3-D cubes, as shown in Figure 3.2. If we continue in this way, we may display any  $n$ -D data as a series of  $(n - 1)$ -D “cubes.” The data cube is a metaphor for multidimensional data storage. The actual physical storage of such data may differ from its logical representation. The important thing to remember is that data cubes are  $n$ -dimensional and do not confine data to 3-D.

The above tables show the data at different degrees of summarization. In the data warehousing research literature, a data cube such as each of the above is often referred to as a **cuboid**. Given a set of dimensions, we can generate a cuboid for each of the possible subsets of the given dimensions. The result would form a *lattice* of cuboids, each showing the data at a different level of summarization, or **group by**. The lattice of cuboids is then referred to as a data cube. Figure 3.3 shows a lattice of cuboids forming a data cube for the dimensions *time*, *item*, *location*, and *supplier*.

The cuboid that holds the lowest level of summarization is called the **base cuboid**. For example, the 4-D cuboid in Figure 3.2 is the base cuboid for the given *time*, *item*, *location*, and *supplier* dimensions. Figure 3.1 is a 3-D (nonbase) cuboid for *time*, *item*, and *location*, summarized for all suppliers. The 0-D cuboid, which holds the highest level of summarization, is called the **apex cuboid**. In our example, this is the total sales, or *dollars\_sold*, summarized over all four dimensions. The apex cuboid is typically denoted by **all**.

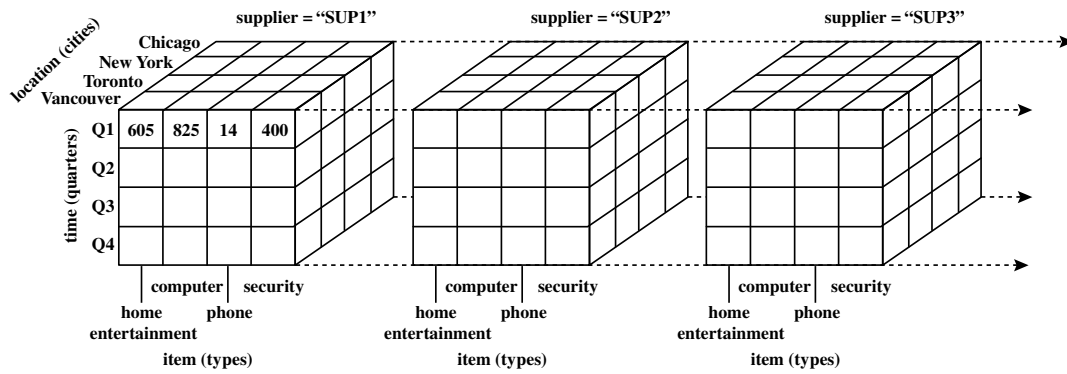


Figure 3.2: A 4-D data cube representation of sales data, according to the dimensions *time*, *item*, *location*, and *supplier*. The measure displayed is *dollars\_sold* (in thousands). For improved readability, only some of the cube values are shown. [TO EDITOR For consistency, please show *time*, *item*, *location* and *supplier* in italics since they are dimension names.]

### 3.2.2 Stars, Snowflakes, and Fact Constellations: Schemas for Multidimensional Databases

The entity-relationship data model is commonly used in the design of relational databases, where a database schema consists of a set of entities and the relationships between them. Such a data model is appropriate for on-line transaction processing. A data warehouse, however, requires a concise, subject-oriented schema that facilitates on-line data analysis.

The most popular data model for a data warehouse is a **multidimensional model**. Such a model can exist in the form of a **star schema**, a **snowflake schema**, or a **fact constellation schema**. Let's have a look at each of these schema types.

**Star schema:** The most common modeling paradigm is the star schema, in which the data warehouse contains (1) a large central table (**fact table**) containing the bulk of the data, with no redundancy, and (2) a set of smaller attendant tables (**dimension tables**), one for each dimension. The schema graph resembles a starburst, with the dimension tables displayed in a radial pattern around the central fact table.

**Example 3.1 Star schema.** A star schema for *AllElectronics* sales is shown in Figure 3.4. Sales are considered along four dimensions, namely, *time*, *item*, *branch*, and *location*. The schema contains a central fact table for *sales* that contains keys to each of the four dimensions, along with two measures: *dollars\_sold* and *units\_sold*. To minimize the size of the fact table, dimension identifiers (such as *time\_key* and *item\_key*) are system-generated identifiers. ■

Notice that in the star schema, each dimension is represented by only one table, and each table contains a set of attributes. For example, the *location* dimension table contains the attribute set  $\{location\_key, street, city, province\_or\_state, country\}$ . This constraint may introduce some redundancy. For example, "Vancouver" and "Victoria" are both cities in the Canadian province of British Columbia. Entries for such cities in the *location* dimension table will create redundancy among the attributes *province\_or\_state* and *country*, that is, (... , Vancouver, British Columbia, Canada) and (... , Victoria, British Columbia, Canada). Moreover, the attributes within a dimension table may form either a hierarchy (total order) or a lattice (partial order).

**Snowflake schema:** The snowflake schema is a variant of the star schema model, where some dimension tables are *normalized*, thereby further splitting the data into additional tables. The resulting schema graph forms a shape similar to a snowflake.

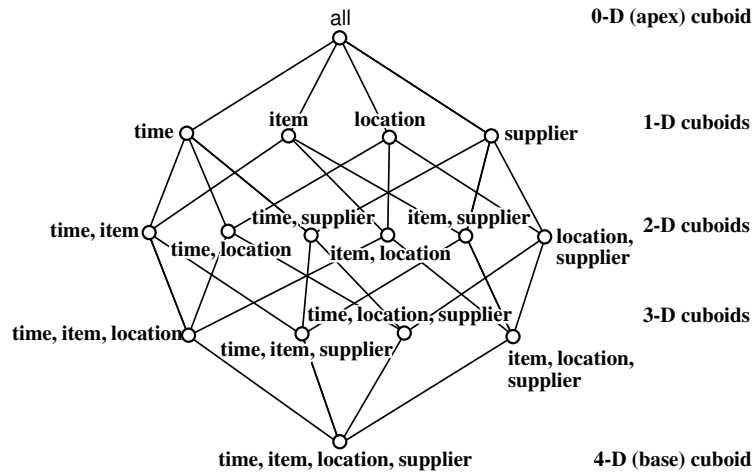


Figure 3.3: Lattice of cuboids, making up a 4-D data cube for the dimensions *time*, *item*, *location*, and *supplier*. Each cuboid represents a different degree of summarization. [TO EDITOR For consistency, please show *time*, *item*, *location* and *supplier* in italics since they are dimension names.]

The major difference between the snowflake and star schema models is that the dimension tables of the snowflake model may be kept in normalized form to reduce redundancies. Such a table is easy to maintain and saves storage space. However, this saving of space is negligible in comparison to the typical magnitude of the fact table. Furthermore, the snowflake structure can reduce the effectiveness of browsing since more joins will be needed to execute a query. Consequently, the system performance may be adversely impacted. Hence, although the snowflake schema reduces redundancy, it is not as popular as the star schema in data warehouse design.

**Example 3.2 Snowflake schema.** A snowflake schema for *AllElectronics* sales is given in Figure 3.5. Here, the *sales* fact table is identical to that of the star schema in Figure 3.4. The main difference between the two schemas is in the definition of dimension tables. The single dimension table for *item* in the star schema is normalized in the snowflake schema, resulting in new *item* and *supplier* tables. For example, the *item* dimension table now contains the attributes *item\_key*, *item\_name*, *brand*, *type*, and *supplier\_key*, where *supplier\_key* is linked to the *supplier* dimension table, containing *supplier\_key* and *supplier\_type* information. Similarly, the single dimension table for *location* in the star schema can be normalized into two new tables: *location* and *city*. The *city\_key* in the new *location* table links to the *city* dimension. Notice that further normalization can be performed on *province\_or\_state* and *country* in the snowflake schema shown in Figure 3.5, when desirable. ■

**Fact constellation:** Sophisticated applications may require multiple fact tables to *share* dimension tables. This kind of schema can be viewed as a collection of stars, and hence is called a **galaxy schema** or a **fact constellation**.

**Example 3.3 Fact constellation.** A fact constellation schema is shown in Figure 3.6. This schema specifies two fact tables, *sales* and *shipping*. The *sales* table definition is identical to that of the star schema (Figure 3.4). The *shipping* table has five dimensions, or keys: *item\_key*, *time\_key*, *shipper\_key*, *from\_location*, and *to\_location*, and two measures: *dollars\_cost* and *units\_shipped*. A fact constellation schema allows dimension tables to be shared between fact tables. For example, the dimensions tables for *time*, *item*, and *location* are shared between both the *sales* and *shipping* fact tables. ■

In data warehousing, there is a distinction between a data warehouse and a data mart. A data warehouse collects information about subjects that span the *entire organization*, such as *customers*, *items*, *sales*, *assets*, and



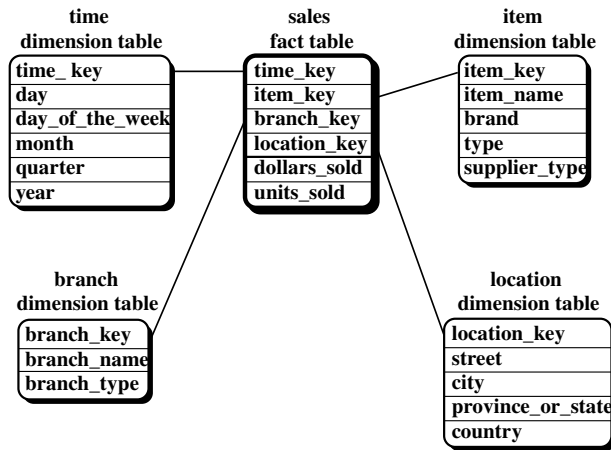


Figure 3.4: Star schema of a data warehouse for sales. [TO EDITOR For consistency, please italicize *time*, *item*, *branch*, *location* and *sales*.]

*personnel*, and thus its scope is *enterprise-wide*. For data warehouses, the fact constellation schema is commonly used since it can model multiple, interrelated subjects. A **data mart**, on the other hand, is a department subset of the data warehouse that focuses on selected subjects, and thus its scope is *department-wide*. For data marts, the *star* or *snowflake* schema are commonly used since both are geared towards modeling single subjects, although the star schema is more popular and efficient.

### 3.2.3 Examples for Defining Star, Snowflake, and Fact Constellation Schemas

“How can I define a multidimensional schema for my data?” Just as relational query languages like SQL can be used to specify relational queries, a **data mining query language** can be used to specify data mining tasks. In particular, we examine how to define data warehouses and data marts in our SQL-based data mining query language, **DMQL**.

Data warehouses and data marts can be defined using two language primitives, one for *cube definition* and one for *dimension definition*. The *cube definition* statement has the following syntax.

```
define cube <cube_name> [(<dimension_list>)] : <measure_list>
```

The *dimension definition* statement has the following syntax.

```
define dimension <dimension_name> as ((<attribute_or_dimension_list>))
```

Let’s look at examples of how to define the star, snowflake, and fact constellation schemas of Examples 3.1 to 3.3 using DMQL. DMQL keywords are displayed in **sans serif** font.

**Example 3.4 Star schema definition.** The star schema of Example 3.1 and Figure 3.4 is defined in DMQL as follows.

```
define cube sales_star [time, item, branch, location]:
    dollars_sold = sum(sales_in_dollars), units_sold = count(*)
define dimension time as (time_key, day, day_of_week, month, quarter, year)
define dimension item as (item_key, item_name, brand, type, supplier_type)
define dimension branch as (branch_key, branch_name, branch_type)
define dimension location as (location_key, street, city, province_or_state, country)
```

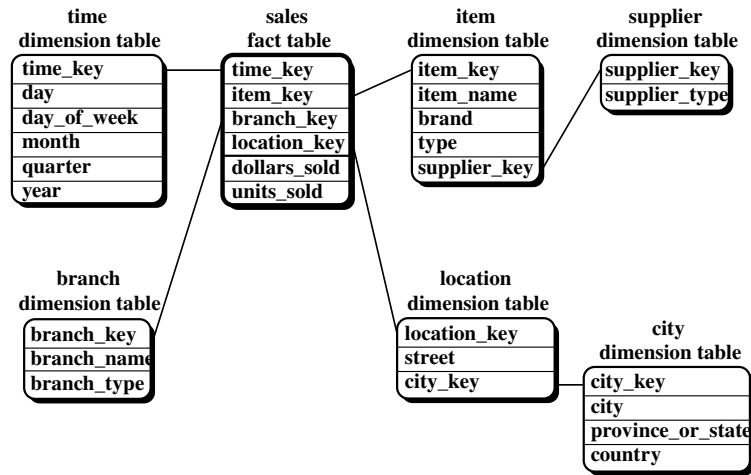


Figure 3.5: Snowflake schema of a data warehouse for sales. [TO EDITOR For consistency, please italicize *time*, *item*, *supplier*, *branch*, *location*, *sales*, and *city*.]

The `define cube` statement defines a data cube called *sales\_star*, which corresponds to the central *sales* fact table of Example 3.1. This command specifies the dimensions and the two measures, *dollars\_sold* and *units\_sold*. The data cube has four dimensions, namely, *time*, *item*, *branch*, and *location*. A `define dimension` statement is used to define each of the dimensions. ■

**Example 3.5 Snowflake schema definition.** The snowflake schema of Example 3.2 and Figure 3.5 is defined in DMQL as follows.

```

define cube sales_snowflake [time, item, branch, location]:
    dollars_sold = sum(sales_in.dollars), units_sold = count(*)
define dimension time as (time_key, day, day_of_week, month, quarter, year)
define dimension item as (item_key, item_name, brand, type, supplier (supplier_key, supplier_type))
define dimension branch as (branch_key, branch_name, branch_type)
define dimension location as (location_key, street, city (city_key, city, province_or_state, country))

```

This definition is similar to that of *sales\_star* (Example 3.4), except that, here, the *item* and *location* dimension tables are normalized. For instance, the *item* dimension of the *sales\_star* data cube has been normalized in the *sales\_snowflake* cube into two dimension tables, *item* and *supplier*. Note that the dimension definition for *supplier* is specified within the definition for *item*. Defining *supplier* in this way implicitly creates a *supplier\_key* in the *item* dimension table definition. Similarly, the *location* dimension of the *sales\_star* data cube has been normalized in the *sales\_snowflake* cube into two dimension tables, *location* and *city*. The dimension definition for *city* is specified within the definition for *location*. In this way, a *city\_key* is implicitly created in the *location* dimension table definition. ■

Finally, a fact constellation schema can be defined as a set of interconnected cubes. Below is an example.

**Example 3.6 Fact constellation schema definition.** The fact constellation schema of Example 3.3 and Figure 3.6 is defined in DMQL as follows.

```

define cube sales [time, item, branch, location]:
    dollars_sold = sum(sales_in.dollars), units_sold = count(*)
define dimension time as (time_key, day, day_of_week, month, quarter, year)

```

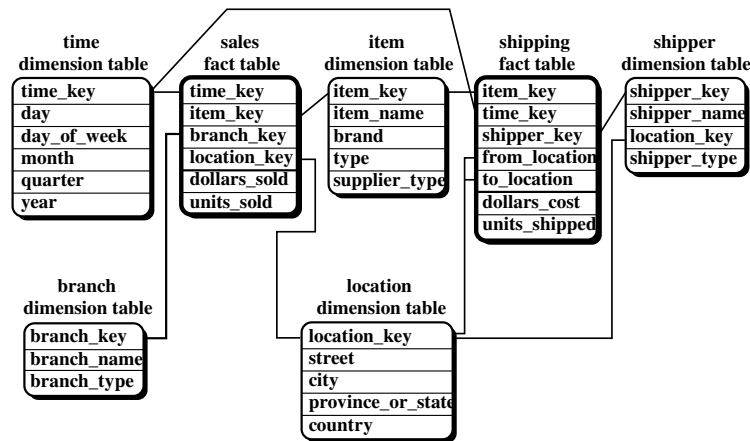


Figure 3.6: Fact constellation schema of a data warehouse for sales and shipping. [TO EDITOR For consistency, please italicize *time*, *item*, *shipping*, *shipper*, *branch*, *location*, and *sales*.]

```

define dimension item as (item_key, item_name, brand, type, supplier_type)
define dimension branch as (branch_key, branch_name, branch_type)
define dimension location as (location_key, street, city, province_or_state, country)
define cube shipping [time, item, shipper, from_location, to_location]:
    dollars_cost = sum(cost_in_dollars), units_shipped = count(*)
define dimension time as time in cube sales
define dimension item as item in cube sales
define dimension shipper as (shipper_key, shipper_name, location as location in cube sales, shipper_type)
define dimension from_location as location in cube sales
define dimension to_location as location in cube sales

```

A `define cube` statement is used to define data cubes for *sales* and *shipping*, corresponding to the two fact tables of the schema of Example 3.3. Note that the *time*, *item*, and *location* dimensions of the *sales* cube are shared with the *shipping* cube. This is indicated for the *time* dimension, for example, as follows. Under the `define cube` statement for *shipping*, the statement “define dimension *time* as *time* in cube *sales*” is specified. ■

### 3.2.4 Measures: Their Categorization and Computation

“How are measures computed?” To answer this question, we first study how measures can be categorized.<sup>1</sup> Note that a *multidimensional point* in the data cube space can be defined by a set of dimension-value pairs, for example,  $\langle \text{time} = \text{“Q1”}, \text{location} = \text{“Vancouver”}, \text{item} = \text{“computer”} \rangle$ . A data cube **measure** is a numerical function that can be evaluated at each point in the data cube space. A measure value is computed for a given point by aggregating the data corresponding to the respective dimension-value pairs defining the given point. We will look at concrete examples of this shortly.

Measures can be organized into three categories, based on the kind of aggregate functions used.

**Distributive:** An aggregate function is *distributive* if it can be computed in a distributed manner as follows. Suppose the data are partitioned into  $n$  sets. We apply the function to each partition, resulting in  $n$  aggregate values. If the result derived by applying the function to the  $n$  aggregate values is the same as that derived by applying the function to the entire data set (without partitioning), the function can be computed in a

<sup>1</sup>This categorization was briefly introduced in Chapter 2 with regards to the computation of measures for descriptive data summaries. We re-examine it here in the context of data cube measures.

distributed manner. For example, `count()` can be computed for a data cube by first partitioning the cube into a set of subcubes, computing `count()` for each subcube, and then summing up the counts obtained for each subcube. Hence, `count()` is a distributive aggregate function. For the same reason, `sum()`, `min()`, and `max()` are distributive aggregate functions. A measure is *distributive* if it is obtained by applying a distributive aggregate function. Distributive measures can be computed efficiently since they can be computed in a distributive manner.

**Algebraic:** An aggregate function is *algebraic* if it can be computed by an algebraic function with  $M$  arguments (where  $M$  is a bounded positive integer), each of which is obtained by applying a distributive aggregate function. For example, `avg()` (average) can be computed by `sum()/count()` where both `sum()` and `count()` are distributive aggregate functions. Similarly, it can be shown that `min_N()` and `max_N()` (which find the  $N$  minimum and  $N$  maximum values, respectively, in a given set), and `standard_deviation()` are algebraic aggregate functions. A measure is *algebraic* if it is obtained by applying an algebraic aggregate function.

**Holistic:** An aggregate function is *holistic* if there is no constant bound on the storage size needed to describe a subaggregate. That is, there does not exist an algebraic function with  $M$  arguments (where  $M$  is a constant) that characterizes the computation. Common examples of holistic functions include `median()`, `mode()`, and `rank()`. A measure is *holistic* if it is obtained by applying a holistic aggregate function.

Most large data cube applications require efficient computation of distributive and algebraic measures. Many efficient techniques for this exist. In contrast, it is difficult to compute holistic measures efficiently. Efficient techniques to *approximate* the computation of some holistic measures, however, do exist. For example, rather than computing the exact `median()`, Equation (2.3) of Chapter 2 can be used to estimate the approximate median value for a large data set. In many cases, such techniques are sufficient to overcome the difficulties of efficient computation of holistic measures.

**Example 3.7 Interpreting measures for data cubes.** Many measures of a data cube can be computed by relational aggregation operations. In Figure 3.4, we saw a star schema for *AllElectronics* sales that contains two measures, namely, *dollars\_sold* and *units\_sold*. In Example 3.4, the *sales\_star* data cube corresponding to the schema was defined using DMQL commands. “But, how are these commands interpreted in order to generate the specified data cube?”

Suppose that the relational database schema of *AllElectronics* is the following:

```
time(time_key, day, day_of_week, month, quarter, year)
item(item_key, item_name, brand, type, supplier_type)
branch(branch_key, branch_name, branch_type)
location(location_key, street, city, province_or_state, country)
sales(time_key, item_key, branch_key, location_key, number_of_units_sold, price)
```

The DMQL specification of Example 3.4 is translated into the following SQL query, which generates the required *sales\_star* cube. Here, the `sum` aggregate function, is used to compute both *dollars\_sold* and *units\_sold*.

```
select s.time_key, s.item_key, s.branch_key, s.location_key,
       sum(s.number_of_units_sold * s.price), sum(s.number_of_units_sold)
from time t, item i, branch b, location l, sales s,
where s.time_key = t.time_key and s.item_key = i.item_key
       and s.branch_key = b.branch_key and s.location_key = l.location_key
group by s.time_key, s.item_key, s.branch_key, s.location_key
```

The cube created in the above query is the base cuboid of the *sales\_star* data cube. It contains all of the dimensions specified in the data cube definition, where the granularity of each dimension is at the **join key** level. A join key is a key that links a fact table and a dimension table. The fact table associated with a base cuboid is sometimes referred to as the **base fact table**.

By changing the **group by** clauses, we can generate other cuboids for the *sales\_star* data cube. For example, instead of grouping by *s.time\_key*, we can group by *t.month*, which will sum up the measures of each group by month. Also, removing “group by *s.branch\_key*” will generate a higher-level cuboid (where sales are summed for all branches, rather than broken down per branch). Suppose we modify the above SQL query by removing *all* of the **group by** clauses. This will result in obtaining the total sum of *dollars\_sold* and the total count of *units\_sold* for the given data. This zero-dimensional cuboid is the apex cuboid of the *sales\_star* data cube. In addition, other cuboids can be generated by applying selection and/or projection operations on the base cuboid, resulting in a lattice of cuboids as described in Section 3.2.1. Each cuboid corresponds to a different degree of summarization of the given data. ■

Most of the current data cube technology confines the measures of multidimensional databases to *numerical data*. However, measures can also be applied to other kinds of data, such as spatial, multimedia, or text data. This will be discussed in future chapters.

### 3.2.5 Concept Hierarchies

A **concept hierarchy** defines a sequence of mappings from a set of low-level concepts to higher-level, more general concepts. Consider a concept hierarchy for the dimension *location*. City values for *location* include Vancouver, Toronto, New York, and Chicago. Each city, however, can be mapped to the province or state to which it belongs. For example, Vancouver can be mapped to British Columbia, and Chicago to Illinois. The provinces and states can in turn be mapped to the country to which they belong, such as Canada or the USA. These mappings form a concept hierarchy for the dimension *location*, mapping a set of low-level concepts (i.e., cities) to higher-level, more general concepts (i.e., countries). The concept hierarchy described above is illustrated in Figure 3.7.

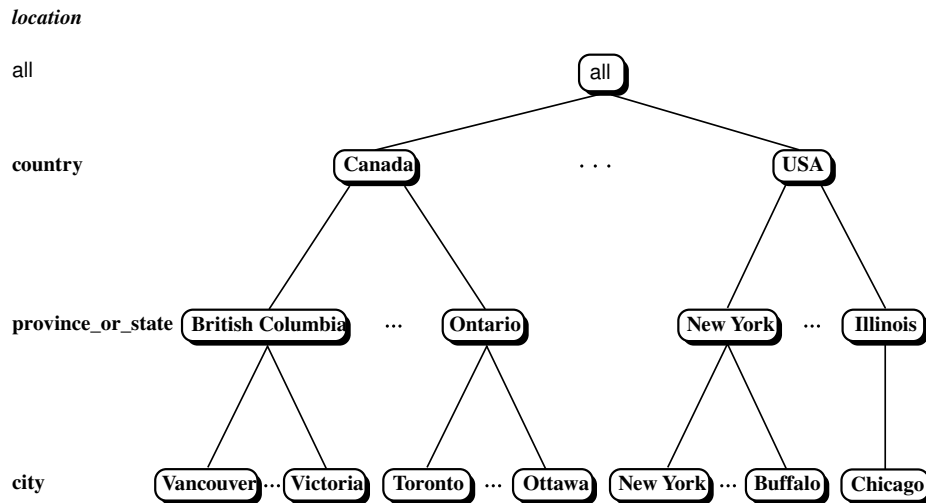


Figure 3.7: A concept hierarchy for the dimension *location*. Due to space limitations, not all of the nodes of the hierarchy are shown (as indicated by the use of “ellipsis” between nodes). [TO EDITOR Please add ellipsis and a node for the city Urbana under the Illinois node. Thanks!]

Many concept hierarchies are implicit within the database schema. For example, suppose that the dimension *location* is described by the attributes *number*, *street*, *city*, *province\_or\_state*, *zipcode*, and *country*. These attributes are related by a total order, forming a concept hierarchy such as “*street* < *city* < *province\_or\_state* < *country*”. This hierarchy is shown in Figure 3.8(a). Alternatively, the attributes of a dimension may be organized in a partial order, forming a lattice. An example of a partial order for the *time* dimension based on the attributes *day*, *week*, *month*, *quarter*, and *year* is “*day* < {*month* < *quarter*; *week*} < *year*”.<sup>2</sup> This lattice structure is shown

<sup>2</sup>Since a *week* often crosses the boundary of two consecutive months, it is usually not treated as a lower abstraction of *month*.

in Figure 3.8(b). A concept hierarchy that is a total or partial order among attributes in a database schema is called a **schema hierarchy**. Concept hierarchies that are common to many applications may be predefined in the data mining system, such as the concept hierarchy for *time*. Data mining systems should provide users with the flexibility to tailor predefined hierarchies according to their particular needs. For example, users may like to define a fiscal year starting on April 1, or an academic year starting on September 1.

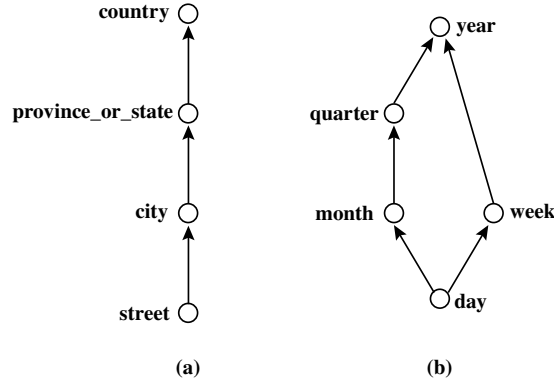


Figure 3.8: Hierarchical and lattice structures of attributes in warehouse dimensions: (a) a hierarchy for *location*; (b) a lattice for *time*.

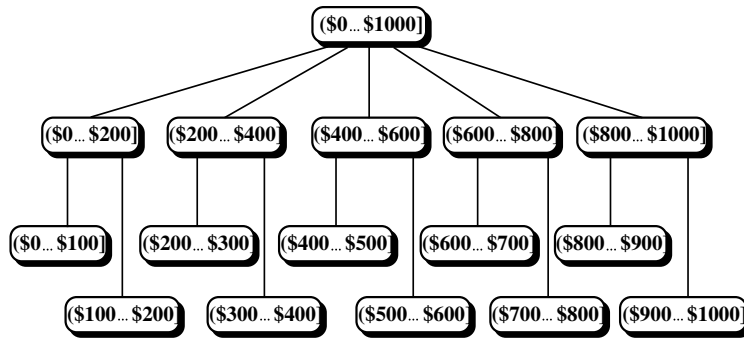


Figure 3.9: A concept hierarchy for the attribute *price*. [TO EDITOR Please change so that all of the categories 0-100, 100-200, 200-300, etc. appear on the same level! (This would be better, conceptually, for the reader! Thanks.)]

Concept hierarchies may also be defined by discretizing or grouping values for a given dimension or attribute, resulting in a **set-grouping hierarchy**. A total or partial order can be defined among groups of values. An example of a set-grouping hierarchy is shown in Figure 3.9 for the dimension *price*, where an interval  $(\$X \dots \$Y]$  denotes the range from  $\$X$  (exclusive) to  $\$Y$  (inclusive).

There may be more than one concept hierarchy for a given attribute or dimension, based on different user viewpoints. For instance, a user may prefer to organize *price* by defining ranges for *inexpensive*, *moderately-priced*, and *expensive*.

Concept hierarchies may be provided manually by system users, domain experts, knowledge engineers, or automatically generated based on statistical analysis of the data distribution. The automatic generation of concept hierarchies is discussed in Chapter 2 as a preprocessing step in preparation for data mining.

Concept hierarchies allow data to be handled at varying levels of abstraction, as we shall see in the following

---

Instead, it is often treated as a lower abstraction of *year*, since a year contains approximately 52 weeks.

subsection.

### 3.2.6 OLAP Operations in the Multidimensional Data Model

---

To Editor: Fig. 2.10 of p. 59 (of Edition 1) should be printed here. However, since including this in .dvi file will not generate correct .ps file, we do not print the graph here.

---

Figure 3.10: Examples of typical OLAP operations on multidimensional data. [TO EDITOR For consistency, please italicize *time*, *item*, and *location*.]

“How are concept hierarchies useful in OLAP?” In the multidimensional model, data are organized into multiple dimensions, and each dimension contains multiple levels of abstraction defined by concept hierarchies. This organization provides users with the flexibility to view data from different perspectives. A number of OLAP data cube operations exist to materialize these different views, allowing interactive querying and analysis of the data at hand. Hence, OLAP provides a user-friendly environment for interactive data analysis.

**Example 3.8 OLAP operations.** Let’s have a look at some typical OLAP operations for multidimensional data. Each of the operations described below is illustrated in Figure 3.10. At the center of the figure is a data cube for *AllElectronics* sales. The cube contains the dimensions *location*, *time*, and *item*, where *location* is aggregated with respect to city values, *time* is aggregated with respect to quarters, and *item* is aggregated with respect to item types. To aid in our explanation, we refer to this cube as the central cube. The measure displayed is *dollars\_sold* (in thousands). (For improved readability, only some of the cubes’ cell values are shown.) The data examined are for the cities Chicago, New York, Toronto, and Vancouver.

**Roll-up:** The roll-up operation (also called the *drill-up* operation by some vendors) performs aggregation on a data cube, either by *climbing up a concept hierarchy* for a dimension or by *dimension reduction*. Figure 3.10 shows the result of a roll-up operation performed on the central cube by climbing up the concept hierarchy for *location* given in Figure 3.7. This hierarchy was defined as the total order “*street* < *city* < *province\_or\_state* < *country*”. The roll-up operation shown aggregates the data by ascending the *location* hierarchy from the level of *city* to the level of *country*. In other words, rather than grouping the data by city, the resulting cube groups the data by country.

When roll-up is performed by dimension reduction, one or more dimensions are removed from the given cube. For example, consider a sales data cube containing only the two dimensions *location* and *time*. Roll-up may be performed by removing, say, the *time* dimension, resulting in an aggregation of the total sales by location, rather than by location and by time.

**Drill-down:** Drill-down is the reverse of roll-up. It navigates from less detailed data to more detailed data. Drill-down can be realized by either *stepping down a concept hierarchy* for a dimension or *introducing additional dimensions*. Figure 3.10 shows the result of a drill-down operation performed on the central cube by stepping

down a concept hierarchy for *time* defined as “*day < month < quarter < year*”. Drill-down occurs by descending the *time* hierarchy from the level of *quarter* to the more detailed level of *month*. The resulting data cube details the total sales per month rather than summarized by quarter.

Since a drill-down adds more detail to the given data, it can also be performed by adding new dimensions to a cube. For example, a drill-down on the central cube of Figure 3.10 can occur by introducing an additional dimension, such as *customer\_group*.

**Slice and dice:** The *slice* operation performs a selection on one dimension of the given cube, resulting in a subcube. Figure 3.10 shows a slice operation where the sales data are selected from the central cube for the dimension *time* using the criterion *time* = “*Q1*”. The *dice* operation defines a subcube by performing a selection on two or more dimensions. Figure 3.10 shows a dice operation on the central cube based on the following selection criteria that involve three dimensions: (*location* = “*Toronto*” or “*Vancouver*”) and (*time* = “*Q1*” or “*Q2*”) and (*item* = “*home entertainment*” or “*computer*”).

**Pivot (rotate):** *Pivot* (also called *rotate*) is a visualization operation that rotates the data axes in view in order to provide an alternative presentation of the data. Figure 3.10 shows a pivot operation where the *item* and *location* axes in a 2-D slice are rotated. Other examples include rotating the axes in a 3-D cube, or transforming a 3-D cube into a series of 2-D planes.

**Other OLAP operations:** Some OLAP systems offer additional drilling operations. For example, **drill-across** executes queries involving (i.e., across) more than one fact table. The **drill-through** operation makes use of relational SQL facilities to drill through the bottom level of a data cube down to its back-end relational tables.

Other OLAP operations may include ranking the top *N* or bottom *N* items in lists, as well as computing moving averages, growth rates, interests, internal rates of return, depreciation, currency conversions, and statistical functions. ■

OLAP offers analytical modeling capabilities, including a calculation engine for deriving ratios, variance, and so on, and for computing measures across multiple dimensions. It can generate summarizations, aggregations, and hierarchies at each granularity level and at every dimension intersection. OLAP also supports functional models for forecasting, trend analysis, and statistical analysis. In this context, an OLAP engine is a powerful data analysis tool.

## OLAP Systems versus Statistical Databases

Many of the characteristics of OLAP systems, such as the use of a multidimensional data model and concept hierarchies, the association of measures with dimensions, and the notions of roll-up and drill-down, also exist in earlier work on statistical databases (SDBs). A **statistical database** is a database system that is designed to support statistical applications. Similarities between the two types of systems are rarely discussed, mainly due to differences in terminology and application domains.

OLAP and SDB systems, however, have distinguishing differences. While SDBs tend to focus on socioeconomic applications, OLAP has been targeted for business applications. Privacy issues regarding concept hierarchies are a major concern for SDBs. For example, given summarized socioeconomic data, it is controversial to allow users to view the corresponding low-level data. Finally, unlike SDBs, OLAP systems are designed for handling huge amounts of data efficiently.

### 3.2.7 A Starnet Query Model for Querying Multidimensional Databases

The querying of multidimensional databases can be based on a **starnet model**. A starnet model consists of radial lines emanating from a central point, where each line represents a concept hierarchy for a dimension. Each abstraction level in the hierarchy is called a **footprint**. These represent the granularities available for use by OLAP operations such as drill-down and roll-up.



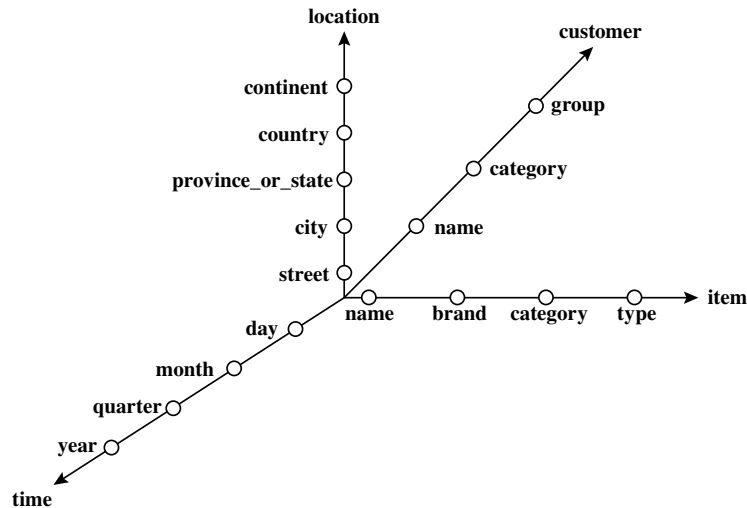


Figure 3.11: Modeling business queries: a starnet model. [TO EDITOR For consistency, please italicize *time*, *item*, *customer*, and *location*.]

**Example 3.9 Starnet.** A starnet query model for the *AllElectronics* data warehouse is shown in Figure 3.11. This starnet consists of four radial lines, representing concept hierarchies for the dimensions *location*, *customer*, *item*, and *time*, respectively. Each line consists of footprints representing abstraction levels of the dimension. For example, the *time* line has four footprints: “day,” “month,” “quarter,” and “year.” A concept hierarchy may involve a single attribute (like *date* for the *time* hierarchy), or several attributes (e.g., the concept hierarchy for *location* involves the attributes *street*, *city*, *province\_or\_state*, and *country*). In order to examine the item sales at *AllElectronics*, users can roll up along the *time* dimension from *month* to *quarter*, or, say, drill down along the *location* dimension from *country* to *city*. Concept hierarchies can be used to **generalize** data by replacing low-level values (such as “day” for the *time* dimension) by higher-level abstractions (such as “year”), or to **specialize** data by replacing higher-level abstractions with lower-level values. ■

### 3.3 Data Warehouse Architecture

In this section, we discuss issues regarding data warehouse architecture. Section 3.3.1 gives a general account of how to design and construct a data warehouse. Section 3.3.2 describes a three-tier data warehouse architecture. Section 3.3.3 describes back-end tools and utilities for data warehouses. Section 3.3.4 describes the metadata repository. Section 3.3.5 presents various types of warehouse servers for OLAP processing.

#### 3.3.1 Steps for the Design and Construction of Data Warehouses

This subsection presents a business analysis framework for data warehouse design. The basic steps involved in the design process are also described.

##### The Design of a Data Warehouse: A Business Analysis Framework

“What can business analysts gain from having a data warehouse?” First, having a data warehouse may provide a *competitive advantage* by presenting relevant information from which to measure performance and make critical adjustments in order to help win over competitors. Second, a data warehouse can enhance business *productivity* since it is able to quickly and efficiently gather information that accurately describes the organization. Third, a

data warehouse facilitates *customer relationship management* since it provides a consistent view of customers and items across all lines of business, all departments, and all markets. Finally, a data warehouse may bring about *cost reduction* by tracking trends, patterns, and exceptions over long periods of time in a consistent and reliable manner.

To design an effective data warehouse we need to understand and analyze business needs and construct a *business analysis framework*. The construction of a large and complex information system can be viewed as the construction of a large and complex building, for which the owner, architect, and builder have different views. These views are combined to form a complex framework that represents the top-down, business-driven, or owner's perspective, as well as the bottom-up, builder-driven, or implementor's view of the information system.

Four different views regarding the design of a data warehouse must be considered: the *top-down view*, the *data source view*, the *data warehouse view*, and the *business query view*.

- The **top-down view** allows the selection of the relevant information necessary for the data warehouse. This information matches the current and coming business needs.
- The **data source view** exposes the information being captured, stored, and managed by operational systems. This information may be documented at various levels of detail and accuracy, from individual data source tables to integrated data source tables. Data sources are often modeled by traditional data modeling techniques, such as the entity-relationship model or CASE (computer-aided software engineering) tools.
- The **data warehouse view** includes fact tables and dimension tables. It represents the information that is stored inside the data warehouse, including precalculated totals and counts, as well as information regarding the source, date, and time of origin, added to provide historical context.
- Finally, the **business query view** is the perspective of data in the data warehouse from the viewpoint of the end user.

Building and using a data warehouse is a complex task since it requires *business skills*, *technology skills*, and *program management skills*. Regarding *business skills*, building a data warehouse involves understanding how such systems store and manage their data, how to build **extractors** that transfer data from the operational system to the data warehouse, and how to build **warehouse refresh software** that keeps the data warehouse reasonably up-to-date with the operational system's data. Using a data warehouse involves understanding the significance of the data it contains, as well as understanding and translating the business requirements into queries that can be satisfied by the data warehouse. Regarding *technology skills*, data analysts are required to understand how to make assessments from quantitative information and derive facts based on conclusions from historical information in the data warehouse. These skills include the ability to discover patterns and trends, to extrapolate trends based on history and look for anomalies or paradigm shifts, and to present coherent managerial recommendations based on such analysis. Finally, *program management skills* involve the need to interface with many technologies, vendors, and end users in order to deliver results in a timely and cost-effective manner.

### The Process of Data Warehouse Design

A data warehouse can be built using a *top-down approach*, a *bottom-up approach*, or a *combination of both*. The **top-down approach** starts with the overall design and planning. It is useful in cases where the technology is mature and well known, and where the business problems that must be solved are clear and well understood. The **bottom-up approach** starts with experiments and prototypes. This is useful in the early stage of business modeling and technology development. It allows an organization to move forward at considerably less expense and to evaluate the benefits of the technology before making significant commitments. In the **combined approach**, an organization can exploit the planned and strategic nature of the top-down approach while retaining the rapid implementation and opportunistic application of the bottom-up approach.

From the software engineering point of view, the design and construction of a data warehouse may consist of the following steps: *planning*, *requirements study*, *problem analysis*, *warehouse design*, *data integration and testing*,

and finally *deployment of the data warehouse*. Large software systems can be developed using two methodologies: the *waterfall method* or the *spiral method*. The **waterfall method** performs a structured and systematic analysis at each step before proceeding to the next, which is like a waterfall, falling from one step to the next. The **spiral method** involves the rapid generation of increasingly functional systems, with short intervals between successive releases. This is considered a good choice for data warehouse development, especially for data marts, because the turnaround time is short, modifications can be done quickly, and new designs and technologies can be adapted in a timely manner.

In general, the warehouse design process consists of the following steps:

1. Choose a *business process* to model, for example, orders, invoices, shipments, inventory, account administration, sales, or the general ledger. If the business process is organizational and involves multiple complex object collections, a data warehouse model should be followed. However, if the process is departmental and focuses on the analysis of one kind of business process, a data mart model should be chosen.
2. Choose the *grain* of the business process. The grain is the fundamental, atomic level of data to be represented in the fact table for this process, for example, individual transactions, individual daily snapshots, and so on.
3. Choose the *dimensions* that will apply to each fact table record. Typical dimensions are time, item, customer, supplier, warehouse, transaction type, and status.
4. Choose the *measures* that will populate each fact table record. Typical measures are numeric additive quantities like *dollars\_sold* and *units\_sold*.

Since data warehouse construction is a difficult and long-term task, its implementation scope should be clearly defined. The goals of an initial data warehouse implementation should be *specific*, *achievable*, and *measurable*. This involves determining the time and budget allocations, the subset of the organization that is to be modeled, the number of data sources selected, and the number and types of departments to be served.

Once a data warehouse is designed and constructed, the initial deployment of the warehouse includes initial installation, rollout planning, training, and orientation. Platform upgrades and maintenance must also be considered. Data warehouse administration includes data refreshment, data source synchronization, planning for disaster recovery, managing access control and security, managing data growth, managing database performance, and data warehouse enhancement and extension. Scope management includes controlling the number and range of queries, dimensions, and reports; limiting the size of the data warehouse; or limiting the schedule, budget, or resources.

Various kinds of data warehouse design tools are available. **Data warehouse development tools** provide functions to define and edit metadata repository contents (such as schemas, scripts, or rules), answer queries, output reports, and ship metadata to and from relational database system catalogues. **Planning and analysis tools** study the impact of schema changes and of refresh performance when changing refresh rates or time windows.

### 3.3.2 A Three-Tier Data Warehouse Architecture

Data warehouses often adopt a three-tier architecture, as presented in Figure 3.12.

1. The bottom tier is a **warehouse database server** that is almost always a relational database system. Back-end tools and utilities are used to feed data into the bottom tier from operational databases or other external sources (such as customer profile information provided by external consultants). These tools and utilities perform data extraction, cleaning, and transformation (e.g., to merge like data from different sources into a unified format), as well as load and refresh functions to update the data warehouse (Section 3.3.3). The data are extracted using application program interfaces known as **gateways**. A gateway is supported by the underlying DBMS and allows client programs to generate SQL code to be executed at a server. Examples of gateways include ODBC (Open Database Connection) and OLE-DB (Open Linking and Embedding for Databases) by Microsoft, and JDBC (Java Database Connection). This tier also contains a metadata repository, which stores information about the data warehouse and its contents. The metadata repository is further described in Section 3.3.4.

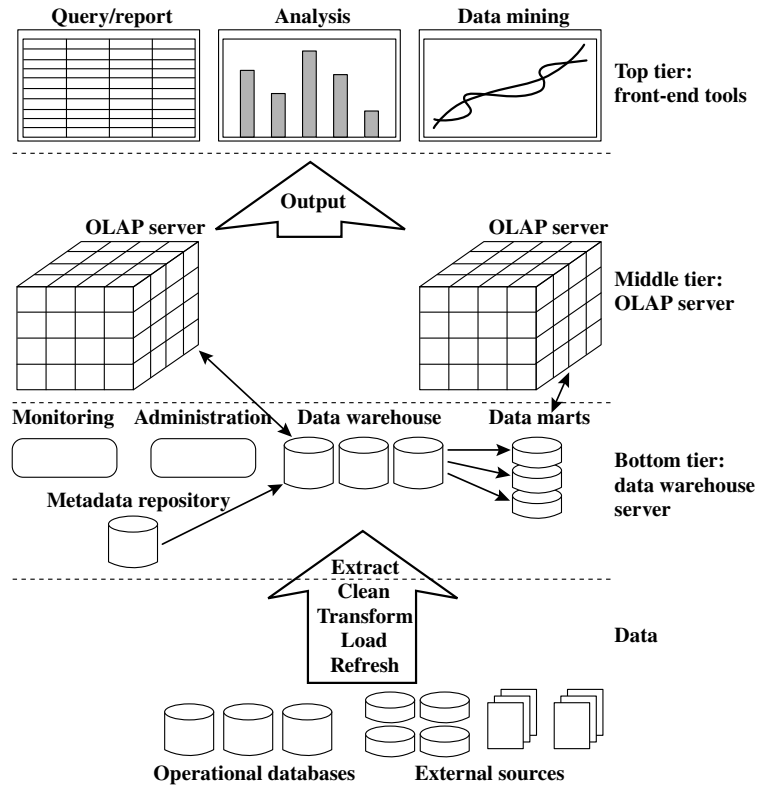


Figure 3.12: A three-tier data warehousing architecture.

2. The middle tier is an **OLAP server** that is typically implemented using either (1) a **relational OLAP (ROLAP)** model, that is, an extended relational DBMS that maps operations on multidimensional data to standard relational operations; or (2) a **multidimensional OLAP (MOLAP)** model, that is, a special-purpose server that directly implements multidimensional data and operations. OLAP servers are discussed in Section 3.3.5.
3. The top tier is a **front-end client layer**, which contains query and reporting tools, analysis tools, and/or data mining tools (e.g., trend analysis, prediction, and so on).

From the architecture point of view, there are three data warehouse models: the *enterprise warehouse*, the *data mart*, and the *virtual warehouse*.

**Enterprise warehouse:** An enterprise warehouse collects all of the information about subjects spanning the entire organization. It provides corporate-wide data integration, usually from one or more operational systems or external information providers, and is cross-functional in scope. It typically contains detailed data as well as summarized data, and can range in size from a few gigabytes to hundreds of gigabytes, terabytes, or beyond. An enterprise data warehouse may be implemented on traditional mainframes, computer superservers, or parallel architecture platforms. It requires extensive business modeling and may take years to design and build.

**Data mart:** A data mart contains a subset of corporate-wide data that is of value to a specific group of users. The scope is confined to specific selected subjects. For example, a marketing data mart may confine its subjects to customer, item, and sales. The data contained in data marts tend to be summarized.

Data marts are usually implemented on low-cost departmental servers that are UNIX/LINUX- or Windows-based. The implementation cycle of a data mart is more likely to be measured in weeks rather than months

or years. However, it may involve complex integration in the long run if its design and planning were not enterprise-wide.

Depending on the source of data, data marts can be categorized as independent or dependent. *Independent* data marts are sourced from data captured from one or more operational systems or external information providers, or from data generated locally within a particular department or geographic area. *Dependent* data marts are sourced directly from enterprise data warehouses.

**Virtual warehouse:** A virtual warehouse is a set of views over operational databases. For efficient query processing, only some of the possible summary views may be materialized. A virtual warehouse is easy to build but requires excess capacity on operational database servers.

“What are the pros and cons of the top-down and bottom-up approaches to data warehouse development?”

The top-down development of an enterprise warehouse serves as a systematic solution and minimizes integration problems. However, it is expensive, takes a long time to develop, and lacks flexibility due to the difficulty in achieving consistency and consensus for a common data model for the entire organization. The bottom-up approach to the design, development, and deployment of independent data marts provides flexibility, low cost, and rapid return of investment. It, however, can lead to problems when integrating various disparate data marts into a consistent enterprise data warehouse.

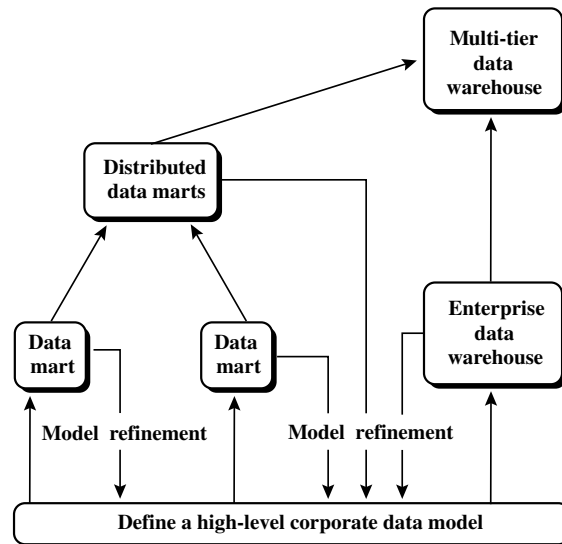


Figure 3.13: A recommended approach for data warehouse development. [TO EDITOR For consistency, please change ‘multi-tier’ to ‘multitier’.]

A recommended method for the development of data warehouse systems is to implement the warehouse in an incremental and evolutionary manner, as shown in Figure 3.13. First, a high-level corporate data model is defined within a reasonably short period of time (such as one or two months) that provides a corporate-wide, consistent, integrated view of data among different subjects and potential usages. This high-level model, although it will need to be refined in the further development of enterprise data warehouses and departmental data marts, will greatly reduce future integration problems. Second, independent data marts can be implemented in parallel with the enterprise warehouse based on the same corporate data model set as above. Third, distributed data marts can be constructed to integrate different data marts via hub servers. Finally, a **multitier data warehouse** is constructed where the enterprise warehouse is the sole custodian of all warehouse data, which is then distributed to the various dependent data marts.

### 3.3.3 Data Warehouse Back-End Tools and Utilities

Data warehouse systems use back-end tools and utilities to populate and refresh their data (Figure 3.12). These tools and utilities include the following functions:

- **Data extraction**, which typically gathers data from multiple, heterogeneous, and external sources
- **Data cleaning**, which detects errors in the data and rectifies them when possible
- **Data transformation**, which converts data from legacy or host format to warehouse format
- **Load**, which sorts, summarizes, consolidates, computes views, checks integrity, and builds indices and partitions
- **Refresh**, which propagates the updates from the data sources to the warehouse

Besides cleaning, loading, refreshing, and metadata definition tools, data warehouse systems usually provide a good set of data warehouse management tools.

Data cleaning and data transformation are important steps in improving the quality of the data and, subsequently, of the data mining results. They are described in Chapter 2 on Data Preprocessing. Since we are mostly interested in the aspects of data warehousing technology related to data mining, we will not get into the details of the remaining tools and recommend interested readers to consult books dedicated to data warehousing technology.

### 3.3.4 Metadata Repository

**Metadata** are data about data. When used in a data warehouse, metadata are the data that define warehouse objects. Figure 3.12 showed a metadata repository within the bottom tier of the data warehousing architecture. Metadata are created for the data names and definitions of the given warehouse. Additional metadata are created and captured for timestamping any extracted data, the source of the extracted data, and missing fields that have been added by data cleaning or integration processes.

A metadata repository should contain the following:

- A description of *the structure of the data warehouse*, which includes the warehouse schema, view, dimensions, hierarchies, and derived data definitions, as well as data mart locations and contents
- *Operational metadata*, which include data lineage (history of migrated data and the sequence of transformations applied to it), currency of data (active, archived, or purged), and monitoring information (warehouse usage statistics, error reports, and audit trails)
- *The algorithms used for summarization*, which include measure and dimension definition algorithms, data on granularity, partitions, subject areas, aggregation, summarization, and predefined queries and reports
- *The mapping from the operational environment to the data warehouse*, which includes source databases and their contents, gateway descriptions, data partitions, data extraction, cleaning, transformation rules and defaults, data refresh and purging rules, and security (user authorization and access control)
- *Data related to system performance*, which include indices and profiles that improve data access and retrieval performance, in addition to rules for the timing and scheduling of refresh, update, and replication cycles
- *Business metadata*, which include business terms and definitions, data ownership information, and charging policies

A data warehouse contains different levels of summarization, of which metadata is one type. Other types include current detailed data (which are almost always on disk), older detailed data (which are usually on tertiary storage), lightly summarized data and highly summarized data (which may or may not be physically housed).

Metadata play a very different role than other data warehouse data, and are important for many reasons. For example, metadata are used as a directory to help the decision support system analyst locate the contents of the data warehouse, as a guide to the mapping of data when the data are transformed from the operational environment to the data warehouse environment, and as a guide to the algorithms used for summarization between the current detailed data and the lightly summarized data, and between the lightly summarized data and the highly summarized data. Metadata should be stored and managed persistently (i.e., on disk).

### 3.3.5 Types of OLAP Servers: ROLAP versus MOLAP versus HOLAP

Logically, OLAP servers present business users with multidimensional data from data warehouses or data marts, without concerns regarding how or where the data are stored. However, the physical architecture and implementation of OLAP servers must consider data storage issues. Implementations of a warehouse server for OLAP processing include the following:

**Relational OLAP (ROLAP) servers:** These are the intermediate servers that stand in between a relational back-end server and client front-end tools. They use a *relational or extended-relational DBMS* to store and manage warehouse data, and OLAP middleware to support missing pieces. ROLAP servers include optimization for each DBMS back end, implementation of aggregation navigation logic, and additional tools and services. ROLAP technology tends to have greater scalability than MOLAP technology. The DSS server of Microstrategy, for example, adopts the ROLAP approach.

**Multidimensional OLAP (MOLAP) servers:** These servers support multidimensional views of data through *array-based multidimensional storage engines*. They map multidimensional views directly to data cube array structures. The advantage of using a data cube is that it allows fast indexing to precomputed summarized data. Notice that with multidimensional data stores, the storage utilization may be low if the data set is sparse. In such cases, sparse matrix compression techniques should be explored (Chapter 4).

Many MOLAP servers adopt a two-level storage representation to handle dense and sparse data sets: denser subcubes are identified and stored as array structures, while sparse subcubes employ compression technology for efficient storage utilization.

**Hybrid OLAP (HOLAP) servers:** The hybrid OLAP approach combines ROLAP and MOLAP technology, benefiting from the greater scalability of ROLAP and the faster computation of MOLAP. For example, a HOLAP server may allow large volumes of detail data to be stored in a relational database, while aggregations are kept in a separate MOLAP store. The Microsoft SQL Server 2000 supports a hybrid OLAP server.

**Specialized SQL servers:** To meet the growing demand of OLAP processing in relational databases, some database system vendors implement specialized SQL servers that provide advanced query language and query processing support for SQL queries over star and snowflake schemas in a read-only environment.

“How are data actually stored in ROLAP and MOLAP architectures?” Let’s first have a look at ROLAP. As its name implies, ROLAP uses relational tables to store data for on-line analytical processing. Recall that the fact table associated with a base cuboid is referred to as a *base fact table*. The base fact table stores data at the abstraction level indicated by the join keys in the schema for the given data cube. Aggregated data can also be stored in fact tables, referred to as **summary fact tables**. Some summary fact tables store both base fact table data and aggregated data, as in Example 3.10. Alternatively, separate summary fact tables can be used for each level of abstraction, to store only aggregated data.

**Example 3.10 A ROLAP data store.** Table 3.4 shows a summary fact table that contains both base fact data and aggregated data. The schema of the table is “ $\langle \text{record\_identifier (RID)}, \text{item}, \dots, \text{day}, \text{month}, \text{quarter}, \text{year}, \text{dollars\_sold} \rangle$ ”, where *day*, *month*, *quarter*, and *year* define the date of sales, and *dollars\_sold* is the sales amount. Consider the tuples with an *RID* of 1001 and 1002, respectively. The data of these tuples are at the base fact level, where the date of sales is October 15, 2003, and October 23, 2003, respectively. Consider the tuple with an *RID* of 5001. This tuple is at a more general level of abstraction than the tuples 1001 and 1002. The *day* value has

been generalized to *all*, so that the corresponding *time* value is October 2003. That is, the *dollars\_sold* amount shown is an aggregation representing the entire month of October 2003, rather than just October 15 or 23, 2003. The special value *all* is used to represent subtotals in summarized data. ■

Table 3.4: Single table for base and summary facts.

<i>RID</i>	<i>item</i>	<i>...</i>	<i>day</i>	<i>month</i>	<i>quarter</i>	<i>year</i>	<i>dollars_sold</i>
1001	TV	...	15	10	Q4	2003	250.60
1002	TV	...	23	10	Q4	2003	175.00
...	...	...	...	...	...	...	...
5001	TV	...	all	10	Q4	2003	45,786.08
...	...	...	...	...	...	...	...

MOLAP uses multidimensional array structures to store data for on-line analytical processing. This structure is discussed in the following section on data warehouse implementation and, in greater detail, in Chapter 4.

Most data warehouse systems adopt a client-server architecture. A relational data store always resides at the data warehouse/data mart server site. A multidimensional data store can reside at either the database server site or the client site.

## 3.4 Data Warehouse Implementation

Data warehouses contain huge volumes of data. OLAP servers demand that decision support queries be answered in the order of seconds. Therefore, it is crucial for data warehouse systems to support highly efficient cube computation techniques, access methods, and query processing techniques. In this section, we present an overview of methods for the efficient implementation of data warehouse systems.

### 3.4.1 Efficient Computation of Data Cubes

At the core of multidimensional data analysis is the efficient computation of aggregations across many sets of dimensions. In SQL terms, these aggregations are referred to as **group-by**'s. Each group-by can be represented by a *cuboid*, where the set of group-by's forms a lattice of cuboids defining a data cube. In this section, we explore issues relating to the efficient computation of data cubes.

#### The **compute cube** Operator and the Curse of Dimensionality

One approach to cube computation extends SQL so as to include a **compute cube** operator. The **compute cube** operator computes aggregates over all subsets of the dimensions specified in the operation. This can require excessive storage space, especially for large numbers of dimensions. We start with an intuitive look at what is involved in the efficient computation of data cubes.

**Example 3.11 A data cube is a lattice of cuboids.** Suppose that you would like to create a data cube for *AllElectronics* sales that contains the following: *city*, *item*, *year*, and *sales\_in\_dollars*. You would like to be able to analyze the data, with queries such as the following:

- “Compute the sum of sales, grouping by city and item.”
- “Compute the sum of sales, grouping by city.”



- “Compute the sum of sales, grouping by item.”

What is the total number of cuboids, or group-by's, that can be computed for this data cube? Taking the three attributes, *city*, *item*, and *year*, as the dimensions for the data cube, and *sales\_in\_dollars* as the measure, the total number of cuboids, or group-by's, that can be computed for this data cube is  $2^3 = 8$ . The possible group-by's are the following:  $\{(city, item, year), (city, item), (city, year), (item, year), (city), (item), (year), ()\}$ , where  $()$  means that the group-by is empty (i.e., the dimensions are not grouped). These group-by's form a lattice of cuboids for the data cube, as shown in Figure 3.14. The **base cuboid** contains all three dimensions, *city*, *item*, and *year*. It can return the total sales for any combination of the three dimensions. The **apex cuboid**, or 0-D cuboid, refers to the case where the group-by is empty. It contains the total sum of all sales. The base cuboid is the least generalized (most specific) of the cuboids. The apex cuboid is the most generalized (least specific) of the cuboids, and is often denoted as *all*. If we start at the apex cuboid and explore downwards in the lattice, this is equivalent to drilling down within the data cube. If we start at the base cuboid and explore upwards, this is akin to rolling up. ■

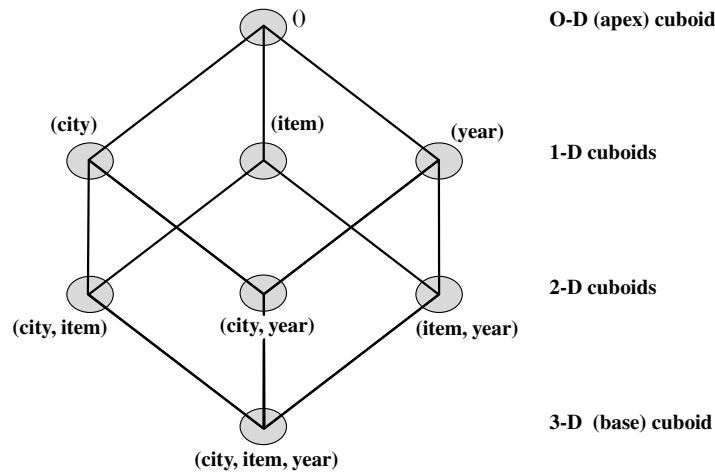


Figure 3.14: Lattice of cuboids, making up a 3-D data cube. Each cuboid represents a different group-by. The base cuboid contains the three dimensions *city*, *item*, and *year*. [TO EDITOR For consistency, please italicize *city*, *item*, and *year*.]

An SQL query containing no group-by, such as “compute the sum of total sales,” is a *zero-dimensional operation*. An SQL query containing one group-by, such as “compute the sum of sales, group by city,” is a *one-dimensional operation*. A cube operator on  $n$  dimensions is equivalent to a collection of **group by** statements, one for each subset of the  $n$  dimensions. Therefore, the cube operator is the  $n$ -dimensional generalization of the **group by** operator.

Based on the syntax of DMQL introduced in Section 3.2.3, the data cube in Example 3.11 could be defined as

```
define cube sales_cube [city, item, year]: sum(sales_in_dollars)
```

For a cube with  $n$  dimensions, there are a total of  $2^n$  cuboids, including the base cuboid. A statement such as

```
compute cube sales_cube
```

would explicitly instruct the system to compute the sales aggregate cuboids for all of the eight subsets of the set  $\{city, item, year\}$ , including the empty subset. A cube computation operator was first proposed and studied by Gray et al. [GCB<sup>+</sup>97].

On-line analytical processing may need to access different cuboids for different queries. Therefore, it may seem like a good idea to compute all or at least some of the cuboids in a data cube in advance. Precomputation leads

to fast response time and avoids some redundant computation. Most, if not all, OLAP products resort to some degree of precomputation of multidimensional aggregates.

A major challenge related to this precomputation, however, is that the required storage space may explode if all of the cuboids in a data cube are precomputed, especially when the cube has many dimensions. The storage requirements are even more excessive when many of the dimensions have associated concept hierarchies, each with multiple levels. This problem is referred to as the **curse of dimensionality**. The extent of the curse of dimensionality is illustrated below.

“How many cuboids are there in an  $n$ -dimensional data cube?” If there were no hierarchies associated with each dimension, then the total number of cuboids for an  $n$ -dimensional data cube, as we have seen above, is  $2^n$ . However, in practice, many dimensions do have hierarchies. For example, the dimension *time* is usually not explored at only one conceptual level, such as *year*, but rather at multiple conceptual levels, such as in the hierarchy “*day* < *month* < *quarter* < *year*”. For an  $n$ -dimensional data cube, the total number of cuboids that can be generated (including the cuboids generated by climbing up the hierarchies along each dimension) is

$$\text{Total number of cuboids} = \prod_{i=1}^n (L_i + 1), \quad (3.1)$$

where  $L_i$  is the number of levels associated with dimension  $i$ . One is added to  $L_i$  in Equation (3.1) to include the *virtual* top level, *all*. (Note that generalizing to *all* is equivalent to the removal of the dimension.) This formula is based on the fact that at most one abstraction level in each dimension will appear in a cuboid. For example, the time dimension as specified above has 4 conceptual levels, or 5 if we include the virtual level *all*. If the cube has 10 dimensions and each dimension has 5 levels (including *all*), the total number of cuboids that can be generated is  $5^{10} \approx 9.8 \times 10^6$ . The size of each cuboid also depends on the *cardinality* (i.e., number of distinct values) of each dimension. For example, if the *AllElectronics* branch in each city sold every item, there would be  $|city| \times |item|$  tuples in the *city-item* group-by alone. As the number of dimensions, number of conceptual hierarchies, or cardinality increases, the storage space required for many of the group-by’s will grossly exceed the (fixed) size of the input relation.

By now, you probably realize that it is unrealistic to precompute and materialize all of the cuboids that can possibly be generated for a data cube (or from a base cuboid). If there are many cuboids, and these cuboids are large in size, a more reasonable option is *partial materialization*, that is, to materialize only *some* of the possible cuboids that can be generated.

### Partial Materialization: Selected Computation of Cuboids

There are three choices for data cube materialization given a base cuboid:

1. **No materialization:** Do not precompute any of the “nonbase” cuboids. This leads to computing expensive multidimensional aggregates on the fly, which can be extremely slow.
2. **Full materialization:** Precompute all of the cuboids. The resulting lattice of computed cuboids is referred to as the *full cube*. This choice typically requires huge amounts of memory space in order to store all of the precomputed cuboids.
3. **Partial materialization:** Selectively compute a proper subset of the whole set of possible cuboids. Alternatively, we may compute a subset of the cube, which contains only those cells that satisfy some user-specified criterion, such as where the tuple count of each cell is above some threshold. We will use the term *subcube* to refer to the latter case, where only some of the cells may be precomputed for various cuboids. Partial materialization represents an interesting trade-off between storage space and response time.

The partial materialization of cuboids or subcubes should consider three factors: (1) identify the subset of cuboids or subcubes to materialize. (2) exploit the materialized cuboids or subcubes during query processing, and (3) efficiently update the materialized cuboids or subcubes during load and refresh.

The selection of the subset of cuboids or subcubes to materialize should take into account the queries in the workload, their frequencies, and their accessing costs. In addition, it should consider workload characteristics, the cost for incremental updates, and the total storage requirements. The selection must also consider the broad context of physical database design, such as the generation and selection of indices. Several OLAP products have adopted heuristic approaches for cuboid and subcube selection. A popular approach is to materialize the set of cuboids on which other frequently referenced cuboids are based. Alternatively, we can compute an *iceberg cube*, which is a data cube that stores only those cube cells whose aggregate value (e.g., *count*) is above some minimum support threshold. Another common strategy is to materialize a *shell cube*. This involves precomputing the cuboids for only a small number of dimensions (such as 3 to 5) of a data cube. Queries on additional combinations of the dimensions can be computed on the fly. Since our aim in this chapter is to provide a solid introduction and overview of data warehousing for data mining, we defer our detailed discussion of cuboid selection and computation to Chapter 4, which studies data warehouse and OLAP implementation in greater depth.

Once the selected cuboids have been materialized, it is important to take advantage of them during query processing. This involves several issues, such as how to determine the relevant cuboid(s) from among the candidate materialized cuboids, how to use available index structures on the materialized cuboids, and how to transform the OLAP operations onto the selected cuboid(s). These issues are discussed in Section 3.4.3 as well as in Chapter 4.

Finally, during load and refresh, the materialized cuboids should be updated efficiently. Parallelism and incremental update techniques for this should be explored.

### 3.4.2 Indexing OLAP Data

To facilitate efficient data accessing, most data warehouse systems support index structures and materialized views (using cuboids). General methods to select cuboids for materialization were discussed in the previous section. In this section, we examine how to index OLAP data by *bitmap indexing* and *join indexing*.

The **bitmap indexing** method is popular in OLAP products because it allows quick searching in data cubes. The bitmap index is an alternative representation of the *record.ID (RID)* list. In the bitmap index for a given attribute, there is a distinct bit vector,  $B_v$ , for each value  $v$  in the domain of the attribute. If the domain of a given attribute consists of  $n$  values, then  $n$  bits are needed for each entry in the bitmap index (i.e., there are  $n$  bit vectors). If the attribute has the value  $v$  for a given row in the data table, then the bit representing that value is set to 1 in the corresponding row of the bitmap index. All other bits for that row are set to 0.

**Example 3.12 Bitmap indexing.** In the *AllElectronics* data warehouse, suppose the dimension *item* at the top level has four values (representing item types): “*home entertainment*”, “*computer*”, “*phone*”, and “*security*”. Each value (e.g., “*computer*”) is represented by a bit vector in the bitmap index table for *item*. Suppose that the cube is stored as a relation table with 100,000 rows. Since the domain of *item* consists of four values, the bitmap index table requires four bit vectors (or lists), each with 100,000 bits. Figure 3.15 shows a base (data) table containing the dimensions *item* and *city*, and its mapping to bitmap index tables for each of the dimensions. ■

Bitmap indexing is advantageous compared to hash and tree indices. It is especially useful for low-cardinality domains because comparison, join, and aggregation operations are then reduced to bit arithmetic, which substantially reduces the processing time. Bitmap indexing leads to significant reductions in space and I/O since a string of characters can be represented by a single bit. For higher-cardinality domains, the method can be adapted using compression techniques.

The **join indexing** method gained popularity from its use in relational database query processing. Traditional indexing maps the value in a given column to a list of rows having that value. In contrast, join indexing registers the joinable rows of two relations from a relational database. For example, if two relations  $R(RID, A)$  and  $S(B, SID)$  join on the attributes  $A$  and  $B$ , then the join index record contains the pair  $(RID, SID)$ , where  $RID$  and  $SID$  are record identifiers from the  $R$  and  $S$  relations, respectively. Hence, the join index records can identify joinable tuples without performing costly join operations. Join indexing is especially useful for maintaining the relationship

Base table			Item bitmap index table					City bitmap index table		
RID	item	city	RID	H	C	P	S	RID	V	T
R1	H	V	R1	1	0	0	0	R1	1	0
R2	C	V	R2	0	1	0	0	R2	1	0
R3	P	V	R3	0	0	1	0	R3	1	0
R4	S	V	R4	0	0	0	1	R4	1	0
R5	H	T	R5	1	0	0	0	R5	0	1
R6	C	T	R6	0	1	0	0	R6	0	1
R7	P	T	R7	0	0	1	0	R7	0	1
R8	S	T	R8	0	0	0	1	R8	0	1

Note: H for “ home entertainment;”C for “ computer;”P for “ phone;”S for “ security;”  
V for “ Vancouver;”T for “ Toronto.”

Figure 3.15: Indexing OLAP data using bitmap indices. [TO EDITOR For consistency with text, please italicize *item* and *city*.]

between a foreign key<sup>3</sup> and its matching primary keys, from the joinable relation.

The star schema model of data warehouses makes join indexing attractive for cross table search because the linkage between a fact table and its corresponding dimension tables are the foreign key of the fact table and the primary key of the dimension table. Join indexing maintains relationships between attribute values of a dimension (e.g., within a dimension table) and the corresponding rows in the fact table. Join indices may span multiple dimensions to form **composite join indices**. We can use join indices to identify subcubes that are of interest.

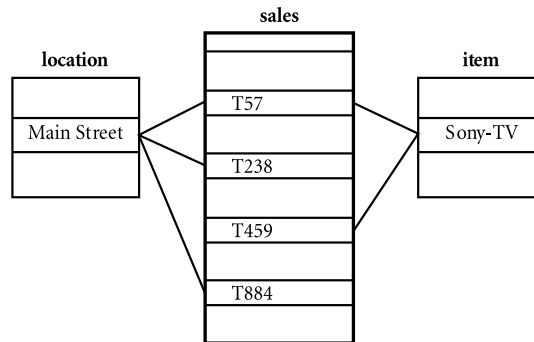


Figure 3.16: Linkages between a *sales* fact table and dimension tables for *location* and *item*. [TO EDITOR For consistency with text, please italicize *location*, *sales*, and *item*.]

**Example 3.13 Join indexing.** In Example 3.4, we defined a star schema for *AllElectronics* of the form “*sales\_star* [*time*, *item*, *branch*, *location*]: *dollars\_sold* = *sum (sales\_in\_dollars)*”. An example of a join index relationship between the *sales* fact table and the dimension tables for *location* and *item* is shown in Figure 3.16. For example, the “*Main Street*” value in the *location* dimension table joins with tuples T57, T238, and T884 of the *sales* fact table. Similarly, the “*Sony-TV*” value in the *item* dimension table joins with tuples T57 and T459 of the *sales* fact table. The corresponding join index tables are shown in Figure 3.17.

Suppose that there are 360 time values, 100 items, 50 branches, 30 locations, and 10 million sales tuples in the *sales\_star* data cube. If the *sales* fact table has recorded sales for only 30 items, the remaining 70 items will obviously not participate in joins. If join indices are not used, additional I/Os have to be performed to bring the joining portions of the fact table and dimension tables together. ■

To further speed up query processing, the join indexing and bitmap indexing methods can be integrated to

<sup>3</sup>A set of attributes in a relation schema that forms a primary key for another relation schema is called a **foreign key**.

Join index table for location/sales		Join index table for item/sales	
location	sales_key	item	sales_key
...	...	...	...
Main Street	T57	Sony-TV	T57
Main Street	T238	Sony-TV	T459
Main Street	T884	...	...
...	...		

Join index table linking two dimensions location/item/sales		
location	item	sales_key
...	...	...
Main Street	Sony-TV	T57
...	...	...

Figure 3.17: Join index tables based on the linkages between the *sales* fact table and dimension tables for *location* and *item* shown in Figure 3.16. [TO EDITOR For consistency with text, please italicize all instances of *location*, *sales*, *sales\_key*, and *item*.]

form **bitmapped join indices**.

### 3.4.3 Efficient Processing of OLAP Queries

The purpose of materializing cuboids and constructing OLAP index structures is to speed up query processing in data cubes. Given materialized views, query processing should proceed as follows:

1. **Determine which operations should be performed on the available cuboids:** This involves transforming any selection, projection, roll-up (group-by), and drill-down operations specified in the query into corresponding SQL and/or OLAP operations. For example, slicing and dicing of a data cube may correspond to selection and/or projection operations on a materialized cuboid.
2. **Determine to which materialized cuboid(s) the relevant operations should be applied:** This involves identifying all of the materialized cuboids that may potentially be used to answer the query, pruning the above set using knowledge of “dominance” relationships among the cuboids, estimating the costs of using the remaining materialized cuboids, and selecting the cuboid with the least cost.

**Example 3.14 OLAP query processing.** Suppose that we define a data cube for *AllElectronics* of the form “*sales\_cube* [*time*, *item*, *location*]:  $\text{sum}(\text{sales\_in\_dollars})$ ”. The dimension hierarchies used are “*day* < *month* < *quarter* < *year*” for *time*, “*item\_name* < *brand* < *type*” for *item*, and “*street* < *city* < *province\_or\_state* < *country*” for *location*.

Suppose that the query to be processed is on {*brand*, *province\_or\_state*}, with the selection constant “*year* = 2004”. Also, suppose that there are four materialized cuboids available, as follows:

- cuboid 1: {*year*, *item\_name*, *city*}
- cuboid 2: {*year*, *brand*, *country*}
- cuboid 3: {*year*, *brand*, *province\_or\_state*}
- cuboid 4: {*item\_name*, *province\_or\_state*} where *year* = 2004

“Which of the above four cuboids should be selected to process the query?” Finer-granularity data cannot be generated from coarser-granularity data. Therefore, cuboid 2 cannot be used since *country* is a more general concept than *province\_or\_state*. Cuboids 1, 3, and 4 can be used to process the query since (1) they have the same set or a superset of the dimensions in the query, (2) the selection clause in the query can imply the selection in the cuboid, and (3) the abstraction levels for the *item* and *location* dimensions in these cuboids are at a finer level than *brand* and *province\_or\_state*, respectively.

“How would the costs of each cuboid compare if used to process the query?” It is likely that using cuboid 1 would cost the most since both *item\_name* and *city* are at a lower level than the *brand* and *province\_or\_state* concepts specified in the query. If there are not many *year* values associated with *items* in the cube, but there are several *item\_names* for each *brand*, then cuboid 3 will be smaller than cuboid 4, and thus cuboid 3 should be chosen to process the query. However, if efficient indices are available for cuboid 4, then cuboid 4 may be a better choice. Therefore, some cost-based estimation is required in order to decide which set of cuboids should be selected for query processing. ■

Since the storage model of a MOLAP server is an  $n$ -dimensional array, the front-end multidimensional queries are mapped directly to server storage structures, which provide direct addressing capabilities. The straightforward array representation of the data cube has good indexing properties, but has poor storage utilization when the data are sparse. For efficient storage and processing, sparse matrix and data compression techniques should therefore be applied. The details of several such methods of cube computation are presented in Chapter 4.

The storage structures used by dense and sparse arrays may differ, making it advantageous to adopt a two-level approach to MOLAP query processing: use array structures for dense arrays, and sparse matrix structures for sparse arrays. The two-dimensional dense arrays can be indexed by B-trees.

To process a query in MOLAP, the dense one- and two-dimensional arrays must first be identified. Indices are then built to these arrays using traditional indexing structures. The two-level approach increases storage utilization without sacrificing direct addressing capabilities.

“Are there any other strategies for answering queries quickly?” Some strategies for answering queries quickly concentrate on providing *intermediate feedback* to the users. For example, in **on-line aggregation**, a data mining system can display “what it knows so far” instead of waiting until the query is fully processed. Such an approximate answer to the given data mining query is periodically refreshed and refined as the computation process continues. Confidence intervals are associated with each estimate, providing the user with additional feedback regarding the reliability of the answer so far. This promotes interactivity with the system—the user gains insight as to whether or not she is probing in the “right” direction without having to wait until the end of the query. While on-line aggregation does not improve the total time to answer a query, the overall data mining process should be quicker due to the increased interactivity with the system.

Another approach is to employ **top  $N$  queries**. Suppose that you are interested in finding only the best-selling items among the millions of items sold at *AllElectronics*. Rather than waiting to obtain a list of all store items, sorted in decreasing order of sales, you would like to see only the top  $N$ . Using statistics, query processing can be optimized to return the top  $N$  items, rather than the whole sorted list. This results in faster response time while helping to promote user interactivity and reduce wasted resources.

The goal of this section was to provide an overview of data warehouse implementation. Chapter 4 presents a more advanced treatment of this topic. It examines the efficient computation of data cubes and processing of OLAP queries in greater depth, providing detailed algorithms.

### 3.5 From Data Warehousing to Data Mining

“How do data warehousing and OLAP relate to data mining?” In this section, we study the usage of data warehousing for information processing, analytical processing, and data mining. We also introduce on-line analytical mining (OLAM), a powerful paradigm that integrates OLAP with data mining technology.

### 3.5.1 Data Warehouse Usage

Data warehouses and data marts are used in a wide range of applications. Business executives use the data in data warehouses and data marts to perform data analysis and make strategic decisions. In many firms, data warehouses are used as an integral part of a *plan-execute-assess* “closed-loop” feedback system for enterprise management. Data warehouses are used extensively in banking and financial services, consumer goods and retail distribution sectors, and controlled manufacturing, such as demand-based production.

Typically, the longer a data warehouse has been in use, the more it will have evolved. This evolution takes place throughout a number of phases. Initially, the data warehouse is mainly used for generating reports and answering predefined queries. Progressively, it is used to analyze summarized and detailed data, where the results are presented in the form of reports and charts. Later, the data warehouse is used for strategic purposes, performing multidimensional analysis and sophisticated slice-and-dice operations. Finally, the data warehouse may be employed for knowledge discovery and strategic decision making using data mining tools. In this context, the tools for data warehousing can be categorized into *access and retrieval tools*, *database reporting tools*, *data analysis tools*, and *data mining tools*.

Business users need to have the means to know what exists in the data warehouse (through metadata), how to access the contents of the data warehouse, how to examine the contents using analysis tools, and how to present the results of such analysis.

There are three kinds of data warehouse applications: *information processing*, *analytical processing*, and *data mining*:

- **Information processing** supports querying, basic statistical analysis, and reporting using crosstabs, tables, charts, or graphs. A current trend in data warehouse information processing is to construct low-cost Web-based accessing tools that are then integrated with Web browsers.
- **Analytical processing** supports basic OLAP operations, including slice-and-dice, drill-down, roll-up, and pivoting. It generally operates on historical data in both summarized and detailed forms. The major strength of on-line analytical processing over information processing is the multidimensional data analysis of data warehouse data.
- **Data mining** supports knowledge discovery by finding hidden patterns and associations, constructing analytical models, performing classification and prediction, and presenting the mining results using visualization tools.

“*How does data mining relate to information processing and on-line analytical processing?*” Information processing, based on queries, can find useful information. However, answers to such queries reflect the information directly stored in databases or computable by aggregate functions. They do not reflect sophisticated patterns or regularities buried in the database. Therefore, information processing is not data mining.

On-line analytical processing comes a step closer to data mining since it can derive information summarized at multiple granularities from user-specified subsets of a data warehouse. Such descriptions are equivalent to the class/concept descriptions discussed in Chapter 1. Since data mining systems can also mine generalized class/concept descriptions, this raises some interesting questions: “*Do OLAP systems perform data mining? Are OLAP systems actually data mining systems?*”

The functionalities of OLAP and data mining can be viewed as disjoint: OLAP is a data summarization/aggregation *tool* that helps simplify data analysis, while data mining allows the *automated discovery* of implicit patterns and interesting knowledge hidden in large amounts of data. OLAP tools are targeted toward simplifying and supporting interactive data analysis, whereas the goal of data mining tools is to automate as much of the process as possible, while still allowing users to guide the process. In this sense, data mining goes one step beyond traditional on-line analytical processing.

An alternative and broader view of data mining may be adopted in which data mining covers both data description and data modeling. Since OLAP systems can present general descriptions of data from data warehouses,

OLAP functions are essentially for user-directed data summary and comparison (by drilling, pivoting, slicing, dicing, and other operations). These are, though limited, data mining functionalities. Yet according to this view, data mining covers a much broader spectrum than simple OLAP operations because it not only performs data summary and comparison, but also performs association, classification, prediction, clustering, time-series analysis, and other data analysis tasks.

Data mining is not confined to the analysis of data stored in data warehouses. It may analyze data existing at more detailed granularities than the summarized data provided in a data warehouse. It may also analyze transactional, spatial, textual, and multimedia data that are difficult to model with current multidimensional database technology. In this context, data mining covers a broader spectrum than OLAP with respect to data mining functionality and the complexity of the data handled.

Since data mining involves more automated and deeper analysis than OLAP, data mining is expected to have broader applications. Data mining can help business managers find and reach more suitable customers, as well as gain critical business insights that may help to drive market share and raise profits. In addition, data mining can help managers understand customer group characteristics and develop optimal pricing strategies accordingly, correct item bundling based not on intuition but on actual item groups derived from customer purchase patterns, reduce promotional spending, and at the same time increase the overall net effectiveness of promotions.

### 3.5.2 From On-Line Analytical Processing to On-Line Analytical Mining

In the field of data mining, substantial research has been performed for data mining at various platforms, including transaction databases, relational databases, spatial databases, text databases, time-series databases, flat files, data warehouses, and so on.

Among many different paradigms and architectures of data mining systems, **on-line analytical mining (OLAM)** (also called **OLAP mining**), which integrates on-line analytical processing (OLAP) with data mining and mining knowledge in multidimensional databases, is particularly important for the following reasons:

- **High quality of data in data warehouses:** Most data mining tools need to work on integrated, consistent, and cleaned data, which requires costly data cleaning, data transformation, and data integration as preprocessing steps. A data warehouse constructed by such preprocessing serves as a valuable source of high-quality data for OLAP as well as for data mining. Notice that data mining may also serve as a valuable tool for data cleaning and data integration as well.
- **Available information processing infrastructure surrounding data warehouses:** Comprehensive information processing and data analysis infrastructures have been or will be systematically constructed surrounding data warehouses, which include accessing, integration, consolidation, and transformation of multiple heterogeneous databases, ODBC/OLE DB connections, Web-accessing and service facilities, and reporting and OLAP analysis tools. It is prudent to make the best use of the available infrastructures rather than constructing everything from scratch.
- **OLAP-based exploratory data analysis:** Effective data mining needs exploratory data analysis. A user will often want to traverse through a database, select portions of relevant data, analyze them at different granularities, and present knowledge/results in different forms. On-line analytical mining provides facilities for data mining on different subsets of data and at different levels of abstraction, by drilling, pivoting, filtering, dicing and slicing on a data cube and on some intermediate data mining results. This, together with data/knowledge visualization tools, will greatly enhance the power and flexibility of exploratory data mining.
- **On-line selection of data mining functions:** Often a user may not know what kinds of knowledge she would like to mine. By integrating OLAP with multiple data mining functions, on-line analytical mining provides users with the flexibility to select desired data mining functions and swap data mining tasks dynamically.



### Architecture for On-Line Analytical Mining

An OLAM server performs analytical mining in data cubes in a similar manner as an OLAP server performs on-line analytical processing. An integrated OLAM and OLAP architecture is shown in Figure 3.18, where the OLAM and OLAP servers both accept user on-line queries (or commands) via a graphical user interface API and work with the data cube in the data analysis via a cube API. A metadata directory is used to guide the access of the data cube. The data cube can be constructed by accessing and/or integrating multiple databases via an MDDB API and/or by filtering a data warehouse via a database API that may support OLE DB or ODBC connections. Since an OLAM server may perform multiple data mining tasks, such as concept description, association, classification, prediction, clustering, time-series analysis, and so on, it usually consists of multiple integrated data mining modules and is more sophisticated than an OLAP server.

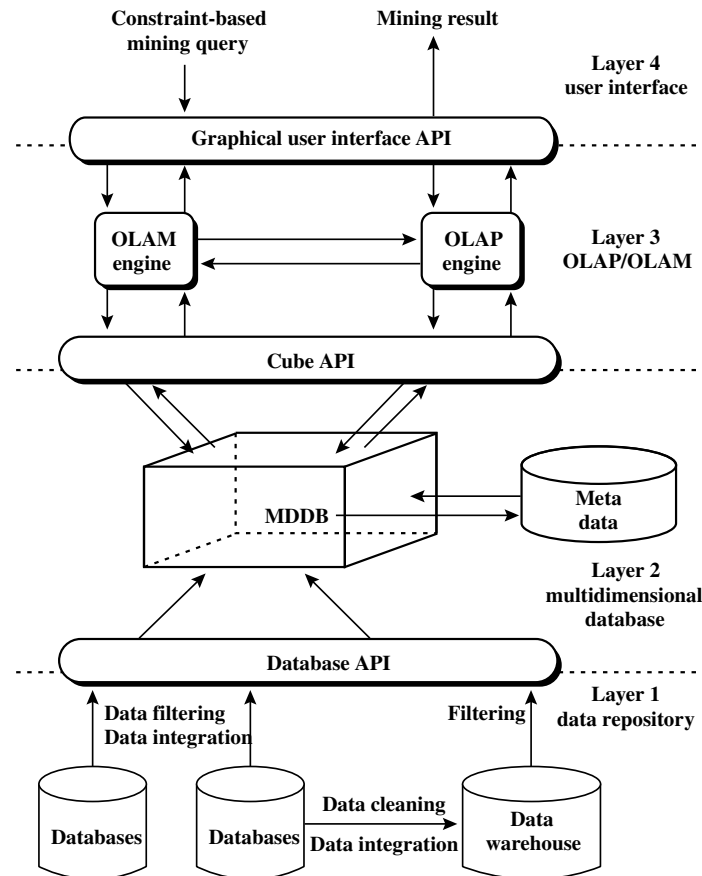


Figure 3.18: An integrated OLAM and OLAP architecture.

Chapter 4 describes data warehouses on a finer level by exploring implementation issues such as data cube computation, OLAP query answering strategies, and methods of generalization. The chapters following it are devoted to the study of data mining techniques. As we have seen, the introduction to data warehousing and OLAP technology presented in this chapter is essential to our study of data mining. This is because data warehousing provides users with large amounts of clean, organized, and summarized data, which greatly facilitates data mining. For example, rather than storing the details of each sales transaction, a data warehouse may store a summary of the transactions per item type for each branch or, summarized to a higher level, for each country. The capability of OLAP to provide multiple and dynamic views of summarized data in a data warehouse sets a solid foundation for successful data mining.

Moreover, we also believe that data mining should be a human-centered process. Rather than asking a data

mining system to generate patterns and knowledge automatically, a user will often need to interact with the system to perform exploratory data analysis. OLAP sets a good example for interactive data analysis and provides the necessary preparations for exploratory data mining. Consider the discovery of association patterns, for example. Instead of mining associations at a primitive (i.e., low) data level among transactions, users should be allowed to specify roll-up operations along any dimension. For example, a user may like to roll up on the *item* dimension to go from viewing the data for particular TV sets that were purchased to viewing the brands of these TVs, such as SONY or Panasonic. Users may also navigate from the transaction level to the customer level or customer-type level in the search for interesting associations. Such an OLAP-style of data mining is characteristic of OLAP mining. In our study of the principles of data mining in this book, we place particular emphasis on OLAP mining, that is, on the *integration of data mining and OLAP technology*.

### 3.6 Summary

- A **data warehouse** is a *subject-oriented, integrated, time-variant, and nonvolatile* collection of data organized in support of management decision making. Several factors distinguish data warehouses from operational databases. Since the two systems provide quite different functionalities and require different kinds of data, it is necessary to maintain data warehouses separately from operational databases.
- A **multidimensional data model** is typically used for the design of corporate *data warehouses* and *departmental data marts*. Such a model can adopt a *star schema*, *snowflake schema*, or *fact constellation schema*. The core of the *multidimensional model* is the **data cube**, which consists of a large set of *facts* (or *measures*) and a number of *dimensions*. Dimensions are the entities or perspectives with respect to which an organization wants to keep records and are hierarchical in nature.
- A data cube consists of a **lattice of cuboids**, each corresponding to a different degree of summarization of the given multidimensional data.
- **Concept hierarchies** organize the values of attributes or dimensions into gradual levels of abstraction. They are useful in mining at multiple levels of abstraction.
- **On-line analytical processing (OLAP)** can be performed in data warehouses/marts using the multidimensional data model. Typical OLAP operations include *roll-up*, *drill-(down, across, through)*, *slice-and-dice*, *pivot (rotate)*, as well as statistical operations such as ranking, computing moving averages and growth rates, and so on. OLAP operations can be implemented efficiently using the data cube structure.
- Data warehouses often adopt a **three-tier architecture**. The bottom tier is a *warehouse database server*, which is typically a relational database system. The middle tier is an *OLAP server*, and the top tier is a *client*, containing query and reporting tools.
- A data warehouse contains **back-end tools and utilities** for populating and refreshing the warehouse. These cover data extraction, data cleaning, data transformation, loading, refreshing, and warehouse management.
- Data warehouse **metadata** are data defining the warehouse objects. A metadata repository provides details regarding the warehouse structure, data history, the algorithms used for summarization, mappings from the source data to warehouse form, system performance, and business terms and issues.
- OLAP servers may use **relational OLAP (ROLAP)**, or **multidimensional OLAP (MOLAP)**, or **hybrid OLAP (HOLAP)**. A ROLAP server uses an extended relational DBMS that maps OLAP operations on multidimensional data to standard relational operations. A MOLAP server maps multidimensional data views directly to array structures. A HOLAP server combines ROLAP and MOLAP. For example, it may use ROLAP for historical data while maintaining frequently accessed data in a separate MOLAP store.
- **Full materialization** refers to the computation of all of the cuboids in the lattice defining a data cube. It typically requires an excessive amount of storage space, particularly as the number of dimensions and size of associated concept hierarchies grow. This problem is known as the **curse of dimensionality**. Alternatively,

**partial materialization** is the selective computation of a subset of the cuboids or subcubes in the lattice. For example, an **iceberg cube** is a data cube that stores only those cube cells whose aggregate value (e.g., count) is above some minimum support threshold.

- OLAP query processing can be made more efficient with the use of indexing techniques. In **bitmap indexing**, each attribute has its own bitmap index table. Bitmap indexing reduces join, aggregation, and comparison operations to bit arithmetic. **Join indexing** registers the joinable rows of two or more relations from a relational database, reducing the overall cost of OLAP join operations. **Bitmapped join indexing**, which combines the bitmap and join index methods, can be used to further speed up OLAP query processing.
- Data warehouses are used for *information processing* (querying and reporting), *analytical processing* (which allows users to navigate through summarized and detailed data by OLAP operations), and *data mining* (which supports knowledge discovery). OLAP-based data mining is referred to as **OLAP mining**, or on-line analytical mining (**OLAM**), which emphasizes the interactive and exploratory nature of OLAP mining.

### 3.7 Exercises

1. State why, for the integration of multiple heterogeneous information sources, many companies in industry prefer the *update-driven approach* (which constructs and uses data warehouses), rather than the *query-driven approach* (which applies wrappers and integrators). Describe situations where the query-driven approach is preferable over the update-driven approach.
2. Briefly compare the following concepts. You may use an example to explain your point(s).
  - (a) Snowflake schema, fact constellation, starnet query model
  - (b) Data cleaning, data transformation, refresh
  - (c) Discovery-driven cube, multifeature cube, virtual warehouse
3. Suppose that a data warehouse consists of the three dimensions *time*, *doctor*, and *patient*, and the two measures *count* and *charge*, where *charge* is the fee that a doctor charges a patient for a visit.
  - (a) Enumerate three classes of schemas that are popularly used for modeling data warehouses.
  - (b) Draw a schema diagram for the above data warehouse using one of the schema classes listed in (a).
  - (c) Starting with the base cuboid [*day, doctor, patient*], what specific *OLAP operations* should be performed in order to list the total fee collected by each doctor in 2004?
  - (d) To obtain the same list, write an SQL query assuming the data is stored in a relational database with the schema *fee* (*day, month, year, doctor, hospital, patient, count, charge*).
4. Suppose that a data warehouse for *Big-University* consists of the following four dimensions: *student*, *course*, *semester*, and *instructor*, and two measures *count* and *avg-grade*. When at the lowest conceptual level (e.g., for a given student, course, semester, and instructor combination), the *avg-grade* measure stores the actual course grade of the student. At higher conceptual levels, *avg-grade* stores the average grade for the given combination.
  - (a) Draw a *snowflake schema* diagram for the data warehouse.
  - (b) Starting with the base cuboid [*student, course, semester, instructor*], what specific *OLAP operations* (e.g., roll-up from *semester* to *year*) should one perform in order to list the average grade of *CS* courses for each *Big-University* student.
  - (c) If each dimension has five levels (including *all*), such as “*student < major < status < university < all*”, how many cuboids will this cube contain (including the base and apex cuboids)?

5. Suppose that a data warehouse consists of the four dimensions, *date*, *spectator*, *location*, and *game*, and the two measures, *count* and *charge*, where *charge* is the fare that a spectator pays when watching a game on a given date. Spectators may be students, adults, or seniors, with each category having its own charge rate.
  - (a) Draw a *star schema* diagram for the data warehouse.
  - (b) Starting with the base cuboid [*date*, *spectator*, *location*, *game*], what specific *OLAP operations* should one perform in order to list the total charge paid by student spectators at GM Place in 2004?
  - (c) *Bitmap indexing* is useful in data warehousing. Taking this cube as an example, briefly discuss advantages and problems of using a bitmap index structure.
6. A data warehouse can be modeled by either a *star schema* or a *snowflake schema*. Briefly describe the similarities and the differences of the two models, and then analyze their advantages and disadvantages with regard to one another. Give your opinion of which might be more empirically useful and state the reasons behind your answer.
7. Design a data warehouse for a regional weather bureau. The weather bureau has about 1,000 probes, which are scattered throughout various land and ocean locations in the region to collect basic weather data, including air pressure, temperature, and precipitation at each hour. All data are sent to the central station, which has collected such data for over 10 years. Your design should facilitate efficient querying and on-line analytical processing, and derive general weather patterns in multidimensional space.
8. A popular data warehouse implementation is to construct a multidimensional database, known as a data cube. Unfortunately, this may often generate a huge, yet very sparse multidimensional matrix.
  - (a) Present an example illustrating such a huge and sparse data cube.
  - (b) Design an implementation method that can elegantly overcome this sparse matrix problem. Note that you need to explain your data structures in detail and discuss the space needed, as well as how to retrieve data from your structures.
  - (c) Modify your design in (b) to handle *incremental data updates*. Give the reasoning behind your new design.
9. Regarding the *computation of measures* in a data cube:
  - (a) Enumerate three categories of measures, based on the kind of aggregate functions used in computing a data cube.
  - (b) For a data cube with the three dimensions *time*, *location*, and *item*, which category does the function *variance* belong to? Describe how to compute it if the cube is partitioned into many chunks.  
Hint: The formula for computing *variance* is  $\frac{1}{N} \sum_{i=1}^N (x_i - \bar{x}_i)^2$ , where  $\bar{x}_i$  is the average of  $x_i$ s.
  - (c) Suppose the function is “*top 10 sales*.” Discuss how to efficiently compute this measure in a data cube.
10. Suppose that we need to record three measures in a data cube: *min*, *average*, and *median*. Design an efficient computation and storage method for each measure given that the cube allows data to be *deleted incrementally* (i.e., in small portions at a time) from the cube.
11. In data warehouse technology, a multiple dimensional view can be implemented by a relational database technique (*ROLAP*), or by a multidimensional database technique (*MOLAP*), or by a hybrid database technique (*HOLAP*).
  - (a) Briefly describe each implementation technique.
  - (b) For each technique, explain how each of the following functions may be implemented:
    - i. The generation of a data warehouse (including aggregation)
    - ii. Roll-up
    - iii. Drill-down
    - iv. Incremental updating

Which implementation techniques do you prefer, and why?

12. Suppose that a data warehouse contains 20 dimensions, each with about five levels of granularity.
  - (a) Users are mainly interested in four particular dimensions, each having three frequently accessed levels for rolling up and drilling down. How would you design a data cube structure to support this preference efficiently?
  - (b) At times, a user may want to *drill through* the cube, down to the raw data for one or two particular dimensions. How would you support this feature?
13. A data cube,  $C$ , has  $n$  dimensions, and each dimension has exactly  $p$  distinct values in the base cuboid. Assume that there are no concept hierarchies associated with the dimensions.
  - (a) What is the *maximum number of cells* possible in the base cuboid?
  - (b) What is the *minimum number of cells* possible in the base cuboid?
  - (c) What is the *maximum number of cells* possible (including both base cells and aggregate cells) in the data cube,  $C$ ?
  - (d) What is the *minimum number of cells* possible in the data cube,  $C$ ?
14. What are the differences between the three main types of data warehouse usage: *information processing*, *analytical processing*, and *data mining*? Discuss the motivation behind *OLAP mining (OLAM)*.

## 3.8 Bibliographic Notes

There are a good number of introductory-level textbooks on data warehousing and OLAP technology, including Kimball and Ross [KR02], Imhoff, Gallemmo and Geiger [IGG03], Inmon [Inm96], Berson and Smith [BS97b], and Thomsen [Tho97]. Chaudhuri and Dayal [CD97] provide a general overview of data warehousing and OLAP technology. A set of research papers on materialized views and data warehouse implementations were collected in *Materialized Views: Techniques, Implementations, and Applications* by Gupta and Mumick [GM99].

The history of decision support systems can be traced back to the 1960s. However, the proposal of the construction of large data warehouses for multidimensional data analysis is credited to Codd [CCS93] who coined the term *OLAP* for *on-line analytical processing*. The OLAP council was established in 1995. Widom [Wid95] identified several research problems in data warehousing. Kimball and Ross [KR02] provide an overview of the deficiencies of SQL regarding the ability to support comparisons that are common in the business world and present a good set of application cases that require data warehousing and OLAP technology. For an overview of OLAP systems versus statistical databases, see Shoshani [Sho97].

Gray et al. [GCB<sup>+</sup>97] proposed the data cube as a relational aggregation operator generalizing group-by, crosstabs, and subtotals. Harinarayan, Rajaraman, and Ullman [HRU96] proposed a greedy algorithm for the partial materialization of cuboids in the computation of a data cube. Sarawagi and Stonebraker [SS94] developed a chunk-based computation technique for the efficient organization of large multidimensional arrays. Agarwal et al. [AAD<sup>+</sup>96] proposed several methods for the efficient computation of multidimensional aggregates for ROLAP servers. A chunk-based multiway array aggregation method for data cube computation in MOLAP was proposed in Zhao, Deshpande, and Naughton [ZDN97]. Ross and Srivastava [RS97] pointed out the problem of the curse of dimensionality in cube materialization and developed a method for computing sparse data cubes. Iceberg queries are first described in Fang, Shivakumar, Garcia-Molina, et al. [FSGM<sup>+</sup>98]. BUC, an efficient bottom-up method for computing iceberg cubes was introduced by Beyer and Ramakrishnan [BR99]. References for the further development of cube computation methods are given in the Bibliographic Notes of Chapter 4. The use of join indices to speed up relational query processing was proposed by Valduriez [Val87]. O'Neil and Graefe [OG95] proposed a bitmapped join index method to speed up OLAP-based query processing. A discussion of the performance of bitmapping and other nontraditional index techniques is given in O'Neil and Quass [OQ97].

For work regarding the selection of materialized cuboids for efficient OLAP query processing, see Chaudhuri and Dayal [CD97], Harinarayan, Rajaraman, and Ullman [HRU96], Sristava et al. [SDJL96]. Methods for cube size estimation can be found in Deshpande et al. [DNR<sup>+</sup>97], Ross and Srivastava [RS97], and Beyer and Ramakrishnan [BR99]. Agrawal, Gupta, and Sarawagi [AGS97] proposed operations for modeling multidimensional databases. Methods for answering queries quickly by on-line aggregation are described in Hellerstein, Haas, and Wang [HHW97] and Hellerstein et al. [HAC<sup>+</sup>99]. Techniques for estimating the top  $N$  queries are proposed in Carey and Kossman [CK98] and Donjerkovic and Ramakrishnan [DR99]. Further studies on intelligent OLAP and discovery-driven exploration of data cubes are presented in the Bibliographic Notes of Chapter 4.

## Chapter 4

# Data Cube Computation and Data Generalization

**Data generalization** is a process that abstracts a large set of task-relevant data in a database from a relatively low conceptual level to higher conceptual levels. Users like the ease and flexibility of having large data sets summarized in concise and succinct terms, at different levels of granularity, and from different angles. Such data descriptions help provide an overall picture of the data at hand.

Data warehousing and OLAP perform data generalization by summarizing data at varying levels of abstraction. An overview of such technology was presented in Chapter 3. From a data analysis point of view, data generalization is a form of *descriptive data mining*, which describes data in a concise and summarative manner and presents interesting general properties of the data. In this chapter, we look at descriptive data mining in greater detail. Descriptive data mining differs from *predictive data mining*, which analyzes data in order to construct one or a set of models, and attempts to predict the behavior of new data sets. Predictive data mining, such as classification, regression analysis, and trend analysis, is covered in later chapters.

This chapter is organized into three main sections. The first two sections expand on notions of data warehouse and OLAP implementation presented in the previous chapter, while the third presents an alternative method for data generalization. In particular, Section 4.1 looks at how to efficiently compute data cubes at varying levels of abstraction. It presents an in-depth look at specific methods for data cube computation. Section 4.2 presents methods for further exploration of OLAP and data cubes. This includes discovery-driven exploration of data cubes, analysis of cubes with sophisticated features, and cube gradient analysis. Finally, Section 4.3 presents another method of data generalization, known as *attribute-oriented induction*.

### 4.1 Efficient Methods for Data Cube Computation

Data cube computation is an essential task in data warehouse implementation. The precomputation of all or part of a data cube can greatly reduce the response time and enhance the performance of on-line analytical processing. However, such computation is challenging since it may require substantial computational time and storage space. This section explores efficient methods for data cube computation. Section 4.1.1 introduces general concepts and computation strategies relating to cube materialization. Sections 4.1.2 to 4.1.5 detail specific computation algorithms, namely, MultiWay array aggregation, BUC, Star-Cubing, the computation of shell fragments, and the computation of cubes involving complex measures.

### 4.1.1 A Road Map for Materialization of Different Kinds of Cubes

Data cubes facilitate the on-line analytical processing of multidimensional data. “But how can we compute data cubes in advance, so that they are handy and readily available for query processing?” This section contrasts full cube materialization (i.e., precomputation) versus various strategies for partial cube materialization. For completeness, we begin with a review of the basic terminology involving data cubes. We also introduce a cube cell notation that is useful for describing data cube computation methods.

#### Cube Materialization: Full Cube, Iceberg Cube, Closed Cube, and Shell Cube

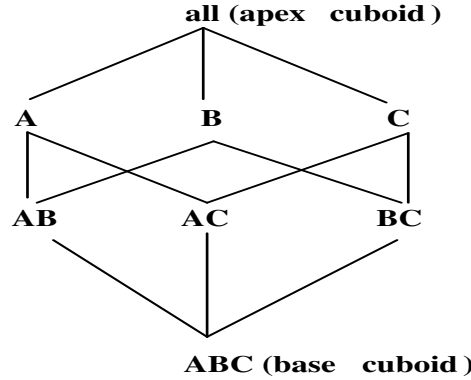


Figure 4.1: Lattice of cuboids, making up a 3-D data cube with the dimensions  $A$ ,  $B$ , and  $C$ .

Figure 4.1 shows a 3-D data cube for the dimensions  $A$ ,  $B$ , and  $C$ , and an aggregate measure,  $M$ . A data cube is a lattice of cuboids. Each cuboid represents a group-by.  $ABC$  is the base cuboid, containing all three of the dimensions. Here, the aggregate measure,  $M$ , is computed for each possible combination of the three dimensions. The base cuboid is the least generalized of all of the cuboids in the data cube. The most generalized cuboid is the apex cuboid, commonly represented as **all**. It contains one value—it aggregates measure  $M$  for all of the tuples stored in the base cuboid. To drill down in the data cube, we move from the apex cuboid, downwards in the lattice. To roll up, we move from the base cuboid, upwards. For the purposes of our discussion in this chapter, we will always use the term data cube to refer to a lattice of cuboids rather than an individual cuboid.

A cell in the base cuboid is a **base cell**. A cell from a non-base cuboid is an **aggregate cell**. An aggregate cell aggregates over one or more dimensions, where each aggregated dimension is indicated by a “\*” in the cell notation. Suppose we have an  $n$ -dimensional data cube. Let  $\mathbf{a} = (a_1, a_2, \dots, a_n, \text{measures}_{\mathbf{a}})$ <sup>1</sup> be a cell from one of the cuboids making up the data cube. We say that  $\mathbf{a}$  is an  **$m$ -dimensional cell** (that is, from an  $m$ -dimensional cuboid) if exactly  $m$  ( $m \leq n$ ) values among  $\{a_1, a_2, \dots, a_n\}$  are not “\*”. If  $m = n$ , then  $\mathbf{a}$  is a base cell; otherwise, it is an aggregate cell (i.e., where  $m \leq n$ ).

**Example 4.1 Base and aggregate cells.** Consider a data cube with the dimensions *month*, *city*, and *customer-group*, and the measure *price*.  $(Jan, *, *, 2800)$  and  $(*, Toronto, *, 1200)$  are 1-D cells,  $(Jan, *, Business, 150)$  is a 2-D cell, and  $(Jan, Toronto, Business, 45)$  is a 3-D cell. Here, all base cells are 3-D, whereas 1-D and 2-D cells are aggregate cells. ■

An ancestor-descendant relationship may exist between cells. In an  $n$ -dimensional data cube, an  $i$ -D cell  $\mathbf{a} = (a_1, a_2, \dots, a_n, \text{measures}_{\mathbf{a}})$  is an **ancestor** of a  $j$ -D cell  $\mathbf{b} = (b_1, b_2, \dots, b_n, \text{measures}_{\mathbf{b}})$ , and  $\mathbf{b}$  is a **descendant** of  $\mathbf{a}$ , if and only if (1)  $i < j$ , and (2) for  $1 \leq m \leq n$ ,  $a_m = b_m$  whenever  $a_m \neq “*”$ . In particular, cell  $\mathbf{a}$  is called a **parent** of cell  $\mathbf{b}$ , and  $\mathbf{b}$  is a **child** of  $\mathbf{a}$ , if and only if  $j = i + 1$  and  $\mathbf{b}$  is a descendant of  $\mathbf{a}$ .

<sup>1</sup>Cells are multidimensional (vectors). Variables representing vectors are represented in bold italic font in this text.



**Example 4.2 Ancestor and descendant cells.** Referring to our previous example, 1-D cell  $\mathbf{a} = (\text{Jan}, *, *, 2800)$ , and 2-D cell  $\mathbf{b} = (\text{Jan}, *, \text{Business}, 150)$ , are *ancestors* of 3-D cell  $\mathbf{c} = (\text{Jan}, \text{Toronto}, \text{Business}, 45)$ ;  $\mathbf{c}$  is a *descendant* of both  $\mathbf{a}$  and  $\mathbf{b}$ ;  $\mathbf{b}$  is a *parent* of  $\mathbf{c}$ , and  $\mathbf{c}$  is a *child* of  $\mathbf{b}$ . ■

In order to ensure fast on-line analytical processing, it is sometimes desirable to precompute the **full cube**, i.e., all the cells of all of the cuboids for a given data cube. This, however, is exponential to the number of dimensions. That is, a data cube of  $n$  dimensions contains  $2^n$  cuboids. There are even more cuboids if we consider concept hierarchies for each dimension.<sup>2</sup> In addition, the size of each cuboid depends on the cardinality of its dimensions. Thus, precomputation of the full cube can require huge and often excessive amounts of memory.

Nonetheless, full cube computation algorithms are important. Individual cuboids may be stored on secondary storage and accessed when necessary. Alternatively, we can use such algorithms to compute smaller cubes, consisting of a subset of the given set of dimensions, or a smaller range of possible values for some of the dimensions. In such cases, the smaller cube is a full cube for the given subset of dimensions and/or dimension values. A thorough understanding of full cube computation methods will help us develop efficient methods for computing partial cubes. Hence, it is important to explore scalable methods for computing all of the cuboids making up a data cube, that is, for full materialization. These methods must take into consideration the limited amount of main memory available for cuboid computation, the total size of the computed data cube, as well as the time required for such computation.

Partial materialization of data cubes offers an interesting trade-off between storage space and response time for OLAP. Instead of computing the full cube, we can compute only a subset of the data cube's cuboids, or subcubes consisting of subsets of cells from the various cuboids.

Many cells in a cuboid may actually be of little or no interest to the data analyst. Recall that each cell in a full cube records an aggregate value. Measures such as *count*, *sum*, or *sales.in.dollars* are commonly used. For many cells in a cuboid, the measure value will be zero. When the product of the cardinalities for the dimensions in a cuboid is large relative to the number of nonzero-valued tuples that are stored in the cuboid, then we say that the cuboid is **sparse**. If a cube contains many sparse cuboids, we say that the cube is **sparse**.

In many cases, a substantial amount of the cube's space could be taken up by a large number of cells with very low measure values. This is because the cube cells are often quite sparsely distributed within a multiple dimensional space. For example, a customer may only buy a few items in a store at a time. Such an event will generate only a few nonempty cells, leaving most other cube cells empty. In such situations, it is useful to materialize only those cells in a cuboid (group-by) whose measure value is above some minimum threshold. In a data cube for sales, say, we may wish to materialize only those cells for which *count*  $\geq 10$  (i.e., where at least 10 tuples exist for the cell's given combination of dimensions), or only those cells representing *sales*  $\geq \$100$ . This not only saves processing time and disk space, but also leads to a more focused analysis. The cells that cannot pass the threshold are likely to be too trivial to warrant further analysis. Such partially materialized cubes are known as **iceberg cubes**. The minimum threshold is called the **minimum support threshold**, or *minimum support* (*min\_sup*), for short. By materializing only a fraction of the cells in a data cube, the result is seen as the "tip of the iceberg", where the "iceberg" is the potential full cube including all cells. An iceberg cube can be specified with an SQL query, as shown in the following example.

**Example 4.3 Iceberg cube.**

```
compute cube sales.iceberg as
select month, city, customer_group, count(*)
from salesInfo
cube by month, city, customer_group
having count(*) >= min_sup
```

The `compute cube` statement specifies the precomputation of the iceberg cube, *sales.iceberg*, with the dimensions *month*, *city*, and *customer\_group*, and the aggregate measure `count()`. The input tuples are in the *salesInfo* relation.

<sup>2</sup>Equation (3.1) gives the total number of cuboids in a data cube where each dimension has an associated concept hierarchy.

The **cube by** clause specifies that aggregates (group-by's) are to be formed for each of the possible subsets of the given dimensions. If we were computing the full cube, each group-by would correspond to a cuboid in the data cube lattice. The constraint specified in the **having** clause is known as the **iceberg condition**. Here, the iceberg measure is *count*. Note that the iceberg cube computed for Example 4.3 could be used to answer group-by queries on any combination of the specified dimensions of the form **having count(\*)**  $\geq v$ , where  $v \geq \text{min\_sup}$ . Instead of *count*, the iceberg condition could specify more complex measures, such as *average*.

If we were to omit the **having** clause of our example, we would end up with the full cube. Let's call this cube *sales\_cube*. The iceberg cube, *sales\_iceberg*, excludes all the cells of *sales\_cube* whose count is less than *min\_sup*. Obviously, if we were to set the minimum support to 1 in *sales\_iceberg*, the resulting cube would be the full cube, *sales\_cube*. ■

A naïve approach to computing an iceberg cube would be to first compute the full cube and then prune the cells that do not satisfy the iceberg condition. However, this is still prohibitively expensive. An efficient approach is to compute only the iceberg cube directly without computing the full cube. Sections 4.1.3 and 4.1.4 discuss methods for efficient iceberg cube computation.

Introducing iceberg cubes will lessen the burden of computing trivial aggregate cells in a data cube. However, we could still end up with a large number of uninteresting cells to compute. For example, suppose that there are 20 base cells for a database of 100 dimensions, denoted as  $\{(a_1, a_2, a_3, \dots, a_{100}) : 10, (a_1, a_2, b_3, \dots, b_{100}) : 10\}$ , where 10 is the cell count. If the minimum support is set to 10, there will still be an impermissible number of cells to compute and store, although most of them are not interesting. For example, there are  $2^{101} - 6$  distinct aggregate cells<sup>3</sup>, like  $\{(a_1, a_2, a_3, a_4, \dots, a_{99}, *) : 10, \dots, (a_1, a_2, *, a_4, \dots, a_{99}, a_{100}) : 10, \dots, (a_1, a_2, a_3, *, \dots, *, *) : 10\}$ , but most of them do not contain much new information. If we ignore all of the aggregate cells that can be obtained by replacing some constants by \*'s while keeping the same measure value, there are only three distinct cells left:  $\{(a_1, a_2, a_3, \dots, a_{100}) : 10, (a_1, a_2, b_3, \dots, b_{100}) : 10, (a_1, a_2, *, \dots, *) : 20\}$ . That is, out of  $2^{101} - 6$  distinct aggregate cells, only three really offer new information.

To systematically compress a data cube, we need to introduce the concept of *closed coverage*. A cell, *c*, is a *closed cell* if there exists no cell, *d*, such that *d* is a specialization (descendant) of cell, *c* (that is, where *d* is obtained by replacing a \* in *c* by a non-\* value), and *d* has the same measure value as *c*. A **closed cube** is a data cube consisting of only closed cells. For example, the three cells derived above are the three closed cells of the data cube for the data set:  $\{(a_1, a_2, a_3, \dots, a_{100}) : 10, (a_1, a_2, b_3, \dots, b_{100}) : 10\}$ . They form the lattice of a closed cube as shown in Figure 4.2. Other nonclosed cells can be derived from their corresponding closed cells in this lattice. For example,  $(a_1, *, *, \dots, *) : 20$  can be derived from  $(a_1, a_2, *, \dots, *) : 20$  because the former is a generalized nonclosed cell of the latter. Similarly, we have  $(a_1, a_2, b_3, *, \dots, *) : 10$ .

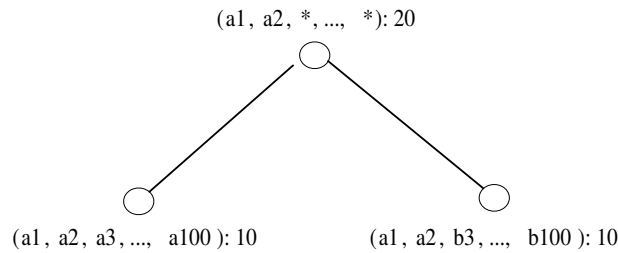


Figure 4.2: Three closed cells forming the lattice of a closed cube.

Another strategy for partial materialization is to precompute only the cuboids involving a small number of dimensions, such as 3 to 5. These cuboids form a **cube shell** for the corresponding data cube. Queries on additional combinations of the dimensions will have to be computed on the fly. For example, we could compute all cuboids with 3 dimensions or less in an *n*-dimensional data cube, resulting in cube shell of size 3. This, however,

<sup>3</sup>The proof is left as an exercise.

can still result in a large number of cuboids to compute, particularly when  $n$  is large. Alternatively, we can choose to precompute only portions or *fragments* of the cube shell, based on cuboids of interest. Section 4.1.5 discusses a method for computing such **shell fragments** and explores how they can be used for efficient OLAP query processing.

### General Strategies for Cube Computation

With different kinds of cubes as described above, we can expect that there are a good number of methods for efficient computation. In general, there are two basic data structures used for storing cuboids. Relational tables are used as the basic data structure for the implementation of relational OLAP (ROLAP), while multidimensional arrays are used as the basic data structure in multidimensional OLAP (MOLAP). Although ROLAP and MOLAP may each explore different cube computation techniques, some optimization “tricks” can be shared among the different data representations. The following are general optimization techniques for the efficient computation of data cubes.

**Optimization Technique 1: Sorting, hashing, and grouping.** Sorting, hashing, and grouping operations should be applied to the dimension attributes in order to reorder and cluster related tuples.

In cube computation, aggregation is performed on the tuples (or cells) that share the same set of dimension values. Thus it is important to explore sorting, hashing, and grouping operations to access and group such data together to facilitate computation of such aggregates.

For example, to compute total sales by *branch*, *day*, and *item*, it is more efficient to sort tuples or cells by *branch*, and then by *day*, and then group them according to the *item* name. Efficient implementations of such operations in large data sets have been extensively studied in the database research community. Such implementations can be extended to data cube computation.

This technique can also be further extended to perform **shared-sorts**, i.e., sharing sorting costs across multiple cuboids when sort-based methods are used, or to perform **shared-partitions**, i.e., sharing the partitioning cost across multiple cuboids when hash-based algorithms are used.

**Optimization Technique 2: Simultaneous aggregation and caching intermediate results.** In cube computation, it is efficient to compute higher-level aggregates from previously computed lower-level aggregates, rather than from the base fact table. Moreover, simultaneous aggregation from cached intermediate computation results may lead to the reduction of expensive disk I/O operations.

For example, to compute sales by *branch*, we can use the intermediate results derived from the computation of a lower-level cuboid, such as sales by *branch* and *day*. This technique can be further extended to perform **amortized-scans**, i.e., computing as many cuboids as possible at the same time to amortize disk reads.

**Optimization Technique 3: Aggregation from the smallest-child, when there exist multiple child cuboids.** When there exist multiple child cuboids, it is usually more efficient to compute the desired parent (i.e., more generalized) cuboid from the smallest, previously computed child cuboid.

For example, to compute a sales cuboid,  $C_{branch}$ , when there exist two previously computed cuboids,  $C_{\{branch, year\}}$ , and  $C_{\{branch, item\}}$ , it is obviously more efficient to compute  $C_{branch}$  from the former than from the latter if there are many more distinct items than distinct years.

There are many other optimization tricks that may further improve the computational efficiency. For example, *string dimension attributes can be mapped to integers with values ranging from zero to the cardinality of the attribute*. However, the following optimization technique plays a particularly important role in iceberg cube computation.

**Optimization Technique 4: The Apriori pruning method can be explored to compute iceberg cubes efficiently.** The **Apriori property**<sup>4</sup>, in the context of data cubes, states as follows: *If a given cell does not satisfy*

<sup>4</sup>The Apriori property was proposed in the Apriori algorithm for association rule mining by R. Agrawal and R. Srikant [AS94]. Many algorithms in association rule mining have adopted this property. Association rule mining is the topic of Chapter 5.

*minimum support, then no descendant (i.e., more specialized or detailed version) of the cell will satisfy minimum support either.* This property can be used to substantially reduce the computation of iceberg cubes.

Recall that the specification of iceberg cubes contains an iceberg condition, which is a constraint on the cells to be materialized. A common iceberg condition is that the cells must satisfy a *minimum support* threshold, such as a minimum count or sum. In this situation, the Apriori property can be used to prune away the exploration of the descendants of the cell. For example, if the count of a cell,  $c$ , in a cuboid is less than a minimum support threshold,  $v$ , then the count of any of  $c$ 's descendant cells in the lower level cuboids can never be higher than  $v$ , and thus can be pruned. In other words, if a condition (e.g., the iceberg condition specified in a **having** clause) is violated for some cell  $c$ , then every descendant of  $c$  will also violate that condition. Measures that obey this property are known as **antimonotonic**<sup>5</sup>. This form of pruning was made popular in association rule mining, yet also aids in data cube computation by cutting processing time and disk space requirements. It can lead to a more focused analysis since cells that cannot pass the threshold are unlikely to be of interest.

In the following subsections, we introduce several popular methods for efficient cube computation that explore some or all of the above optimization strategies. Section 4.1.2 describes the *multiway array aggregation* (MultiWay) method for computing full cubes. The remaining sections describe methods for computing iceberg cubes. Section 4.1.3 describes a method known as BUC, which computes iceberg cubes from the apex cuboid, downwards. Section 4.1.4 describes the **Star-Cubing** method, which integrates top-down and bottom-up computation. Section 4.1.5 describes a minimal cubing approach that computes shell fragments for efficient high-dimensional OLAP. Finally, Section 4.1.6 describes a method for computing iceberg cubes with complex measures, such as *average*. To simplify our discussion, we exclude the cuboids that would be generated by climbing up any existing hierarchies for the dimensions. Such kinds of cubes can be computed by extension of the discussed methods. Methods for efficient computation of closed cubes are left as an exercise for interested readers.

#### 4.1.2 Multiway Array Aggregation for Full Cube Computation

The *Multiway Array Aggregation* (or simply MultiWay) method computes a full data cube by using a multidimensional array as its basic data structure. It is a typical MOLAP approach that uses direct array addressing, where dimension values are accessed via the position or index of their corresponding array locations. Hence, MultiWay cannot perform any value-based reordering as an optimization technique. A different approach is developed for the array-based cube construction, as follows.

1. Partition the array into chunks. A **chunk** is a subcube that is small enough to fit into the memory available for cube computation. **Chunking** is a method for dividing an  $n$ -dimensional array into small  $n$ -dimensional chunks, where each chunk is stored as an object on disk. The chunks are compressed so as to remove wasted space resulting from empty array cells (i.e., cells that do not contain any valid data, that is, whose cell count is zero). For instance, “*chunkID + offset*” can be used as a cell addressing mechanism to **compress a sparse array structure** and when searching for cells within a chunk. Such a compression technique is powerful enough to handle sparse cubes, both on disk and in memory.
2. Compute aggregates by visiting (i.e., accessing the values at) cube cells. The order in which cells are visited can be optimized so as to *minimize the number of times that each cell must be revisited*, thereby reducing memory access and storage costs. The trick is to exploit this ordering so that partial aggregates can be computed simultaneously, and any unnecessary revisiting of cells is avoided.

Since this chunking technique involves “overlapping” some of the aggregation computations, it is referred to as **multiway array aggregation**. It performs **simultaneous aggregation**—that is, it computes aggregations simultaneously on multiple dimensions.

We explain this approach to array-based cube construction by looking at a concrete example.

---

<sup>5</sup> **Antimonotone** is based on *condition violation*. This differs from **monotone**, which is based on *condition satisfaction*.

**Example 4.4 Multiway array cube computation.** Consider a 3-D data array containing the three dimensions  $A$ ,  $B$ , and  $C$ . The 3-D array is partitioned into small, memory-based chunks. In this example, the array is partitioned into 64 chunks as shown in Figure 4.3. Dimension  $A$  is organized into four equal-sized partitions,  $a_0, a_1, a_2$ , and  $a_3$ . Dimensions  $B$  and  $C$  are similarly organized into four partitions each. Chunks 1, 2, ..., 64 correspond to the subcubes  $a_0b_0c_0, a_1b_0c_0, \dots, a_3b_3c_3$ , respectively. Suppose that the cardinality of the dimensions  $A$ ,  $B$ , and  $C$  is 40, 400, and 4000, respectively. Thus, the size of the array for each dimension,  $A$ ,  $B$ , and  $C$ , is also 40, 400, and 4000, respectively. The size of each partition in  $A$ ,  $B$ , and  $C$  is therefore 10, 100, and 1000, respectively.

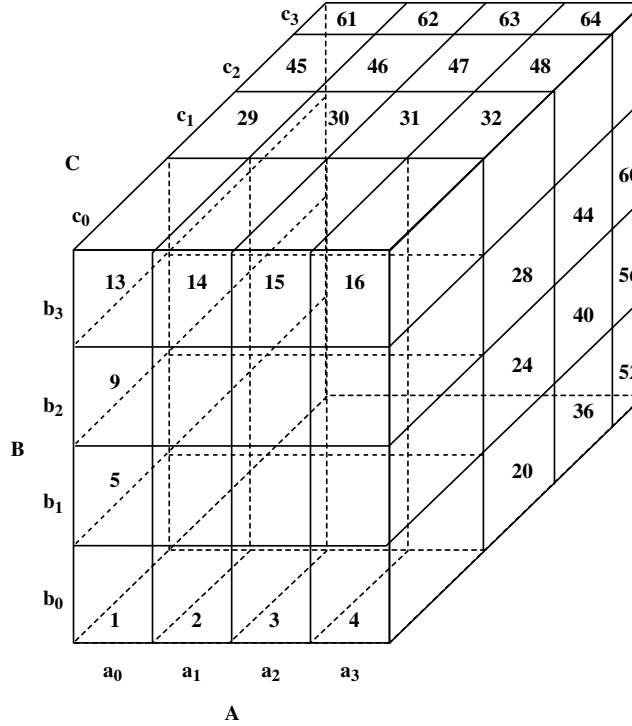


Figure 4.3: A 3-D array for the dimensions  $A$ ,  $B$ , and  $C$ , organized into 64 *chunks*. Each chunk is small enough to fit into the memory available for cube computation.

Full materialization of the corresponding data cube involves the computation of all of the cuboids defining this cube. The resulting full cube consists of the following cuboids.

- The base cuboid, denoted by  $ABC$  (from which all of the other cuboids are directly or indirectly computed). This cube is already computed and corresponds to the given 3-D array.
- The 2-D cuboids,  $AB$ ,  $AC$ , and  $BC$ , which respectively correspond to the group-by's  $AB$ ,  $AC$ , and  $BC$ . These cuboids must be computed.
- The 1-D cuboids,  $A$ ,  $B$ , and  $C$ , which respectively correspond to the group-by's  $A$ ,  $B$ , and  $C$ . These cuboids must be computed.
- The 0-D (apex) cuboid, denoted by  $all$ , which corresponds to the group-by  $()$ ; that is, there is no group-by here. This cuboid must be computed. It consists of one value, which is simply the total count of all of the tuples in  $ABC$ .

Let's look at how the multiway array aggregation technique is used in this computation. There are many possible orderings with which chunks can be read into memory for use in cube computation. Consider the ordering

labeled from 1 to 64, shown in Figure 4.3. Suppose we would like to compute the  $b_0c_0$  chunk of the  $BC$  cuboid. We allocate space for this chunk in *chunk memory*. By scanning chunks 1 to 4 of  $ABC$ , the  $b_0c_0$  chunk is computed. That is, the cells for  $b_0c_0$  are aggregated over  $a_0$  to  $a_3$ . The chunk memory can then be assigned to the next chunk,  $b_1c_0$ , which completes its aggregation after the scanning of the next four chunks of  $ABC$ : 5 to 8. Continuing in this way, the entire  $BC$  cuboid can be computed. Therefore, only *one* chunk of  $BC$  needs to be in memory, at a time, for the computation of all of the chunks of  $BC$ .

In computing the  $BC$  cuboid, we will have scanned each of the 64 chunks. “Is there a way to avoid having to rescan all of these chunks for the computation of other cuboids, such as  $AC$  and  $AB$ ?” The answer is, most definitely—yes. This is where the “multiway computation” or “simultaneous aggregation” idea comes in. For example, when chunk 1 (i.e.,  $a_0b_0c_0$ ) is being scanned (say, for the computation of the 2-D chunk  $b_0c_0$  of  $BC$ , as described above), all of the other 2-D chunks relating to  $a_0b_0c_0$  can be simultaneously computed. That is, when  $a_0b_0c_0$  is being scanned, each of the three chunks,  $b_0c_0$ ,  $a_0c_0$ , and  $a_0b_0$ , on the three 2-D aggregation planes,  $BC$ ,  $AC$ , and  $AB$ , should be computed then as well. In other words, multiway computation simultaneously aggregates to each of the 2-D planes while a 3-D chunk is in memory.

Now let’s look at how different orderings of chunk scanning and of cuboid computation can affect the overall data cube computation efficiency. Recall that the size of the dimensions  $A$ ,  $B$ , and  $C$  is 40, 400, and 4000, respectively. Therefore, the largest 2-D plane is  $BC$  (of size  $400 \times 4000 = 1,600,000$ ). The second largest 2-D plane is  $AC$  (of size  $40 \times 4000 = 160,000$ ).  $AB$  is the smallest 2-D plane (with a size of  $40 \times 400 = 16,000$ ).

Suppose that the chunks are scanned in the order shown, from chunk 1 to 64. By scanning in this order, one chunk of the largest 2-D plane,  $BC$ , is *fully* computed for each row scanned. That is,  $b_0c_0$  is fully aggregated after scanning the row containing chunks 1 to 4;  $b_1c_0$  is fully aggregated after scanning chunks 5 to 8, and so on. In comparison, the complete computation of one chunk of the second largest 2-D plane,  $AC$ , requires scanning 13 chunks, given the ordering from 1 to 64. That is,  $a_0c_0$  is fully aggregated only after the scanning of chunks 1, 5, 9, and 13. Finally, the complete computation of one chunk of the smallest 2-D plane,  $AB$ , requires scanning 49 chunks. For example,  $a_0b_0$  is fully aggregated after scanning chunks 1, 17, 33, and 49. Hence,  $AB$  requires the longest scan of chunks in order to complete its computation. To avoid bringing a 3-D chunk into memory more than once, the minimum memory requirement for holding all relevant 2-D planes in chunk memory, according to the chunk ordering of 1 to 64, is as follows:  $40 \times 400$  (for the whole  $AB$  plane) +  $40 \times 1000$  (for one row of the  $AC$  plane) +  $100 \times 1000$  (for one chunk of the  $BC$  plane) =  $16,000 + 40,000 + 100,000 = 156,000$  memory units.

Suppose, instead, that the chunks are scanned in the order 1, 17, 33, 49, 5, 21, 37, 53, and so on. That is, suppose the scan is in the order of first aggregating towards the  $AB$  plane, and then towards the  $AC$  plane, and lastly towards the  $BC$  plane. The minimum memory requirement for holding 2-D planes in chunk memory would be as follows:  $400 \times 4000$  (for the whole  $BC$  plane) +  $40 \times 1000$  (for one row of the  $AC$  plane) +  $10 \times 100$  (for one chunk of the  $AB$  plane) =  $1,600,000 + 40,000 + 1000 = 1,641,000$  memory units. Notice that this is *more than 10 times* the memory requirement of the scan ordering of 1 to 64.

Similarly, we can work out the minimum memory requirements for the multiway computation of the 1-D and 0-D cuboids. Figure 4.4 shows the most efficient ordering and the least efficient ordering, based on the minimum memory requirements for the data cube computation. The most efficient ordering is the chunk ordering of 1 to 64. ■

Example 4.4 assumes that there is enough memory space for *one-pass* cube computation (i.e., to compute all of the cuboids from one scan of all of the chunks). If there is insufficient memory space, the computation will require more than one pass through the 3-D array. In such cases, however, the basic principle of ordered chunk computation remains the same. **MultiWay** is most effective when the product of the cardinalities of dimensions is moderate and the data are not too sparse. When the dimensionality is high or the data are very sparse, the in-memory arrays become too large to fit in memory, and this method becomes infeasible.

With the use of appropriate sparse array compression techniques and careful ordering of the computation of cuboids, it has been shown by experiments that **MultiWay** array cube computation is significantly faster than traditional ROLAP (relational record-based) computation. Unlike ROLAP, the array structure of **MultiWay** does not require saving space to store search keys. Furthermore, **MultiWay** uses direct array addressing, which is faster than the key-based addressing search strategy of ROLAP. For ROLAP cube computation, instead of cubing a

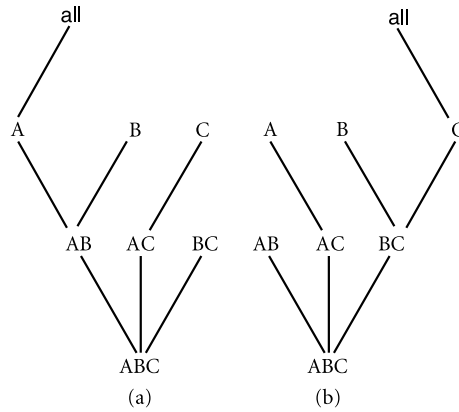


Figure 4.4: Two orderings of multiway array aggregation for computation of the 3-D cube of Example 4.4: (a) most efficient ordering of array aggregation (minimum memory requirements = 156,000 memory units); (b) least efficient ordering of array aggregation (minimum memory requirements = 1,641,000 memory units). [TO EDITOR We may redo this figure to show ordering of movement through lattice.]

table directly, it can be faster to convert the table to an array, cube the array, and then convert the result back to a table. However, this observation works only for cubes with a relatively small number of dimensions since the number of cuboids to be computed is exponential to the number of dimensions.

“What would happen if we tried to use MultiWay to compute iceberg cubes?” Remember that the Apriori property states that if a given cell does not satisfy minimum support, then neither will any of its descendants. Unfortunately, MultiWay’s computation starts from the base cuboid and progresses upwards towards more generalized, ancestor cuboids. It cannot take advantage of Apriori pruning, which requires a parent node to be computed before its child (i.e., more specific) nodes. For example, if the count of a cell  $c$  in, say,  $AB$ , does not satisfy the minimum support specified in the iceberg condition, then we cannot prune away computation of  $c$ ’s ancestors in the  $A$  or  $B$  cuboids, since the count of these cells may be greater than that of  $c$ .

### 4.1.3 BUC: Computing Iceberg Cubes from the Apex Cuboid Downwards

BUC is an algorithm for the computation of sparse and iceberg cubes. Unlike MultiWay, BUC constructs the cube from the apex cuboid towards the base cuboid. This allows BUC to share data partitioning costs. This order of processing also allows BUC to prune during construction, using the Apriori property.

Figure 4.1 shows a lattice of cuboids, making up a 3-D data cube with the dimensions  $A$ ,  $B$ , and  $C$ . The apex (0-D) cuboid, representing the concept **all** (that is,  $(*, *, *)$ ), is at the top of the lattice. This is the most aggregated or generalized level. The 3-D base cuboid,  $ABC$ , is at the bottom of the lattice. It is the least aggregated (most detailed or specialized) level. This representation of a lattice of cuboids, with the apex at the top and the base at the bottom, is commonly accepted in data warehousing. It consolidates the notions of *drill down* (where we can move from a highly aggregated cell to lower, more detailed cells) and *roll up* (where we can move from detailed, low level cells to higher level, more aggregated cells).

BUC stands for “Bottom-Up Construction”. However, according to the lattice convention described above and used throughout this book, the order of processing of BUC is actually top-down! The authors of BUC view a lattice of cuboids in the reverse order, with the apex cuboid at the bottom and the base cuboid at the top. In that view, BUC does bottom-up construction. However, since we adopt the application world view where *drill-down* refers to drilling from the apex cuboid down towards the base cuboid, the exploration process of BUC is regarded as top-down.

The BUC algorithm is shown in Figure 4.5. We first give an explanation of the algorithm, and then follow up

**Algorithm: BUC.** Algorithm for the computation of sparse and iceberg cubes.

**Input:**

- *input*: the relation to aggregate;
- *dim*: the starting dimension for this iteration.

**Globals:**

- constant *numDims*: the total number of dimensions;
- constant *cardinality[numDims]*: the cardinality of each dimension;
- constant *min\_sup*: the minimum number of tuples in a partition in order for it to be output;
- *outputRec*: the current output record;
- *dataCount[numDims]*: stores the size of each partition. *dataCount[i]* is a list of integers of size *cardinality[i]*.

**Output:** Recursively output the iceberg cube cells satisfying the minimum support.

**Method:**

```

(1) Aggregate(input); // scans entire input to compute measure, e.g., total count, and places the result in outputRec
(2) if input.count() == 1 then { // Optimization
    WriteAncestors(input[0], dim); return; }
(3) write outputRec;
(4) for (d = dim; d < numDims; d++) do { //Partition each dimension
(5)   C = cardinality[d];
(6)   Partition(input, d, C, dataCount[d]); //create C partitions of data for dimension d
(7)   k = 0;
(8)   for (i = 0; i < C; i++) do // for each partition (each value of dimension d)
(9)     c = dataCount[d][i];
(10)    if c >= min_sup then // test the iceberg condition
(11)      outputRec.dim[d] = input[k].dim[d];
(12)      BUC(input[k...k + c], d + 1); // aggregate on next dimension
(13)    endif
(14)    k += c;
(15)  endfor
(16)  outputRec.dim[d] = all;
(17) endfor

```

■

Figure 4.5: BUC algorithm for the computation of sparse or iceberg cubes [BR99].

with an example. Initially, the algorithm is called with the input relation (set of tuples). BUC aggregates the entire input (line 1) and writes the resulting total (line 3). (Line 2 is an optimization feature that is discussed later in our example.) For each dimension *d* (line 4), the input is partitioned on *d* (line 6). On return from Partition(), *dataCount* contains the total number of tuples for each distinct value of dimension *d*. Each distinct value of *d* forms its own partition. Line 8 iterates through each partition. Line 10 tests the partition for minimum support. That is, if the number of tuples in the partition satisfies (i.e., is  $\geq$ ) the minimum support, then the partition becomes the input relation for a recursive call made to BUC, which computes the iceberg cube on the partitions for dimensions *d* + 1 to *numDims* (line 12). Note that for a full cube (that is, where minimum support in the having clause is 1), the minimum support condition is always satisfied. Thus, the recursive call descends one level deeper into the lattice. Upon return from the recursive call, we continue with the next partition for *d*. After all the partitions have been processed, the entire process is repeated for each of the remaining dimensions.

We explain how BUC works with the following example.

**Example 4.5 BUC construction of an iceberg cube.** Consider the iceberg cube expressed in SQL as follows:

```

compute cube iceberg_cube as
select A, B, C, D, count(*)
from R
cube by A, B, C, D

```





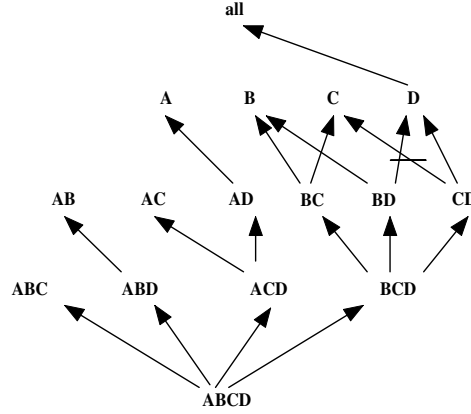


Figure 4.7: Bottom-up computation.

be, thereby providing BUC with greater opportunity for pruning. Similarly, the more uniform a dimension is (i.e., having less skew), the better it is for pruning.

BUC's major contribution is the idea of sharing partitioning costs. However, unlike MultiWay, it does not share the computation of aggregates between parent and child group-by's. [TO EDITOR We have used "group-by's" instead of "group-bys" (although, there may be some inconsistencies). However, "group-bys" (without the apostrophe) is more correct, but harder to read. Which should we use?] For example, the computation of cuboid  $AB$  does not help that of  $ABC$ . The latter needs to be computed essentially from scratch.

#### 4.1.4 Star-cubing: Computing Iceberg Cubes Using a Dynamic Star-tree Structure

In this section, we describe the Star-Cubing algorithm for computing iceberg cubes. Star-Cubing combines the strengths of the other methods we have studied to this point. It integrates top-down and bottom-up cube computation and explores both multidimensional aggregation (similar to MultiWay) and Apriori-like pruning (similar to BUC). It operates from a data structure called a star-tree, which performs lossless data compression, thereby reducing the computation time and memory requirements.

A key idea behind Star-Cubing is the concept of *shared dimensions*. To build up to this notion, let's have a look at Figure 4.7, which shows a bottom-up computation for a cube involving the dimensions  $A$ ,  $B$ ,  $C$ , and  $D$ . The order of computation is from the base (least generalized) cuboid, upwards towards the apex (most generalized) cuboid. This order of computation is similar to that of MultiWay. Note that for ease of our future explanation, the ordering shown for the exploration of cuboids is depth-first *from right to left*. For example, cuboid  $BCD$  was explored first, followed by  $CD$ ,  $D$ , and *all*, followed by  $C$ , then  $BD$ , and so on. The ancestors of  $ABC$  are not shown (namely,  $AB$ ,  $AC$ ,  $BC$ ,  $A$ ,  $B$ ,  $C$ ) because they would have already been computed along the way from other cuboids. This is different from Example 4.4 for MultiWay, in which the exploration of cuboids happened to be from *left to right*. The Star-Cubing algorithm explores both the bottom-up and top-down computation models: On the global computation order, it uses the bottom-up model similar to Figure 4.7. However, it has a sub-layer underneath based on the top-down model, which explores the notion of shared dimensions, as we shall see below. This integration allows the algorithm to aggregate on multiple dimensions while still partitioning parent group-by's and pruning child group-by's that do not satisfy the iceberg condition.

By studying Figure 4.7, we can make the following generalization: all the cuboids in the subtree rooted at  $ACD$  include dimension  $A$ , all those rooted at  $ABD$  include dimensions  $AB$ , and all those rooted at  $ABC$  include dimensions  $ABC$  (even though there is only one such cuboid). We call these common dimensions the **shared dimensions** of those particular subtrees. Based on this concept, Figure 4.7 is extended to Figure 4.8, which shows the spanning tree marked with the shared dimensions. For example,  $ACD/A$  means cuboids  $ACD$  has shared dimension  $A$ ,  $ABD/AB$  means cuboid  $ABD$  has shared dimension  $AB$ , and so on.

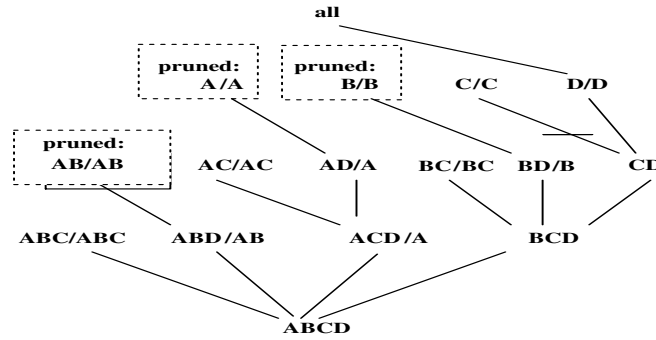


Figure 4.8: Star-Cubing: Bottom-up computation with top-down expansion of shared dimensions.

The introduction of shared dimensions facilitates shared computation. Since the shared dimensions are identified early on in the tree expansion, we can avoid recomputing them later. For example, cuboid  $AB$  extending from  $ABD$  in Figure 4.8 would actually be pruned because  $AB$  was already computed in  $ABD/AB$ . Similarly, cuboid  $A$  extending from  $AD$  would also be pruned because it was already computed in  $ACD/A$ .

Shared dimensions allow us to do Apriori-like pruning if the measure of an iceberg cube, such as *count*, is antimonotonic, that is, if the aggregate value on a shared dimension does not satisfy the iceberg condition, then *all of the cells descending from this shared dimension cannot satisfy the iceberg condition either*. Such cells and all of their descendants can be pruned. This is because these descendant cells are, by definition, more specialized (i.e., contain more dimensions) than those in the shared dimension(s). The number of tuples covered by the descendant cells will be less than or equal to the number of tuples covered by the shared dimensions. Therefore, if the aggregate value on a shared dimension fails the iceberg condition, the descendent cells cannot satisfy it either.

**Example 4.6 Pruning shared dimensions.** If the value in the shared dimension  $A$  is  $a_1$  and it fails to satisfy the iceberg condition, the whole subtree rooted at  $a_1CD/a_1$  (including  $a_1C/a_1C$ ,  $a_1D/a_1$ ,  $a_1/a_1$ ) can be pruned since they are all more specialized versions of  $a_1$ . ■

To explain how the Star-Cubing algorithm works, we need to explain a few more concepts, namely, *cuboid trees*, *star nodes*, and *star trees*.

We use trees to represent individual cuboids. Figure 4.9 shows a fragment of the **cuboid tree** of the base cuboid,  $ABCD$ . Each level in the tree represents a dimension, and each node represents an attribute value. Each node has four fields: the attribute value, aggregate value, pointer(s) to possible descendant(s), and pointer to possible sibling. Tuples in the cuboid are inserted one by one into the tree. A path from the root to a leaf node represents a tuple. For example, node  $c_2$  in the tree has an aggregate (count) value of 5, which indicates that there are five cells of value  $(a_1, b_1, c_2, *)$ . This representation collapses the common prefixes to save memory usage and allows us to aggregate the values at internal nodes. With aggregate values at internal nodes, we can prune based on shared dimensions. For example, the cuboid tree of  $AB$  can be used to prune possible cells in  $ABD$ .

If the single dimensional aggregate on an attribute value  $p$  does not satisfy the iceberg condition, it is useless to distinguish such nodes in the iceberg cube computation. Thus the node  $p$  can be replaced by  $*$  so that the cuboid tree can be further compressed. We say that the node  $p$  in an attribute  $A$  is a **star node** if the single dimensional aggregate on  $p$  does not satisfy the iceberg condition; otherwise,  $p$  is a *non-star node*. A cuboid tree that consists of only non-star nodes and star nodes is called a **star-tree**.

The following is an example of star-tree construction.

**Example 4.7 Star-tree construction.** A base cuboid table is shown in Table 4.1. There are 5 tuples and 4 dimensions. The cardinalities for dimensions  $A$ ,  $B$ ,  $C$ ,  $D$  are 2, 4, 3, 4, respectively. The one-dimensional aggregates for all attributes are shown in Table 4.2. Suppose  $\text{min\_support} = 2$  in the iceberg condition. Clearly, only attribute values  $a_1, a_2, b_1, c_3, d_4$  satisfy the condition. All the other values are below the threshold and thus

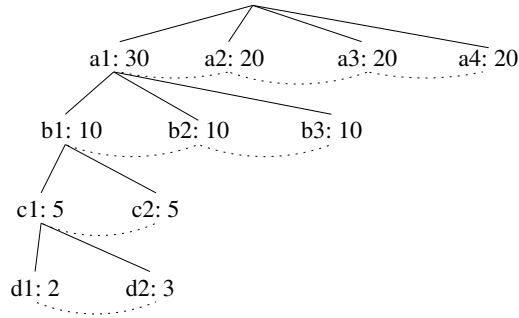


Figure 4.9: A fragment of the base cuboid tree. [TO EDITOR Please replace  $a1$  by  $a_1$ ,  $a2$  by  $a_2$ ,  $b1$  by  $b_1$ , etc. (change in notation style for consistency with text)!]

$A$	$B$	$C$	$D$	count
$a_1$	$b_1$	$c_1$	$d_1$	1
$a_1$	$b_1$	$c_3$	$d_3$	1
$a_1$	$b_2$	$c_2$	$d_2$	1
$a_2$	$b_3$	$c_3$	$d_4$	1
$a_2$	$b_4$	$c_3$	$d_4$	1

Table 4.1: Base (Cuboid) Table: Before star reduction.

become star nodes. By collapsing star nodes, the reduced base table is Table 4.3. Notice that the table contains two fewer rows and also fewer distinct values than Table 4.1.

We use the reduced base table to construct the cuboid tree since it is smaller. The resultant star-tree is shown in Figure 4.10. To help identify which nodes are star-nodes, a **star-table** is constructed for each star-tree. Figure 4.10 also shows the corresponding star-table for the star-tree (where only the star nodes are shown in the star-table). In actual implementation, a bit-vector or hash table could be used to represent the star-table for fast lookup. ■

By collapsing star nodes, the star-tree provides a *lossless* compression of the original data. It provides a good improvement in memory usage, yet the time required to search for nodes or tuples in the tree is costly. To reduce this cost, the nodes in the star-tree are sorted in alphabetic order for each dimension, with the star-nodes appearing first. In general, nodes are sorted in the order  $*, p_1, p_2, \dots, p_n$  at each level.

Now, let's see how the **Star-Cubing** algorithm uses star trees to compute an iceberg cube. The algorithm is given in Figure 4.1.4.

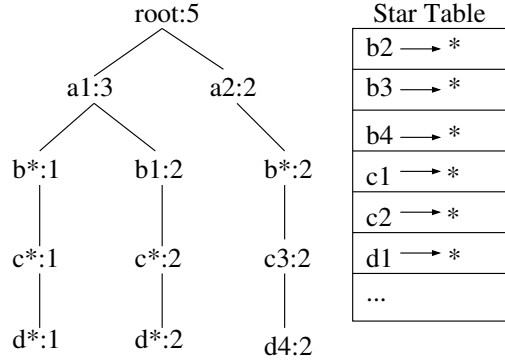
**Example 4.8 Star-cubing.** Using the star-tree generated in Example 4.7, we start the process of aggregation by traversing in a bottom-up fashion. Traversal is depth-first. The first stage (i.e., the processing of the first branch of the tree) is shown in Figure 4.11. The leftmost tree in the figure is the base star-tree. Each attribute value is shown with its corresponding aggregate value. In addition, subscripts by the nodes in the tree show the order of traversal. The remaining four trees are  $BCD$ ,  $ACD/A$ ,  $ABD/AB$ ,  $ABC/ABC$ . They are the child trees of the base star-tree, and correspond to the level of 3-dimensional cuboids above the base cuboid in Figure 4.8. The

Dimension	count = 1	count $\geq 2$
$A$	-	$a_1(3), a_2(2)$
$B$	$b_2, b_3, b_4$	$b_1(2)$
$C$	$c_1, c_2$	$c_3(3)$
$D$	$d_1, d_2, d_3$	$d_4(2)$

Table 4.2: One-Dimensional Aggregates.

<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	count
$a_1$	$b_1$	*	*	2
$a_1$	*	*	*	1
$a_2$	*	$c_3$	$d_4$	2

Table 4.3: Compressed Base Table: After star reduction.

Figure 4.10: Star tree and star table.  
[TO EDITOR Please replace  $a_1$  by  $a_1$ , etc!]

subscripts in them correspond to the same subscripts in the base tree—they denote the step or order in which they are created during the tree traversal. For example, when the algorithm is at step 1, the  $BCD$  child tree root is created. At step 2, the  $ACD/A$  child tree root is created. At step 3, the  $ABD/AB$  tree root and the  $b^*$  node in  $BCD$  are created.

When the algorithm has reached step 5, the trees in memory are exactly as shown in Figure 4.11. Since the depth-first traversal has reached a leaf at this point, it starts backtracking. Before traversing back, the algorithm notices that all possible nodes in the base dimension ( $ABC$ ) have been visited. This means the  $ABC/ABC$  tree is complete so the count is output and the tree is destroyed. Similarly, upon moving back from  $d^*$  to  $c^*$  and seeing that  $c^*$  has no siblings, the count in  $ABD/AB$  is also output and the tree is destroyed.

When the algorithm is at  $b^*$  during the back-traversal, it notices that there exists a sibling in  $b_1$ . Therefore, it will keep  $ACD/A$  in memory and perform depth-first search on  $b_1$  just as it did on  $b^*$ . This traversal and the resultant trees are shown in Figure 4.12. The child trees  $ACD/A$  and  $ABD/AB$  are created again but now with the new values from the  $b_1$  subtree. For example, notice that the aggregate count of  $c^*$  in the  $ACD/A$  tree has increased from 1 to 3. The trees that remained intact during the last traversal are reused and the new aggregate values are added on. For instance, another branch is added to the  $BCD$  tree.

Just like before, the algorithm will reach a leaf node at  $d^*$  and traverse back. This time, it will reach  $a_1$  and notice that there exists a sibling in  $a_2$ . In this case, all child trees except  $BCD$  in Figure 4.12 are destroyed. Afterwards, the algorithm will perform the same traversal on  $a_2$ .  $BCD$  continues to grow while the other subtrees start fresh with  $a_2$  instead of  $a_1$ . ■

There are two conditions that a node must satisfy in order to generate child trees: (1) the measure of the node must satisfy the iceberg condition; and (2) the tree to be generated must include at least one non-star (i.e., nontrivial) node. This is because if all the nodes were star nodes, then none of them would satisfy *min\_support*. Therefore, it would be a complete waste to compute them. This pruning is observed in Figures 4.11 and 4.12. For example, the left subtree extending from node  $a_1$  in the base-tree in Figure 4.11 does not include any non-star nodes. Therefore, the  $a_1CD/a_1$  subtree should not have been generated. It is shown, however, for illustration of the child tree generation process.

Star-Cubing is sensitive to the ordering of dimensions, as with other iceberg cube construction algorithms. For best performance, the dimensions are processed in order of decreasing cardinality. This leads to a better chance

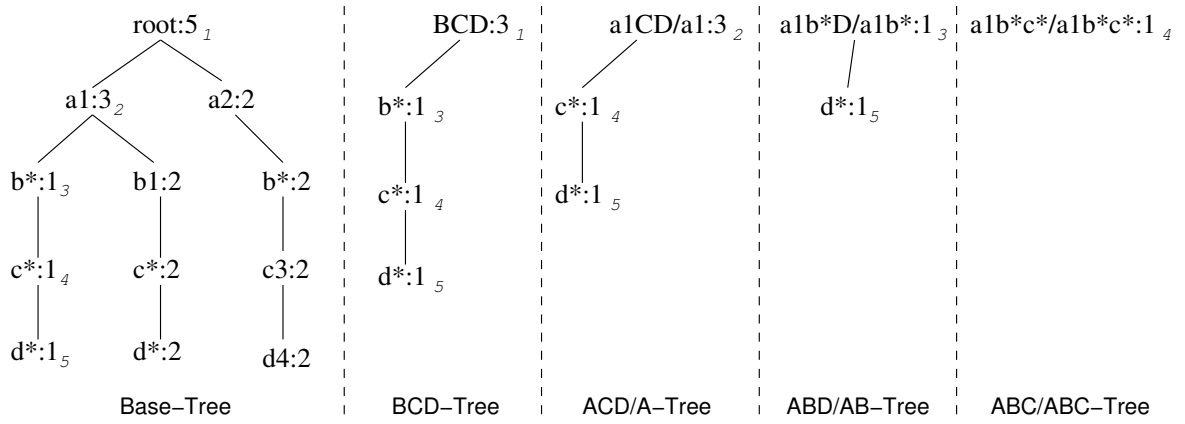


Figure 4.11: Aggregation Stage One: Processing of the left-most branch of Base-Tree.  
 [TO EDITOR Please italicize A,B,C,D (e.g., *BCD*-tree). Please replace  $a_1$  by  $a_1$ , etc!]

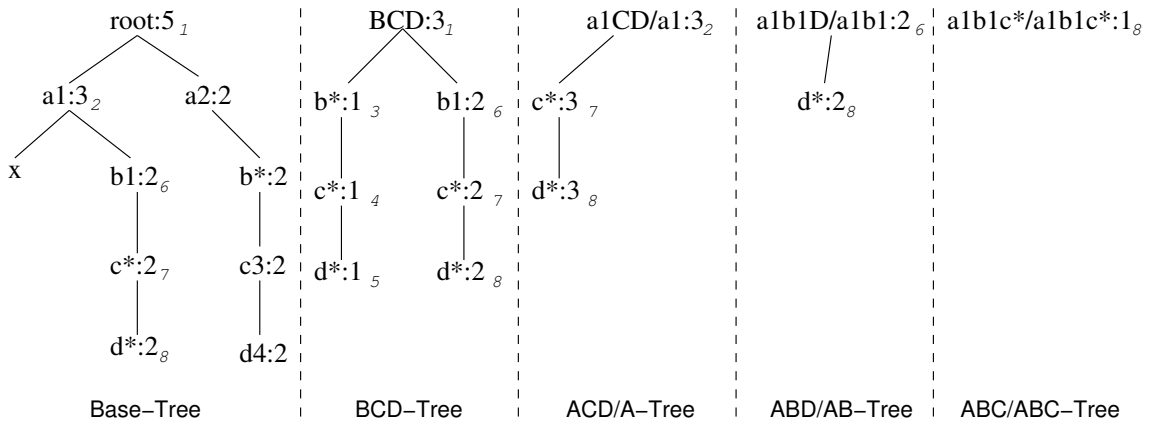


Figure 4.12: Aggregation Stage Two: Processing of the second branch of Base-Tree.  
 [TO EDITOR Please replace  $a_1$  by  $a_1$ , etc!]

**Algorithm:** Star-Cubing. Compute iceberg cubes by Star-Cubing.

**Input:**

- $R$ : a relational table;  $R$ ,
- $min\_support$ : minimum support threshold for the iceberg condition (taking count as the measure).

**Output:** The computed iceberg cube.

**Method:** Each star-tree corresponds to one cuboid tree node, and vice versa.

```

BEGIN
  scan  $R$  twice, create star-table  $S$  and star-tree  $T$ ;
  output count of  $T.root$ ;
  call starcubing( $T, T.root$ );
END

procedure starcubing( $T, cnode$ ) //  $cnode$ : current node
{
  (1) for each non-null child  $C$  of  $T$ 's cuboid tree
  (2)   insert or aggregate  $cnode$  to the corresponding
        position or node in  $C$ 's star-tree;
  (3) if ( $cnode.count \geq min\_support$ ) then {
  (4)   if ( $cnode \neq root$ ) then
  (5)     output  $cnode.count$ ;
  (6)   if ( $cnode$  is a leaf) then
  (7)     output  $cnode.count$ ;
  (8)   else { // initiate a new cuboid tree
  (9)     create  $C_C$  as a child of  $T$ 's cuboid tree;
  (10)    let  $T_C$  be  $C_C$ 's star-tree;
  (11)     $T_C.root$ 's count =  $cnode.count$ ;
  (12)   }
  (13) }
  (14) if ( $cnode$  is not a leaf) then
  (15)   [OLD: call][NEW: ] starcubing( $T, cnode.first\_child$ );
  (16) if ( $C_C$  is not null) then {
  (17)   [OLD: call][NEW: ] starcubing( $T_C, T_C.root$ );
  (18)   remove  $C_C$  from  $T$ 's cuboid tree; }
  (19) if ( $cnode$  has sibling) then
  (20)   [OLD: call][NEW: ] starcubing( $T, cnode.sibling$ );
  (21) remove  $T$ ;
}
```

Figure 4.13: The Star-Cubing algorithm.

of early pruning. This is because the higher the cardinality, the smaller the partitions, and therefore the higher possibility that the partition will be pruned.

Star-Cubing can also be used for full cube computation. When computing the full cube for a dense data set, Star-Cubing's performance is comparable with MultiWay, and is much faster than BUC. If the data set is sparse, Star-Cubing is significantly faster than MultiWay, and faster than BUC, in most cases. For iceberg cube computation, Star-Cubing is faster than BUC, where the data are skewed and the speedup factor increases as  $min\_support$  decreases.

#### 4.1.5 Precomputing Shell Fragments for Fast High-Dimensional OLAP

Recall the reason that we are interested in precomputing data cubes: Data cubes facilitate fast on-line analytical processing (OLAP) in a multidimensional data space. However, a full data cube of high dimensionality needs massive storage space and unrealistic computation time. Iceberg cubes provide a more feasible alternative, as we have seen, wherein the iceberg condition is used to specify the computation of only a subset of the full cube's cells. However, although an iceberg cube is smaller and requires less computation time than its corresponding full cube, it is not an ultimate solution. For one, the computation and storage of the iceberg cube itself can still be

costly. For example, if the base cuboid cell,  $(a_1, a_2, \dots, a_{60})$ , passes minimum support (or the iceberg threshold), it will generate  $2^{60}$  iceberg cube cells. Second, it is difficult to determine an appropriate iceberg threshold. Setting the threshold too low will result in a huge cube, while setting the threshold too high may invalidate many useful applications. Third, an iceberg cube cannot be incrementally updated. Once an aggregate cell falls below the iceberg threshold and is pruned, its measure value is lost. Any incremental update would require recomputing the cells from scratch. This is extremely undesirable for large real-life applications where incremental appending of new data is the norm.

One possible solution, which has been implemented in some commercial data warehouse systems, is to compute a thin **cube shell**. For example, we could compute all cuboids with 3 dimensions or less in a 60-dimensional data cube, resulting in cube shell of size 3. The resulting set of cuboids would require much less computation and storage than the full 60-dimensional data cube. However, there are two disadvantages of this approach. First, we would still need to compute  $\binom{60}{3} + \binom{60}{2} + 60 = 36,050$  cuboids, each with many cells. Second, such a cube shell does not support high-dimensional OLAP because (1) it does not support OLAP on 4 or more dimensions, and (2) it cannot even support drilling along three dimensions, such as, say,  $(A_4, A_5, A_6)$ , on a subset of data selected based on the constants provided in three other dimensions, such as  $(A_1, A_2, A_3)$ . This requires the computation of the corresponding six-dimensional cuboid.

Instead of computing a cube shell, we can compute only portions or fragments of it. This section discusses the *shell fragment* approach for OLAP query processing. It is based on the following key observation about OLAP in high-dimensional space. Although a data cube may contain many dimensions, *most OLAP operations are performed on only a small number of dimensions at a time*. In other words, an OLAP query is likely to ignore many dimensions (i.e., treating them as irrelevant), fix some dimensions (e.g., using query constants as instantiations), and leave only a few to be manipulated (for drilling, pivoting, etc.). This is because it is neither realistic nor fruitful for anyone to comprehend the changes of thousands of cells involving tens of dimensions simultaneously in a high-dimensional space at the same time. Instead, it is more natural to first locate some cuboids of interest and then drill along one or two dimensions to examine the changes of a few related dimensions. Most analysts will only need to examine, at any one moment, the combinations of a small number of dimensions. This implies that if multidimensional aggregates can be computed quickly on a *small number of dimensions inside a high-dimensional space*, we may still achieve fast OLAP without materializing the original high-dimensional data cube. Computing the full cube (or, often even an iceberg cube or shell cube) can be excessive. Instead, a *semi-on-line computation model with certain pre-processing* may offer a more feasible solution. Given a base cuboid, some quick preparation computation can be done first (i.e., off-line). After that, a query can then be computed on-line using the preprocessed data.

The shell fragment approach follows such a semi-on-line computation strategy. It involves two algorithms: one for computing shell fragment cubes, and one for query processing with the fragment cubes. The shell fragment approach can handle databases of extremely high dimensionality and can quickly compute small local cubes on-line. It explores the *inverted index* data structure, which is popular in information retrieval and Web-based information systems. The basic idea is as follows. Given a high-dimensional data set, we partition the dimensions into a set of disjoint dimension *fragments*, convert each fragment into its corresponding inverted index representation, and then construct *shell fragment cubes* while keeping the inverted indices associated with the cube cells. Using the precomputed shell fragment cubes, we can dynamically assemble and compute cuboid cells of the required data cube on-line. This is made efficient by set intersection operations on the inverted indices.

To illustrate the shell fragment approach, we use the tiny database of Table 4.4 as a running example. Let the cube measure be `count()`. Other measures will be discussed later. We first look at how to construct the inverted index for the given database.

**Example 4.9 Construct the inverted index.** For each attribute value in each dimension, list the tuple identifiers (*TIDs*) of all the tuples that have that value. For example, attribute value  $a_2$  appears in tuples 4 and 5. The TID-list for  $a_2$  then contains exactly 2 items, namely 4 and 5. The resulting inverted index table is shown in Table 4.5. It retains all of the information of the original database. It uses exactly the same amount of memory as the original database.

■



<i>TID</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>
1	$a_1$	$b_1$	$c_1$	$d_1$	$e_1$
2	$a_1$	$b_2$	$c_1$	$d_2$	$e_1$
3	$a_1$	$b_2$	$c_1$	$d_1$	$e_2$
4	$a_2$	$b_1$	$c_1$	$d_1$	$e_2$
5	$a_2$	$b_1$	$c_1$	$d_1$	$e_3$

Table 4.4: The original database.

Attribute Value	Tuple ID List	List Size
$a_1$	{1, 2, 3}	3
$a_2$	{4, 5}	2
$b_1$	{1, 4, 5}	3
$b_2$	{2, 3}	2
$c_1$	{1, 2, 3, 4, 5}	5
$d_1$	{1, 3, 4, 5}	4
$d_2$	{2}	1
$e_1$	{1, 2}	2
$e_2$	{3, 4}	2
$e_3$	{5}	1

Table 4.5: The inverted index.

“How do we compute shell fragments of a data cube?” The shell fragment computation algorithm, **Frag-Shells**, is summarized in Figure 4.14. We first partition all the dimensions of the given data set into independent groups of dimensions, called **fragments** (line 1). We scan the base cuboid and construct an inverted index for each attribute (lines 2 to 6). Line 3 is for when the measure is other than the tuple `count()`, which will be described later. For each fragment, we compute the full *local* (i.e., fragment-based) data cube while retaining the inverted indices (lines 7 to 8). Consider a database of 60 dimensions, namely,  $A_1, A_2, \dots, A_{60}$ . We can first partition the 60 dimensions into 20 fragments of size 3:  $(A_1, A_2, A_3), (A_4, A_5, A_6), \dots, (A_{58}, A_{59}, A_{60})$ . For each fragment, we compute its full data cube while recording the inverted indices. For example, in fragment  $(A_1, A_2, A_3)$ , we would compute seven cuboids:  $A_1, A_2, A_3, A_1A_2, A_2A_3, A_1A_3, A_1A_2A_3$ . Furthermore, an inverted index is retained for each cell in the cuboids. That is, for each cell, its associated TID-list is recorded.

The benefit of computing local cubes of each shell fragment instead of computing the complete cube shell can be seen by a simple calculation. For a base cuboid of 60 dimensions, there are only  $7 \times 20 = 140$  cuboids to be computed according to the above shell fragment partitioning. This is in contrast to the 36,050 cuboids computed for the cube shell of size 3 described earlier! Notice that the above fragment partitioning is based simply on the grouping of consecutive dimensions. A more desirable approach would be to partition based on popular dimension groupings. Such information can be obtained from domain experts or the past history of OLAP queries.

Let’s return to our running example to see how shell fragments are computed.

**Example 4.10 Compute shell fragments.** Suppose we are to compute the shell fragments of size 3. We first divide the 5 dimensions into 2 fragments, namely  $(A, B, C)$  and  $(D, E)$ . For each fragment, we compute the full local data cube by intersecting the TID-lists in Table 4.5 in a top-down depth-first order in the cuboid lattice. For example, to compute the cell  $(a_1, b_2, *)$ , we intersect the tuple ID lists of  $a_1$  and  $b_2$  to obtain a new list of {2, 3}. Cuboid  $AB$  is shown in Table 4.6.

After computing cuboid  $AB$ , we can then compute cuboid  $ABC$  by intersecting all pairwise combinations between Table 4.6 and the row  $c_1$  in Table 4.5. Notice that because cell  $(a_2, b_2)$  is empty, it can be effectively discarded in subsequent computations, based on the Apriori property. The same process can be applied to compute fragment  $(D, E)$ , which is completely independent from computing  $(A, B, C)$ . Cuboid  $DE$  is shown in Table 4.7. ■

**Algorithm: Frag-Shells.** Compute shell fragments on a given high-dimensional base table (i.e., base cuboid).

**Input:** A base cuboid,  $B$ , of  $n$  dimensions, namely,  $(A_1, \dots, A_n)$ .

**Output:**

- a set of fragment partitions,  $\{P_1, \dots, P_k\}$ , and their corresponding (local) fragment cubes,  $\{S_1, \dots, S_k\}$ , where  $P_i$  represents some set of dimension(s) and  $P_1 \cup \dots \cup P_k$  make up all the  $n$  dimensions
- an *ID\_measure* array if the measure is not the tuple count, `count()`

**Method:**

- (1) partition the set of dimensions  $(A_1, \dots, A_n)$  into a set of  $k$  fragments  $P_1, \dots, P_k$  (based on data & query distribution)
- (2) **scan** base cuboid,  $B$ , once and do the following {
- (3)   **insert** each  $\langle TID, measure \rangle$  into *ID\_measure* array
- (4)   **for each** attribute value  $a_j$  of each dimension  $A_i$
- (5)     build an inverted index entry:  $\langle a_j, TIDlist \rangle$
- (6)   }
- (7) **for each** fragment partition  $P_i$
- (8)   build a local fragment cube,  $S_i$ , by intersecting their corresponding TID-lists and computing their measures

■

Figure 4.14: Algorithm for shell fragment computation.

Cell	Intersection	Tuple ID List	List Size
$(a_1, b_1)$	$\{1, 2, 3\} \cap \{1, 4, 5\}$	$\{1\}$	1
$(a_1, b_2)$	$\{1, 2, 3\} \cap \{2, 3\}$	$\{2, 3\}$	2
$(a_2, b_1)$	$\{4, 5\} \cap \{1, 4, 5\}$	$\{4, 5\}$	2
$(a_2, b_2)$	$\{4, 5\} \cap \{2, 3\}$	$\{\}$	0

Table 4.6: Cuboid  $AB$ .

Cell	Intersection	Tuple ID List	List Size
$(d_1, e_1)$	$\{1, 3, 4, 5\} \cap \{1, 2\}$	$\{1\}$	1
$(d_1, e_2)$	$\{1, 3, 4, 5\} \cap \{3, 4\}$	$\{3, 4\}$	2
$(d_1, e_3)$	$\{1, 3, 4, 5\} \cap \{5\}$	$\{5\}$	1
$(d_2, e_1)$	$\{2\} \cap \{1, 2\}$	$\{2\}$	1

Table 4.7: Cuboid  $DE$ .

If the measure in the iceberg condition is `count()` (as in tuple counting), there is no need to reference the original database for this since the *length* of the TID-list is equivalent to the tuple count. “Do we need to reference the original database if computing other measures, such as `average()`?” Actually, we can build and reference an *ID\_measure* array instead, which stores what we need to compute other measures. For example, to compute `average()`, we let the *ID\_measure* array hold three elements, namely,  $(TID, item\_count, sum)$ , for each cell (line 3 of the shell computation algorithm). The `average()` measure for each aggregate cell can then be computed by accessing only this *ID\_measure* array, using `sum()/item_count()`. Considering a database with  $10^6$  tuples, each taking 4 bytes each for *TID*, *item\_count* and *sum*, the *ID\_measure* array requires 12 MB, whereas the corresponding database of 60 dimensions will require  $(60 + 3) \times 4 \times 10^6 = 252$  MB (assuming each attribute value takes 4 bytes). Obviously, *ID\_measure* array is a more compact data structure and is more likely to fit in memory than the corresponding high-dimensional database.

To illustrate the design of the *ID\_measure* array, let’s look at the following example.

**Example 4.11 Computing cubes with the `average()` measure.** Suppose that Table 4.8 shows an example sales database where each tuple has two associated values, such as *item\_count* and *sum*, where *item\_count* is the count of items sold.

To compute a data cube for this database with the measure `average()` we need to have a TID-list for each

<i>TID</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>item_count</i>	<i>sum</i>
1	$a_1$	$b_1$	$c_1$	$d_1$	$e_1$	5	70
2	$a_1$	$b_2$	$c_1$	$d_2$	$e_1$	3	10
3	$a_1$	$b_2$	$c_1$	$d_1$	$e_2$	8	20
4	$a_2$	$b_1$	$c_1$	$d_1$	$e_2$	5	40
5	$a_2$	$b_1$	$c_1$	$d_1$	$e_3$	2	30

Table 4.8: A database with two measure values.

<i>TID</i>	<i>item_count</i>	<i>sum</i>
1	5	70
2	3	10
3	8	20
4	5	40
5	2	30

Table 4.9: *ID\_measure* array of Table 4.8.

cell:  $\{TID_1, \dots, TID_n\}$ . Because each TID is uniquely associated with a particular set of measure values, all future computations just need to fetch the measure values associated with the tuples in the list. In other words, by keeping an *ID\_measure* array in memory for on-line processing, we can handle complex algebraic measures, such as average, variance, and standard deviation. Table 4.9 shows what exactly should be kept for our example, which is substantially smaller than the database itself. ■

The shell fragments are negligible in both storage space and computation time in comparison with the full data cube. Note that we can also use the *Frag-Shells* algorithm to compute the full data cube by including all of the dimensions as a single fragment. Since the order of computation with respect to the cuboid lattice is top-down and depth-first (similar to that of BUC), the algorithm can perform Apriori pruning if applied to the construction of iceberg cubes.

“Once we have computed the shell fragments, how can they be used to answer OLAP queries?” Given the precomputed shell fragments, we can view the cube space as a virtual cube and perform OLAP queries related to the cube on-line. In general, there are two types of queries: (1) *point query* and (2) *subcube query*.

In a **point query**, all of the *relevant* dimensions in the cube have been instantiated (that is, there are no *inquired* dimensions in the relevant set of dimensions). For example, in an  $n$ -dimensional data cube,  $A_1 A_2 \dots A_n$ , a point query could be in the form of  $\langle A_1, A_5, A_9 : M? \rangle$ , where  $A_1 = \{a_{11}, a_{18}\}$ ,  $A_5 = \{a_{52}, a_{55}, a_{59}\}$ ,  $A_9 = a_{94}$ , and  $M$  is the inquired measure for each corresponding cube cell. For a cube with a small number of dimensions, we can use “\*” to represent a “don’t care” position where the corresponding dimension is *irrelevant*, that is, neither inquired nor instantiated. For example, in the query  $\langle a_2, b_1, c_1, d_1, * : \text{count}()? \rangle$  for the database in Table 4.4, the first four dimension values are instantiated to  $a_2$ ,  $b_1$ ,  $c_1$ , and  $d_1$ , respectively, while the last dimension is irrelevant, and  $\text{count}()$  (which is the tuple count by context) is the inquired measure.

In a **subcube query**, at least one of the *relevant* dimensions in the cube is *inquired*. For example, in an  $n$ -dimensional data cube  $A_1 A_2 \dots A_n$ , a subcube query could be in the form  $\langle A_1, A_5?, A_9, A_{21}? : M? \rangle$ , where  $A_1 = \{a_{11}, a_{18}\}$  and  $A_9 = a_{94}$ ,  $A_5$  and  $A_{21}$  are the inquired dimensions, and  $M$  is the inquired measure. For a cube with a small number of dimensions, we can use “\*” for an irrelevant dimension and “?” for an inquired one. For example, in the query  $\langle a_2, ?, c_1, *, ? : \text{count}()? \rangle$  we see that the first and third dimension values are instantiated to  $a_2$  and  $c_1$ , respectively, while the fourth is irrelevant, and the second and the fifth are inquired. A *subcube query* computes all possible value combinations of the inquired dimensions. It essentially returns a local data cube consisting of the inquired dimensions.

“How can we use shell fragments to answer a point query?” Since a point query explicitly provides the set of instantiated variables on the set of relevant dimensions, we can make maximal use of the precomputed shell fragments by finding the *best fitting* (that is, *dimension-wise completely matching*) fragments to fetch and intersect

the associated TID-lists.

Let the point query be of the form  $\langle \alpha_i, \alpha_j, \alpha_k, \alpha_p : M^? \rangle$ , where  $\alpha_i$  represents a set of instantiated values of dimension  $A_i$ , and so on for  $\alpha_j, \alpha_k$ , and  $\alpha_p$ . First, we check the shell fragment schema to determine which dimensions among  $A_i, A_j, A_k$  and  $A_p$  are in the same fragment(s). Suppose  $A_i$  and  $A_j$  are in the same fragment, while  $A_k$ , and  $A_p$  are in two other fragments. We fetch the corresponding TID-lists on the precomputed 2-D fragment for dimensions  $A_i$  and  $A_j$  using the instantiations  $\alpha_i$  and  $\alpha_j$ , and fetch the TID-lists on the 1-D fragments for dimensions  $A_k$  and  $A_p$  using the instantiations  $\alpha_k$  and  $\alpha_p$ , respectively. The obtained TID-lists are intersected to derive the TID-list table. This table is then used to derive the specified measure (e.g., by taking the length of the TID-lists for tuple `count()`, or by fetching `item_count()` and `sum()` from the *ID\_measure* array to compute `average()`) for the final set of cells.

**Example 4.12 Point query.** Suppose a user wants to compute the point query,  $\langle a_2, b_1, c_1, d_1, * : \text{count}()^? \rangle$ , for our database in Table 4.4 and that the shell fragments for the partitions  $(A, B, C)$  and  $(D, E)$  are precomputed as described in Example 4.10. The query is broken down into two subqueries based on the precomputed fragments:  $\langle a_2, b_1, c_1, *, * \rangle$  and  $\langle *, *, *, d_1, * \rangle$ . The best fit precomputed shell fragments for the two subqueries are  $ABC$  and  $D$ . The fetch of the TID-lists for the two subqueries returns two lists:  $\{4, 5\}$  and  $\{1, 3, 4, 5\}$ . Their intersection is the list  $\{4, 5\}$ , which is of size 2. Thus the final answer is `count() = 2`. ■

A subcube query returns a local data cube based on the instantiated and inquired dimensions. Such a data cube needs to be aggregated in a multidimensional way so that on-line analytical processing (such as drilling, dicing, pivoting, etc.) can be made available to users for flexible manipulation and analysis. Since instantiated dimensions usually provide highly selective constants that dramatically reduce the size of the valid TID-lists, we should make maximal use of the precomputed shell fragments by finding the fragments that best fit the set of instantiated dimensions, and fetching and intersecting the associated TID-lists to derive the reduced TID-list. This list can then be used to intersect the best fitting shell fragments consisting of the inquired dimensions. This will generate the relevant and inquired base cuboid, which can then be used to compute the relevant subcube on the fly using an efficient on-line cubing algorithm.

Let the subcube query be of the form  $\langle \alpha_i, \alpha_j, A_k?, \alpha_p, A_q? : M^? \rangle$ , where  $\alpha_i, \alpha_j$  and  $\alpha_p$  represent a set of instantiated values of dimension  $A_i, A_j$ , and  $A_p$ , respectively, and  $A_k$  and  $A_q$  represent two inquired dimensions. First, we check the shell fragment schema to determine which dimensions among (1)  $A_i, A_j$ , and  $A_p$ , and (2) among  $A_k$  and  $A_q$  are in the same fragment partition. Suppose  $A_i$  and  $A_j$  belong to the same fragment, as do  $A_k$  and  $A_q$ , but that  $A_p$  is in a different fragment. We fetch the corresponding TID-lists in the pre-computed 2-D fragment for  $A_i$  and  $A_j$  using the instantiations  $\alpha_i$  and  $\alpha_j$ , then fetch the TID-list on the precomputed 1-D fragment for  $A_p$  using instantiation  $\alpha_p$ , and then fetch the TID-lists on the pre-computed 1-D fragments for  $A_k$  and  $A_q$ , respectively, using no instantiations (i.e., all possible values). The obtained TID-lists are intersected to derive the final TID-lists, which are used to fetch the corresponding measures from the *ID\_measure* array to derive the “base cuboid” of a 2-D subcube for 2 dimensions  $(A_k, A_q)$ . A fast cube computation algorithm can be applied to compute this 2-D cube based on the derived base cuboid. The computed 2-D cube is then ready for OLAP operations.

**Example 4.13 Subcube query.** Suppose a user wants to compute the subcube query,  $\langle a_2, b_1, ?, *, ? : \text{count}()^? \rangle$ , for our database in Table 4.4, and that the shell fragments have been precomputed as described in Example 4.10. The query can be broken into three best fit fragments according to the instantiated and inquired dimensions:  $AB, C$ , and  $E$ , where  $AB$  has the instantiation  $(a_2, b_1)$ . The fetch of the TID-lists for these partitions returns:  $(a_2, b_1): \{4, 5\}$ ,  $(c_1): \{1, 2, 3, 4, 5\}$ , and  $\{(e_1: \{1, 2\}), (e_2: \{3, 4\}), (e_3: \{5\})\}$ , respectively. The intersection of these corresponding TID-lists contains a cuboid with two tuples:  $\{(c_1, e_2): \{4\}^6, (c_1, e_3): \{5\}\}$ . This base cuboid can be used to compute the 2-D data cube, which is trivial. ■

For large data sets, a fragment size of 2 or 3 typically results in reasonable storage requirements for the shell fragments and for fast query response time. Querying with shell fragments is substantially faster than answering

<sup>6</sup>That is, the intersection of the TID-lists for  $(a_2, b_1)$ ,  $(c_1)$  and  $(e_2)$  is  $\{4\}$ .

queries using precomputed data cubes that are stored on disk. In comparison to full cube computation, **Frag-Shells** is recommended if there are less than four inquired dimensions. Otherwise, more efficient algorithms, such as **Star-Cubing**, can be used for fast on-line cube computation. **Frag-Shells** can easily be extended to allow incremental updates, the details of which are left as an exercise.

#### 4.1.6 Computing Cubes with Complex Iceberg Conditions

The iceberg cubes we have discussed so far contain only simple iceberg conditions, such as  $count \geq 50$  or  $price\_sum \geq 1000$  (specified in the **having** clause). Such conditions have a nice property: *if [OLD: an iceberg][NEW: the] condition is violated for some cell  $c$ , then every descendant of  $c$  will also violate that condition*. For example, if the quantity of an item  $I$  sold in a region  $R_1$  is less than 50, then the same item  $I$  sold in a subregion of  $R_1$  can never satisfy the condition  $count \geq 50$ . Conditions that obey this property are known as antimonotonic.

Not all iceberg conditions are antimonotonic. For example, the condition  $avg(price) \geq 800$  is not antimonotonic. This is because if the average price of an item, such as, say, “TV”, in region  $R_1$ , is less than \$800, then a descendant of the cell representing “TV” and  $R_1$ , such as “TV” in a subregion of  $R_1$ , can still have an average price of over \$800.

“Can we still push such an iceberg condition deep into the cube computation process for improved efficiency?” To answer this question, we first look at an example.

**Example 4.14 Iceberg cube with the average measure.** Consider the *salesInfo* table given in Table 4.10, which registers sales related to month, day, city, customer group, item, and price.

month	day	city	cust_group	item	price
Jan	10	Chicago	Education	HP Printer	485
Jan	15	Chicago	Household	Sony TV	1,200
Jan	20	New York	Education	Canon Camera	1,280
Feb	20	New York	Business	IBM Laptop	2,500
Mar	4	Vancouver	Education	Seagate HD	520
...	...	...	...	...	...

Table 4.10: A *salesInfo* table.

Suppose, as data analysts, we have the following query: *Find groups of sales that contain at least 50 items and whose average item price is at least \$800, grouped by month, city, and/or customer group*. We can specify an iceberg cube, *sales\_avg\_iceberg*, to answer the query, as follows.

```
compute cube sales_avg_iceberg as
select month, city, customer_group, avg(price), count(*)
from salesInfo
cube by month, city, customer_group
having avg(price) >= 800 and count(*) >= 50
```

Here, the iceberg condition involves the measure *average*, which is not antimonotonic. This implies that if a cell,  $c$ , cannot satisfy the iceberg condition, “ $average(c) \geq v$ ”, we cannot prune away the descendants of  $c$  because it is possible that the average value for some of them may satisfy the condition. ■

“How can we compute *sales\_avg\_iceberg*?” It would be highly inefficient to first materialize the full data cube and then select the cells satisfying the **having** clause of the iceberg condition. We have seen that a cube with an antimonotonic iceberg condition can be computed efficiently by exploring the Apriori property. However, since this iceberg cube involves a non-antimonotonic iceberg condition, Apriori pruning cannot be applied. “Can we transform the non-antimonotonic condition to a somewhat weaker but antimonotonic one so that we can still take advantage of pruning?”

The answer is “yes”. Here we examine one interesting such method. A cell  $c$  is said to have  $n$  base cells if it covers  $n$  nonempty descendant base cells. The **top- $k$  average** of  $c$ , denoted as  $avg^k(c)$ , is the average value (i.e., price) of the *top- $k$  base cells* of  $c$  (i.e., the first  $k$  cells when all the base cells in  $c$  are sorted in value-descending order) if  $k \leq n$ ; or  $-\infty$  if  $k > n$ . With this notion of top- $k$  average, we can transform the original iceberg condition “ $avg(price) \geq v$  and  $count(*) \geq k$ ” into the weaker but antimonotonic condition “ $avg^k(c) \geq v$ ”. The reasoning is that if the average of the top- $k$  nonempty descendant base cells of a cell  $c$  is less than  $v$ , there exists no subset from this set of base cells that can contain  $k$  or more base cells and have a bigger average value than  $v$ . Thus, it is safe to prune away the cell  $c$ .

It is costly to sort and keep the top- $k$  base cell values for each aggregated cell. For efficient implementation, we can use only a few records to register some aggregated values to facilitate similar pruning. For example, we could use one record,  $r_0$ , to keep the sum and count of the cells whose value is no less than  $v$ , and a few records, such as  $r_1$ ,  $r_2$ , and  $r_3$ , to keep the sum and count of the cells whose price falls into the range of  $[0.8 - 1.0)$ ,  $[0.6 - 0.8)$ ,  $[0.4 - 0.6)$  of  $v$ , respectively. If the counts of  $r_0$  and  $r_1$  are no less than  $k$  but the average of the two is less than  $v$ , there is no hope of finding any descendants of  $c$  that can satisfy the iceberg condition. Thus  $c$  and its descendants can be pruned off in iceberg cube computation.

Similar transformation methods can be applied to many other iceberg conditions, such as those involving *average* on a set of positive and negative values, *range*, *variance*, and *standard deviation*. Details of the transformation methods are left as an exercise for interested readers.

## 4.2 Further Development of Data Cube and OLAP Technology

In this section, we study further developments of data cube and OLAP technology. Section 4.2.1 describes data mining by *discovery-driven exploration of data cubes*, where anomalies in the data are automatically detected and marked for the user with visual cues. Section 4.2.2 describes *multifeature cubes* for complex data mining queries involving multiple dependent aggregates at multiple granularity. Section 4.2.3 presents methods for *constrained gradient analysis* in data cubes, which identifies cube cells that have dramatic changes in value in comparison with their siblings, ancestors, or descendants.

### 4.2.1 Discovery-Driven Exploration of Data Cubes

As studied in previous sections, a data cube may have a large number of cuboids, and each cuboid may contain a large number of (aggregate) cells. With such an overwhelmingly large space, it becomes a burden for users to even just browse a cube, let alone think of exploring it thoroughly. Tools need to be developed to assist users in intelligently exploring the huge aggregated space of a data cube.

**Discovery-driven exploration** is such a cube exploration approach. In discovery-driven exploration, precomputed measures indicating data exceptions are used to guide the user in the data analysis process, at all levels of aggregation. We hereafter refer to these measures as *exception indicators*. Intuitively, an **exception** is a data cube cell value that is significantly different from the value anticipated, based on a statistical model. The model considers variations and patterns in the measure value across *all of the dimensions* to which a cell belongs. For example, if the analysis of *item-sales* data reveals an increase in sales in December in comparison to all other months, this may seem like an exception in the time dimension. However, it is not an exception if the item dimension is considered, since there is a similar increase in sales for other items during December. The model considers exceptions hidden at all aggregated group-by's of a data cube. Visual cues such as background color are used to reflect the degree of exception of each cell, based on the precomputed exception indicators. Efficient algorithms have been proposed for cube construction, as discussed in Section 4.1. The computation of exception indicators can be overlapped with cube construction, so that the overall construction of data cubes for discovery-driven exploration is efficient.

Three measures are used as exception indicators to help identify data anomalies. These measures indicate the degree of surprise that the quantity in a cell holds, with respect to its expected value. The measures are computed and associated with every cell, for all levels of aggregation. They are

- **SelfExp**: This indicates the degree of surprise of the cell value, relative to other cells at the same level of aggregation.
- **InExp**: This indicates the degree of surprise somewhere beneath the cell, if we were to drill down from it.
- **PathExp**: This indicates the degree of surprise for each drill-down path from the cell.

The use of these measures for discovery-driven exploration of data cubes is illustrated in the following example.

**Example 4.15 Discovery-driven exploration of a data cube.** Suppose that you would like to analyze the monthly sales at *AllElectronics* as a percentage difference from the previous month. The dimensions involved are *item*, *time*, and *region*. You begin by studying the data aggregated over all items and sales regions for each month, as shown in Figure 4.15.

Sum of sales	Month											
	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Total		1%	-1%	0%	1%	3%	-1%	-9%	-1%	2%	-4%	3%

Figure 4.15: Change in sales over time.

To view the exception indicators, you would click on a button marked **highlight exceptions** on the screen. This translates the SelfExp and InExp values into visual cues, displayed with each cell. The background color of each cell is based on its SelfExp value. In addition, a box is drawn around each cell, where the thickness and color of the box are a function of its InExp value. Thick boxes indicate high InExp values. In both cases, the darker the color, the greater the degree of exception. For example, the dark, thick boxes for sales during July, August, and September signal the user to explore the lower-level aggregations of these cells by drilling down.

Drill-downs can be executed along the aggregated *item* or *region* dimensions. “Which path has more exceptions?” you wonder. To find this out, you select a cell of interest and trigger a path exception module that colors each dimension based on the PathExp value of the cell. This value reflects the degree of surprise of that path. Suppose that the path along *item* contains more exceptions.

Avg. sales	Month											
Item	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Sony b/w printer		9%	-8%	2%	-5%	14%	-4%	0%	41%	-13%	-15%	-11%
Sony color printer		0%	0%	3%	2%	4%	-10%	-13%	0%	4%	-6%	4%
HP b/w printer		-2%	1%	2%	3%	8%	0%	-12%	-9%	3%	-3%	6%
HP color printer		0%	0%	-2%	1%	0%	-1%	-7%	-2%	1%	-4%	1%
IBM desktop computer		1%	-2%	-1%	-1%	3%	3%	-10%	4%	1%	-4%	-1%
IBM laptop computer		0%	0%	-1%	3%	4%	2%	-10%	-2%	0%	-9%	3%
Toshiba desktop computer		-2%	-5%	1%	1%	-1%	1%	5%	-3%	-5%	-1%	-1%
Toshiba laptop computer		1%	0%	3%	0%	-2%	-2%	-5%	3%	2%	-1%	0%
Logitech mouse		3%	-2%	-1%	0%	4%	6%	-11%	2%	1%	-4%	0%
Ergo-way mouse		0%	0%	2%	3%	1%	-2%	-2%	-5%	0%	-5%	8%

Figure 4.16: Change in sales for each *item-time* combination.

A drill-down along *item* results in the cube slice of Figure 4.16, showing the sales over time for each item. At this point, you are presented with many different sales values to analyze. By clicking on the **highlight exceptions** button, the visual cues are displayed, bringing focus towards the exceptions. Consider the sales difference of 41% for “Sony b/w printers” in September. This cell has a dark background, indicating a high SelfExp value, meaning

that the cell is an exception. Consider now the the sales difference of  $-15\%$  for “Sony b/w printers” in November, and of  $-11\%$  in December. The  $-11\%$  value for December is marked as an exception, while the  $-15\%$  value is not, even though  $-15\%$  is a bigger deviation than  $-11\%$ . This is because the exception indicators consider all of the dimensions that a cell is in. Notice that the December sales of most of the other items have a large positive value, while the November sales do not. Therefore, by considering the position of the cell in the cube, the sales difference for “Sony b/w printers” in December is exceptional, while the November sales difference of this item is not.

The InExp values can be used to indicate exceptions at lower levels that are not visible at the current level. Consider the cells for “IBM desktop computers” in July and September. These both have a dark, thick box around them, indicating high InExp values. You may decide to further explore the sales of “IBM desktop computers” by drilling down along *region*. The resulting sales difference by *region* is shown in Figure 4.17, where the highlight exceptions option has been invoked. The visual cues displayed make it easy to instantly notice an exception for the sales of “IBM desktop computers” in the southern region, where such sales have decreased by  $-39\%$  and  $-34\%$  in July and September, respectively. These detailed exceptions were far from obvious when we were viewing the data as an *item-time* group-by, aggregated over *region* in Figure 4.16. Thus, the InExp value is useful for searching for exceptions at lower-level cells of the cube. Since there are no other cells in Figure 4.17 having a high InExp value, you may roll up back to the data of Figure 4.16, and choose another cell from which to drill down. In this way, the exception indicators can be used to guide the discovery of interesting anomalies in the data. ■

Avg. sales	Month											
Region	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
North		-1%	-3%	-1%	0%	3%	4%	-7%	1%	0%	-3%	-3%
South		-1%	1%	-9%	6%	-1%	-39%	9%	-34%	4%	1%	7%
East		-1%	-2%	2%	-3%	1%	18%	-2%	11%	-3%	-2%	-1%
West		4%	0%	-1%	-3%	5%	1%	-18%	8%	5%	-8%	1%

Figure 4.17: Change in sales for the item *IBM desktop computer* per region.

“How are the exception values computed?” The SelfExp, InExp, and PathExp measures are based on a statistical method for table analysis. They take into account all of the group-by’s (aggregations) in which a given cell value participates. A cell value is considered an exception based on how much it differs from its expected value, where its expected value is determined with a statistical model described below. The difference between a given cell value and its expected value is called a **residual**. Intuitively, the larger the residual, the more the given cell value is an exception. The comparison of residual values requires us to scale the values based on the expected standard deviation associated with the residuals. A cell value is therefore considered an exception if its scaled residual value exceeds a prespecified threshold. The SelfExp, InExp, and PathExp measures are based on this scaled residual.

The expected value of a given cell is a function of the higher-level group-by’s of the given cell. For example, given a cube with the three dimensions,  $A$ ,  $B$ , and  $C$ , the expected value for a cell at the  $i$ th position in  $A$ , the  $j$ th position in  $B$ , and the  $k$ th position in  $C$  is a function of  $\gamma, \gamma_i^A, \gamma_j^B, \gamma_k^C, \gamma_{ij}^{AB}, \gamma_{ik}^{AC},$  and  $\gamma_{jk}^{BC}$ , which are coefficients of the statistical model used. The coefficients reflect how different the values at more detailed levels are, based on generalized impressions formed by looking at higher-level aggregations. In this way, the exception quality of a cell value is based on the exceptions of the values below it. Thus, when seeing an exception, it is natural for the user to further explore the exception by drilling down.

“How can the data cube be efficiently constructed for discovery-driven exploration?” This computation consists of three phases. The first step involves the computation of the aggregate values defining the cube, such as **sum** or **count**, over which exceptions will be found. The second phase consists of model fitting, in which the coefficients mentioned above are determined and used to compute the standardized residuals. This phase can be overlapped with the first phase since the computations involved are similar. The third phase computes the SelfExp, InExp, and PathExp values, based on the standardized residuals. This phase is computationally similar to phase 1. Therefore, the computation of data cubes for discovery-driven exploration can be done efficiently.



### 4.2.2 Complex Aggregation at Multiple Granularity: Multifeature Cubes

Data cubes facilitate the answering of data mining queries as they allow the computation of aggregate data at multiple levels of granularity. In this section, you will learn about *multifeature cubes*, which compute complex queries involving multiple dependent aggregates at multiple granularity. These cubes are very useful in practice. Many complex data mining queries can be answered by multifeature cubes without any significant increase in computational cost, in comparison to cube computation for simple queries with standard data cubes.

All of the examples in this section are from the Purchases data of *AllElectronics*, where an *item* is purchased in a sales *region* on a business day (*year, month, day*). The shelf life in months of a given item is stored in *shelf*. The item price and sales (in dollars) at a given region are stored in *price* and *sales*, respectively. To aid in our study of multifeature cubes, let's first look at an example of a simple data cube.

**Example 4.16 Query 1: A simple data cube query.** Find the total sales in 2004, broken down by *item*, *region*, and *month*, with subtotals for each dimension.

To answer Query 1, a data cube is constructed that aggregates the total sales at the following eight different levels of granularity:  $\{(item, region, month), (item, region), (item, month), (month, region), (item), (month), (region), ()\}$ , where  $()$  represents all. Query 1 uses a typical data cube like that introduced in the previous chapter. We call such a data cube a simple data cube since it does not involve any dependent aggregates. ■

“What is meant by ‘dependent aggregates’?” We answer this by studying the following example of a complex query.

**Example 4.17 Query 2: A complex query.** Grouping by all subsets of  $\{item, region, month\}$ , find the maximum *price* in 2004 for each group, and the total *sales* among all maximum price tuples.

The specification of such a query using standard SQL can be long, repetitive, and difficult to optimize and maintain. Alternatively, Query 2 can be specified concisely using an extended SQL syntax as follows:

```
select    item, region, month, max(price), sum(R.sales)
from      Purchases
where     year = 2004
cube by   item, region, month: R
such that R.price = max(price)
```

The tuples representing purchases in 2004 are first selected. The **cube by** clause computes aggregates (or group-by's) for all possible combinations of the attributes *item*, *region*, and *month*. It is an  $n$ -dimensional generalization of the **group by** clause. The attributes specified in the **cube by** clause are the **grouping attributes**. Tuples with the same value on all grouping attributes form one group. Let the groups be  $g_1, \dots, g_r$ . For each group of tuples  $g_i$ , the maximum price  $max_{g_i}$  among the tuples forming the group is computed. The variable  $R$  is a **grouping variable**, ranging over all tuples in group  $g_i$  whose price is equal to  $max_{g_i}$  (as specified in the **such that** clause). The sum of sales of the tuples in  $g_i$  that  $R$  ranges over is computed and returned with the values of the grouping attributes of  $g_i$ . The resulting cube is a **multifeature cube** in that it supports complex data mining queries for which multiple dependent aggregates are computed at a variety of granularities. For example, the sum of sales returned in Query 2 is dependent on the set of maximum price tuples for each group. ■

Let's look at another example.

**Example 4.18 Query 3: An even more complex query.** Grouping by all subsets of  $\{item, region, month\}$ , find the maximum *price* in 2004 for each group. Among the maximum price tuples, find the minimum and maximum item *shelf lives*. Also find the fraction of the total *sales* due to tuples that have minimum shelf life within the set of all maximum price tuples, and the fraction of the total *sales* due to tuples that have maximum shelf life within the set of all maximum price tuples.

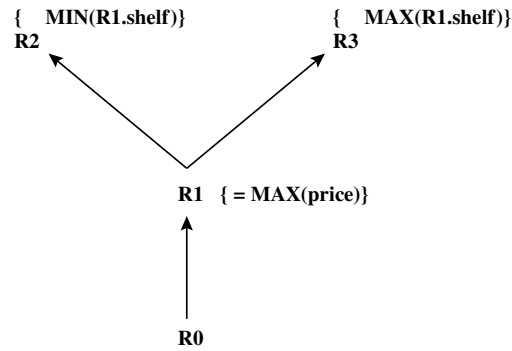


Figure 4.18: A multifeature cube graph for Query 3.

The **multifeature cube graph** of Figure 4.18 helps illustrate the aggregate dependencies in the query. There is one node for each grouping variable, plus an additional initial node, R0. Starting from node R0, the set of maximum price tuples in 2004 is first computed (node R1). The graph indicates that grouping variables R2 and R3 are “dependent” on R1, since a directed line is drawn from R1 to each of R2 and R3. In a multifeature cube graph, a directed line from grouping variable  $R_i$  to  $R_j$  means that  $R_j$  always ranges over a subset of the tuples that  $R_i$  ranges over. When expressing the query in extended SQL, we write “ $R_j$  in  $R_i$ ” as shorthand to refer to this case. For example, the minimum shelf life tuples at R2 range over the maximum price tuples at R1, that is, “R2 in R1”. Similarly, the maximum shelf life tuples at R3 range over the maximum price tuples at R1, that is, “R3 in R1”.

From the graph, we can express Query 3 in extended SQL as follows:

```

select    item, region, month, max(price), min(R1.shelf), max(R1.shelf),
          sum(R1.sales), sum(R2.sales), sum(R3.sales)
from      Purchases
where     year = 2004
cube by   item, region, month: R1, R2, R3
such that R1.price = max(price) and
          R2 in R1 and R2.shelf = min(R1.shelf) and
          R3 in R1 and R3.shelf = max(R1.shelf)

```

■

“How can multifeature cubes be computed efficiently?” The computation of a multifeature cube depends on the types of aggregate functions used in the cube. In Chapter 3, we saw that aggregate functions can be categorized as either distributive, algebraic, or holistic. Multifeature cubes can be organized into the same categories and computed efficiently by minor extension of the previously studied cube computation methods.

### 4.2.3 Constrained Gradient Analysis in Data Cubes

Many data cube applications need to analyze the *changes of complex measures* in multidimensional space. For example, in real estate, we may want to ask what are the *changes* of the *average* house price in the Vancouver area in the year 2004 compared against 2003, and the answer could be “the average price for those sold to professionals in the West End went down by 20%, while those sold to business people in Metrotown went up by 10%, etc.” Expressions such as “professionals in the West End” correspond to cuboid cells and describe *sectors* of the business modeled by the data cube.

The problem of mining *changes of complex measures* in a multidimensional space was first proposed by Imielinski, Khachiyan, and Abdulghani as the **cubegrade** problem, which can be viewed as a generalization of association

rules<sup>7</sup> and data cubes. It studies how changes in a set of measures (aggregates) of interest are associated with changes in the underlying characteristics of sectors, where changes in sector characteristics are expressed in terms of dimensions of the cube and are limited to *specialization* (drill-down), *generalization* (roll-up), and *mutation* (a change in one of the cube's dimensions). For example, we may want to ask “what kind of sector characteristics are associated with major changes in average house price in the Vancouver area in 2004?” The answer will be *pairs of sectors*, associated with major changes in average house price, including for example “the sector of professional buyers in the West End area of Vancouver” versus “the sector of all buyers in the entire area of Vancouver” as a specialization (or generalization). The cubegrade problem is significantly more expressive than association rules since it captures data trends and handles complex measures, not just **count**, as association rules do. The problem has broad applications, from trend analysis to answering “what-if” questions and discovering exceptions or outliers.

The curse of dimensionality and the need for understandable results pose serious challenges for finding an efficient and scalable solution to the cubegrade problem. Here we examine a confined but interesting version of the cubegrade problem, called **constrained multidimensional gradient analysis**, which reduces the search space and derives interesting results. It incorporates the following types of constraints:

1. **Significance constraint:** This ensures that we examine only the cells that have certain “statistical significance” in the data, such as containing at least a specified number of base cells or at least a certain total sales. In the data cube context, this constraint acts as the iceberg condition, which prunes a huge number of trivial cells from the answer set.
2. **Probe constraint:** This selects a subset of cells (called **probe cells**) from all of the possible cells as starting points for examination. Since the cubegrade problem needs to compare each cell in the cube with other cells that are either specializations, generalizations, or mutations of the given cell, it extracts pairs of similar cell characteristics associated with big changes in measure in a data cube. Given two cells, **a**, **b**, and **c**, if **a** is a specialization of **b**, then we say it is an **descendant** of **b**, in which case, **b** is a generalization or **ancestor** of **a**. Cell **c** is a mutation of **a** if the two have identical values in all but one dimension, where the dimension for which they vary cannot have a value of “\*”. Cells **a** and **c** are considered **siblings**. Even when considering only iceberg cubes, a large number of pairs may still be generated. Probe constraints allow the user to specify a subset of cells that are of interest for the analysis task. In this way, the study is focused only on these cells and their relationships with corresponding ancestors, descendants, and siblings.
3. **Gradient constraint:** This specifies the user's range of interest on the gradient (measure change). A user is typically interested in only certain types of changes between the cells (sectors) under comparison. For example, we may be interested in only those cells whose average profit increases by more than 40% compared to that of the probe cells. Such changes can be specified as a threshold in the form of either a ratio or a difference between certain measure values of the cells under comparison. A cell that captures the change from the probe cell is referred to as a **gradient cell**.

The following example illustrates each of the above types of constraints.

$c_1$	(2000, Vancouver, Business, PC, 300, \$2100)
$c_2$	(*, Vancouver, Business, PC, 2800, \$1900)
$c_3$	(*, Toronto, Business, PC, 7900, \$2350)
$c_4$	(*, *, Business, PC, 58600, \$2250)

Table 4.11: A set of base and aggregate cells.

**Example 4.19 Constrained average gradient analysis.** The base table,  $D$ , for *AllElectronics* sales has the schema

$sales(year, city, customer\_group, item\_group, count, avg\_price).$

<sup>7</sup>Association rules were introduced in Chapter 1. They are often used in market basket analysis to find associations between items purchased in transactional sales databases. Association rule mining is described in detail in Chapter 5.

Attributes *year*, *city*, *customer\_group*, and *item\_group* are the *dimensional attributes*; and *count* and *avg\_price* are the *measure attributes*. Table 4.11 shows a set of base and aggregate cells. Tuple  $c_1$  is a base cell, while tuples  $c_2$ ,  $c_3$ , and  $c_4$  are aggregate cells. Tuple  $c_3$  is a sibling of  $c_2$ ,  $c_4$  is an ancestor of  $c_2$ , and  $c_1$  is a descendant of  $c_2$ .

Suppose that the significance constraint,  $C_{sig}$ , is ( $count \geq 100$ ), meaning that a cell with *count* no less than 100 is regarded as significant. Suppose that the probe constraint,  $C_{prb}$ , is ( $city = \text{"Vancouver"} , customer\_group = \text{"Business"} , item\_group = *$ ). This means that the set of *probe cells*,  $P$ , is the set of aggregate tuples regarding the sales of the Business customer group in Vancouver, for every product group, provided the *count* in the tuple is greater than or equal to 100. It is easy to see that  $c_2 \in P$ .

Let the gradient constraint,  $C_{grad}(c_g, c_p)$  be ( $avg\_price(c_g)/avg\_price(c_p) \geq 1.4$ ). The constrained gradient analysis problem is thus to find all pairs,  $(c_g, c_p)$ , where  $c_p$  is a probe cell in  $P$ ;  $c_g$  is a sibling, ancestor, or descendant of  $c_p$ ;  $c_g$  is a significant cell; and  $c_g$ 's average price is at least 40% more than  $c_p$ 's. ■

If a data cube is fully materialized, the query posed in Example 4.19 becomes a relatively simple retrieval of the pairs of computed cells that satisfy the constraints. Unfortunately, the number of aggregate cells is often too huge to be precomputed and stored. Typically, only the base table or cuboid is available so that the task then becomes how to efficiently compute the gradient-probe pairs from it.

One rudimentary approach to computing such gradients is to conduct a search for the gradient cells, once per probe cell. This approach is inefficient because it would involve a large amount of repeated work for different probe cells. A suggested method is a set-oriented approach that starts with a set of probe cells, utilizes constraints early on during search and explores pruning, when possible, during progressive computation of pairs of cells. With each gradient cell, the set of all possible probe cells that might co-occur in interesting gradient-probe pairs are associated with some descendants of the gradient cell. These probe cells are considered “live probe cells”. This set is used to search for future gradient cells, while considering significance constraints and gradient constraints to reduce the search space as follows:

1. The significance constraints can be used directly for pruning: If a cell,  $c$ , cannot satisfy the significance constraint, then  $c$  and its descendants can be pruned since none of them can be significant; and
2. Since the gradient constraint may specify a complex measure (such as  $avg \geq v$ ), the incorporation of both the significance constraint and the gradient constraint can be used for pruning in a manner similar to that discussed in Section 4.1.6 on computing cubes with complex iceberg conditions. That is, we can explore a weaker but antimonotonic form of the constraint, such as the *top-k average*,  $avg^k(c) \geq v$ , where  $k$  is the significance constraint (such as 100 in Example 4.19), and  $v$  is derived from the gradient constraint based on  $v = c_g \times v_p$ , where  $c_g$  is the *gradient\_constraint\_threshold*, and  $v_p$  is the value of the corresponding probe cell. That is, if the current cell,  $c$ , cannot satisfy this constraint, further exploration of its descendants will be useless and thus can be pruned.

The *constrained cube gradient analysis* has been shown to be effective at exploring the significant changes among related cube cells in multidimensional space.

### 4.3 Attribute-Oriented Induction—An Alternative Method for Data Generalization and Concept Description

*Data generalization* summarizes data by replacing relatively low level values (such as numeric values for an attribute *age*) with higher level concepts (such as *young*, *middle-aged*, and *senior*). Given the large amount of data stored in databases, it is useful to be able to describe concepts in concise and succinct terms at generalized (rather than low) levels of abstraction. Allowing data sets to be generalized at multiple levels of abstraction facilitates users in examining the general behavior of the data. Given the *AllElectronics* database, for example, instead of examining individual customer transactions, sales managers may prefer to view the data generalized to higher levels, such as

summarized by customer groups according to geographic regions, frequency of purchases per group, and customer income.

This leads us to the notion of *concept description*, which is a form of data generalization. A concept typically refers to a collection of data such as *frequent\_buyers*, *graduate\_students*, and so on. As a data mining task, concept description is not a simple enumeration of the data. Instead, **concept description** generates descriptions for the *characterization* and *comparison* of the data. It is sometimes called **class description**, when the concept to be described refers to a class of objects. **Characterization** provides a concise and succinct summarization of the given collection of data, while concept or class **comparison** (also known as **discrimination**) provides descriptions comparing two or more collections of data.

Up to this point, we have studied data cube (or OLAP) approaches to concept description using multidimensional, multilevel data generalization in data warehouses. “*Is data cube technology sufficient to accomplish all kinds of concept description tasks for large data sets?*” Consider the following cases.

- **Complex data types and aggregation:** Data warehouses and OLAP tools are based on a multidimensional data model that views data in the form of a data cube, consisting of dimensions (or attributes) and measures (aggregate functions). However, many current OLAP systems confine dimensions to nonnumeric data and measures to numeric data. In reality, the database can include attributes of various data types, including numeric, nonnumeric, spatial, text, or image, which ideally should be included in the concept description. Furthermore, the aggregation of attributes in a database may include sophisticated data types, such as the collection of nonnumeric data, the merging of spatial regions, the composition of images, the integration of texts, and the grouping of object pointers. Therefore, OLAP, with its restrictions on the possible dimension and measure types, represents a simplified model for data analysis. Concept description should handle complex data types of the attributes and their aggregations, as necessary.
- **User-control versus automation:** On-line analytical processing in data warehouses is a user-controlled process. The selection of dimensions and the application of OLAP operations, such as drill-down, roll-up, slicing, and dicing, are primarily directed and controlled by the users. Although the control in most OLAP systems is quite user-friendly, users do require a good understanding of the role of each dimension. Furthermore, in order to find a satisfactory description of the data, users may need to specify a long sequence of OLAP operations. It is often desirable to have a more automated process that helps users determine which dimensions (or attributes) should be included in the analysis, and the degree to which the given data set should be generalized in order to produce an interesting summarization of the data.

This section presents an alternative method for concept description, called *attribute-oriented induction*, which works for complex types of data and relies on a data-driven generalization process.

#### 4.3.1 Attribute-Oriented Induction for Data Characterization

The attribute-oriented induction (AOI) approach to concept description was first proposed in 1989, a few years prior to the introduction of the data cube approach. The data cube approach is essentially based on *materialized views* of the data, which typically have been precomputed in a data warehouse. In general, it performs off-line aggregation before an OLAP or data mining query is submitted for processing. On the other hand, the attribute-oriented induction approach is basically a *query-oriented*, generalization-based, on-line data analysis technique. Note that there is no inherent barrier distinguishing the two approaches based on on-line aggregation versus off-line precomputation. Some aggregations in the data cube can be computed on-line, while off-line precomputation of multidimensional space can speed up attribute-oriented induction as well.

The general idea of attribute-oriented induction is to first collect the task-relevant data using a database query and then perform generalization based on the examination of the number of distinct values of each attribute in the relevant set of data. The generalization is performed by either *attribute removal* or *attribute generalization*. Aggregation is performed by merging identical generalized tuples and accumulating their respective counts. This reduces the size of the generalized data set. The resulting generalized relation can be mapped into different forms for presentation to the user, such as charts or rules.

The following examples illustrate the process of attribute-oriented induction. We first discuss its use for characterization. The method is extended for the mining of class comparisons in Section 4.3.4.

**Example 4.20 A data mining query for characterization.** Suppose that a user would like to describe the general characteristics of graduate students in the *Big-University* database, given the attributes *name*, *gender*, *major*, *birth\_place*, *birth\_date*, *residence*, *phone#* (telephone number), and *gpa* (grade point average). A data mining query for this characterization can be expressed in the data mining query language, DMQL, as follows:

```
use Big.University.DB
mine characteristics as "Science.Students"
in relevance to name, gender, major, birth_place, birth_date, residence, phone#, gpa
from student
where status in "graduate"
```

We will see how this example of a typical data mining query can apply attribute-oriented induction for mining characteristic descriptions.

First, **data focusing** should be performed *prior* to attribute-oriented induction. This step corresponds to the specification of the task-relevant data (i.e., data for analysis). The data are collected based on the information provided in the data mining query. Since a data mining query is usually relevant to only a portion of the database, selecting the relevant set of data not only makes mining more efficient, but also derives more meaningful results than mining the entire database.

Specifying the set of relevant attributes (i.e., attributes for mining, as indicated in DMQL with the **in relevance** to clause) may be difficult for the user. A user may select only a few attributes that she feels may be important, while missing others that could also play a role in the description. For example, suppose that the dimension *birth\_place* is defined by the attributes *city*, *province\_or\_state*, and *country*. Of these attributes, let's say that the user has only thought to specify *city*. In order to allow generalization on the *birth\_place* dimension, the other attributes defining this dimension should also be included. In other words, having the system automatically include *province\_or\_state* and *country* as relevant attributes allows *city* to be generalized to these higher conceptual levels during the induction process.

At the other extreme, suppose that the user may have introduced too many attributes by specifying all of the possible attributes with the clause "**in relevance to \***". In this case, all of the attributes in the relation specified by the **from** clause would be included in the analysis. Many of these attributes are unlikely to contribute to an interesting description. A correlation-based (Section 2.4.1) or entropy-based (Section 2.6.1) analysis method can be used to perform attribute *relevance analysis* and filter out statistically irrelevant or weakly relevant attributes from the descriptive mining process. Other approaches, such as attribute subset selection, are also described in Chapter 2.

"What does the 'where status in "graduate"' clause mean?" This **where** clause implies that a concept hierarchy exists for the attribute *status*. Such a concept hierarchy organizes primitive-level data values for *status*, such as "M.Sc.", "M.A.", "M.B.A.", "Ph.D.", "B.Sc.", "B.A.", into higher conceptual levels, such as "graduate" and "undergraduate". This use of concept hierarchies does not appear in traditional relational query languages, yet is likely to become a common feature in data mining query languages.

The data mining query presented above is transformed into the following relational query for the collection of the task-relevant set of data.

```
use Big.University.DB
select name, gender, major, birth_place, birth_date, residence, phone#, gpa
from student
where status in { "M.Sc.", "M.A.", "M.B.A.", "Ph.D." }
```

The transformed query is executed against the relational database, *Big-University-DB*, and returns the data shown in Table 4.12. This table is called the (task-relevant) **initial working relation**. It is the data on which

induction will be performed. Note that each tuple is, in fact, a conjunction of attribute-value pairs. Hence, we can think of a tuple within a relation as a rule of conjuncts, and of induction on the relation as the generalization of these rules. ■

Table 4.12: Initial working relation: a collection of task-relevant data.

<i>name</i>	<i>gender</i>	<i>major</i>	<i>birth_place</i>	<i>birth_date</i>	<i>residence</i>	<i>phone#</i>	<i>gpa</i>
Jim Woodman	M	CS	Vancouver, BC, Canada	8-12-76	3511 Main St., Richmond	687-4598	3.67
Scott Lachance	M	CS	Montreal, Que, Canada	28-7-75	345 1st Ave., Richmond	253-9106	3.70
Laura Lee	F	physics	Seattle, WA, USA	25-8-70	125 Austin Ave., Burnaby	420-5232	3.83
...	...	...	...	...	...	...	...

“Now that the data are ready for attribute-oriented induction, how is attribute-oriented induction performed?” The essential operation of attribute-oriented induction is *data generalization*, which can be performed in either of two ways on the initial working relation: *attribute removal* and *attribute generalization*.

**Attribute removal** is based on the following rule: *If there is a large set of distinct values for an attribute of the initial working relation, but either (1) there is no generalization operator on the attribute (e.g., there is no concept hierarchy defined for the attribute), or (2) its higher-level concepts are expressed in terms of other attributes, then the attribute should be removed from the working relation.*

Let’s examine the reasoning behind this rule. An attribute-value pair represents a conjunct in a generalized tuple, or rule. The removal of a conjunct eliminates a constraint and thus generalizes the rule. If, as in case 1, there is a large set of distinct values for an attribute but there is no generalization operator for it, the attribute should be removed because it cannot be generalized, and preserving it would imply keeping a large number of disjuncts which contradicts the goal of generating concise rules. On the other hand, consider case 2, where the higher-level concepts of the attribute are expressed in terms of other attributes. For example, suppose that the attribute in question is *street*, whose higher-level concepts are represented by the attributes  $\langle \textit{city}, \textit{province\_or\_state}, \textit{country} \rangle$ . The removal of *street* is equivalent to the application of a generalization operator. This rule corresponds to the generalization rule known as *dropping conditions* in the machine learning literature on *learning from examples*.

**Attribute generalization** is based on the following rule: *If there is a large set of distinct values for an attribute in the initial working relation, and there exists a set of generalization operators on the attribute, then a generalization operator should be selected and applied to the attribute.* This rule is based on the following reasoning. Use of a generalization operator to generalize an attribute value within a tuple, or rule, in the working relation will make the rule cover more of the original data tuples, thus generalizing the concept it represents. This corresponds to the generalization rule known as *climbing generalization trees* in *learning from examples*, or *concept tree ascension*.

Both rules, *attribute removal* and *attribute generalization*, claim that if there is a *large* set of distinct values for an attribute, further generalization should be applied. This raises the question: how large is “a large set of distinct values for an attribute” considered to be?

Depending on the attributes or application involved, a user may prefer some attributes to remain at a rather low abstraction level while others are generalized to higher levels. The control of how high an attribute should be generalized is typically quite subjective. The control of this process is called **attribute generalization control**. If the attribute is generalized “too high,” it may lead to overgeneralization, and the resulting rules may not be very informative. On the other hand, if the attribute is not generalized to a “sufficiently high level,” then undergeneralization may result, where the rules obtained may not be informative either. Thus, a balance should be attained in attribute-oriented generalization.

There are many possible ways to control a generalization process. We will describe two common approaches and then illustrate how they work with an example.

The first technique, called **attribute generalization threshold control**, either sets one generalization threshold for all of the attributes, or sets one threshold for each attribute. If the number of distinct values in an attribute is greater than the attribute threshold, further attribute removal or attribute generalization should be performed.

Data mining systems typically have a default attribute threshold value generally ranging from 2 to 8) and should allow experts and users to modify the threshold values as well. If a user feels that the generalization reaches too high a level for a particular attribute, the threshold can be increased. This corresponds to drilling down along the attribute. Also, to further generalize a relation, the user can reduce the threshold of a particular attribute, which corresponds to rolling up along the attribute.

The second technique, called **generalized relation threshold control**, sets a threshold for the generalized relation. If the number of (distinct) tuples in the generalized relation is greater than the threshold, further generalization should be performed. Otherwise, no further generalization should be performed. Such a threshold may also be preset in the data mining system (usually within a range of 10 to 30), or set by an expert or user, and should be adjustable. For example, if a user feels that the generalized relation is too small, she can increase the threshold, which implies drilling down. Otherwise, to further generalize a relation, she can reduce the threshold, which implies rolling up.

These two techniques can be applied in sequence: first apply the attribute threshold control technique to generalize each attribute, and then apply relation threshold control to further reduce the size of the generalized relation. No matter which generalization control technique is applied, the user should be allowed to adjust the generalization thresholds in order to obtain interesting concept descriptions.

In many database-oriented induction processes, users are interested in obtaining quantitative or statistical information about the data at different levels of abstraction. Thus, it is important to accumulate count and other aggregate values in the induction process. Conceptually, this is performed as follows. The aggregate function, **count**, is associated with each database tuple. Its value for each tuple in the initial working relation is initialized to 1. Through attribute removal and attribute generalization, tuples within the initial working relation may be generalized, resulting in groups of *identical tuples*. In this case, all of the identical tuples forming a group should be merged into one tuple. The count of this new, generalized tuple is set to the total number of tuples from the initial working relation that are represented by (i.e., were merged into) the new generalized tuple. For example, suppose that by attribute-oriented induction, 52 data tuples from the initial working relation are all generalized to the same tuple,  $T$ . That is, the generalization of these 52 tuples resulted in 52 identical instances of tuple  $T$ . These 52 identical tuples are merged to form one instance of  $T$ , whose count is set to 52. Other popular aggregate functions that could also be associated with each tuple include **sum** and **avg**. For a given generalized tuple, **sum** contains the sum of the values of a given numeric attribute for the initial working relation tuples making up the generalized tuple. Suppose that tuple  $T$  contained  $\text{sum}(\text{units\_sold})$  as an aggregate function. The sum value for tuple  $T$  would then be set to the total number of units sold for each of the 52 tuples. The aggregate **avg** (average) is computed according to the formula,  $\text{avg} = \text{sum}/\text{count}$ .

**Example 4.21 Attribute-oriented induction.** Here we show how attribute-oriented induction is performed on the initial working relation of Table 4.12. For each attribute of the relation, the generalization proceeds as follows:

1. *name*: Since there are a large number of distinct values for *name* and there is no generalization operation defined on it, this attribute is removed.
2. *gender*: Since there are only two distinct values for *gender*, this attribute is retained and no generalization is performed on it.
3. *major*: Suppose that a concept hierarchy has been defined that allows the attribute *major* to be generalized to the values  $\{\text{arts\&science}, \text{engineering}, \text{business}\}$ . Suppose also that the attribute generalization threshold is set to 5, and that there are over 20 distinct values for *major* in the initial working relation. By attribute generalization and attribute generalization control, *major* is therefore generalized by climbing the given concept hierarchy.
4. *birth\_place*: This attribute has a large number of distinct values; therefore, we would like to generalize it. Suppose that a concept hierarchy exists for *birth\_place*, defined as “*city* < *province\_or\_state* < *country*”. If the number of distinct values for *country* in the initial working relation is greater than the attribute generalization threshold, then *birth\_place* should be removed, since even though a generalization operator exists for it, the



generalization threshold would not be satisfied. If instead, the number of distinct values for *country* is less than the attribute generalization threshold, then *birth\_place* should be generalized to *birth\_country*.

5. *birth\_date*: Suppose that a hierarchy exists that can generalize *birth\_date* to *age*, and *age* to *age\_range*, and that the number of age ranges (or intervals) is small with respect to the attribute generalization threshold. Generalization of *birth\_date* should therefore take place.
6. *residence*: Suppose that *residence* is defined by the attributes *number*, *street*, *residence\_city*, *residence\_province\_or\_state*, and *residence\_country*. The number of distinct values for *number* and *street* will likely be very high, since these concepts are quite low level. The attributes *number* and *street* should therefore be removed, so that *residence* is then generalized to *residence\_city*, which contains fewer distinct values.
7. *phone#*: As with the attribute *name* above, this attribute contains too many distinct values and should therefore be removed in generalization.
8. *gpa*: Suppose that a concept hierarchy exists for *gpa* that groups values for grade point average into numerical intervals like {3.75–4.0, 3.5–3.75, ...}, which in turn are grouped into descriptive values, such as {*excellent*, *very good*, ...}. The attribute can therefore be generalized.

The generalization process will result in groups of identical tuples. For example, the first two tuples of Table 4.12 both generalize to the same identical tuple (namely, the first tuple shown in Table 4.13). Such identical tuples are then merged into one, with their **counts** accumulated. This process leads to the generalized relation shown in Table 4.13.

Table 4.13: A generalized relation obtained by attribute-oriented induction on the data of Table 4.12.

<i>gender</i>	<i>major</i>	<i>birth_country</i>	<i>age_range</i>	<i>residence_city</i>	<i>gpa</i>	<b>count</b>
M	Science	Canada	20-25	Richmond	very_good	16
F	Science	Foreign	25-30	Burnaby	excellent	22
...	...	...	...	...	...	...

Based on the vocabulary used in OLAP, we may view **count** as a *measure*, and the remaining attributes as *dimensions*. Note that aggregate functions, such as **sum**, may be applied to numerical attributes, like *salary* and *sales*. These attributes are referred to as *measure attributes*. ■

Implementation techniques and methods of presenting the derived generalization are discussed in the following subsections.

### 4.3.2 Efficient Implementation of Attribute-Oriented Induction

“How is attribute-oriented induction actually implemented?” The previous subsection provided an introduction to attribute-oriented induction. The general procedure is summarized in Figure 4.19. The efficiency of this algorithm is analyzed as follows:

- Step 1 of the algorithm is essentially a relational query to collect the task-relevant data into the **working relation**, *W*. Its processing efficiency depends on the query processing methods used. Given the successful implementation and commercialization of database systems, this step is expected to have good performance.
- Step 2 collects statistics on the working relation. This requires scanning the relation at most once. The cost for computing the minimum desired level and determining the mapping pairs,  $(v, v')$ , for each attribute is dependent on the number of distinct values for each attribute and is smaller than *N*, the number of tuples in the initial relation.

**Algorithm: Attribute oriented induction.** Mining generalized characteristics in a relational database given a user's data mining request.

**Input:**

- $DB$ , a relational database;
- $DMQuery$ , a data mining query;
- $a\_list$ , a list of attributes (containing attributes,  $a_i$ );
- $Gen(a_i)$ , a set of concept hierarchies or generalization operators on attributes,  $a_i$ ;
- $a\_gen\_thresh(a_i)$ , attribute generalization thresholds for each  $a_i$ .

**Output:**  $P$ , a *Prime\_generalized\_relation*.

**Method:**

1.  $W \leftarrow \text{get\_task\_relevant\_data}(DMQuery, DB)$ ; // Let  $W$ , the working relation, hold the task-relevant data.
2.  $\text{prepare\_for\_generalization}(W)$ ; // This is implemented as follows.
  - (a) Scan  $W$  and collect the distinct values for each attribute,  $a_i$ . (Note: If  $W$  is very large, this may be done by examining a sample of  $W$ .)
  - (b) For each attribute  $a_i$ , determine whether  $a_i$  should be removed, and if not, compute its minimum desired level  $L_i$  based on its given or default attribute threshold, and determine the mapping-pairs  $(v, v')$ , where  $v$  is a distinct value of  $a_i$  in  $W$ , and  $v'$  is its corresponding generalized value at level  $L_i$ .
3.  $P \leftarrow \text{generalization}(W)$ ;  
 The *Prime\_generalized\_relation*,  $P$ , is derived by replacing each value  $v$  in  $W$  by its corresponding  $v'$  in the mapping while accumulating count and computing any other aggregate values.  
 This step can be implemented efficiently using either of the two following variations:
  - (a) For each generalized tuple, insert the tuple into a sorted prime relation  $P$  by a binary search: if the tuple is already in  $P$ , simply increase its count and other aggregate values accordingly; otherwise, insert it into  $P$ .
  - (b) Since in most cases the number of distinct values at the prime relation level is small, the prime relation can be coded as an  $m$ -dimensional array where  $m$  is the number of attributes in  $P$ , and each dimension contains the corresponding generalized attribute values. Each array element holds the corresponding count and other aggregation values, if any. The insertion of a generalized tuple is performed by measure aggregation in the corresponding array element.

Figure 4.19: Basic algorithm for attribute-oriented induction.

- Step 3 derives the **prime relation**,  $P$ . This is performed by inserting generalized tuples into  $P$ . There are a total of  $N$  tuples in  $W$  and  $p$  tuples in  $P$ . For each tuple,  $t$ , in  $W$ , we substitute its attribute values based on the derived mapping-pairs. This results in a generalized tuple,  $t'$ . If variation (a) is adopted, each  $t'$  takes  $O(\log p)$  to find the location for count increment or tuple insertion. Thus the total time complexity is  $O(N \times \log p)$  for all of the generalized tuples. If variation (b) is adopted, each  $t'$  takes  $O(1)$  to find the tuple for count increment. Thus the overall time complexity is  $O(N)$  for all of the generalized tuples.

Many data analysis tasks need to examine a good number of dimensions or attributes. This may involve *dynamically* introducing and testing additional attributes rather than just those specified in the mining query. Moreover, a user with little knowledge of the *truly* relevant set of data may simply specify “in relevance to  $*$ ” in the mining query, which includes all the attributes into the analysis. Therefore, an advanced concept description mining process needs to perform attribute relevance analysis on large sets of attributes to select the most relevant ones. Such analysis may employ correlation or entropy measures, as described in Chapter 2 on data preprocessing.

### 4.3.3 Presentation of the Derived Generalization

“Attribute-oriented induction generates one or a set of generalized descriptions. How can these descriptions be visualized?” The descriptions can be presented to the user in a number of different ways. Generalized descriptions resulting from attribute-oriented induction are most commonly displayed in the form of a **generalized relation** (or **table**).

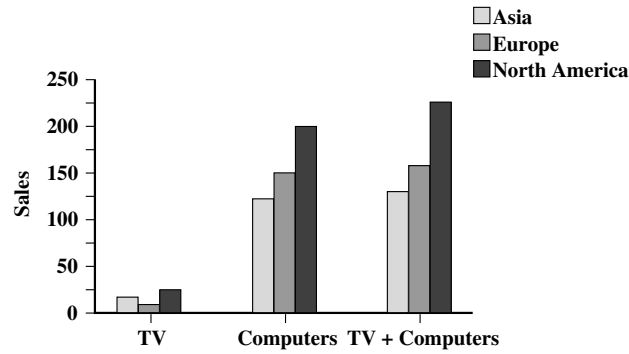


Figure 4.20: Bar chart representation of the sales in 2004.

**Example 4.22 Generalized relation (table).** Suppose that attribute-oriented induction was performed on a *sales* relation of the *AllElectronics* database, resulting in the generalized description of Table 4.14 for sales in 2004. The description is shown in the form of a generalized relation. Table 4.13 of Example 4.21 is another example of a generalized relation. ■

Table 4.14: A generalized relation for the sales in 2004.

<i>location</i>	<i>item</i>	<i>sales</i> (in million dollars)	<i>count</i> (in thousands)
Asia	TV	15	300
Europe	TV	12	250
North_America	TV	28	450
Asia	computer	120	1000
Europe	computer	150	1200
North_America	computer	200	1800

Descriptions can also be visualized in the form of **cross-tabulations**, or **crosstabs**. In a two-dimensional crosstab, each row represents a value from an attribute, and each column represents a value from another attribute. In an  $n$ -dimensional crosstab (for  $n > 2$ ), the columns may represent the values of more than one attribute, with subtotals shown for attribute-value groupings. This representation is similar to *spreadsheets*. It is easy to map directly from a data cube structure to a crosstab.

Table 4.15: A crosstab for the sales in 2004.

<i>location \ item</i>	TV		computer		<i>both_items</i>	
	<i>sales</i>	<i>count</i>	<i>sales</i>	<i>count</i>	<i>sales</i>	<i>count</i>
Asia	15	300	120	1000	135	1300
Europe	12	250	150	1200	162	1450
North_America	28	450	200	1800	228	2250
<i>all_regions</i>	45	1000	470	4000	525	5000

**Example 4.23 Cross-tabulation.** The generalized relation shown in Table 4.14 can be transformed into the 3-D cross-tabulation shown in Table 4.15. ■

Generalized data can be presented graphically, using bar charts, pie charts, and curves. Visualization with graphs is popular in data analysis. Such graphs and curves can represent 2-D or 3-D data.

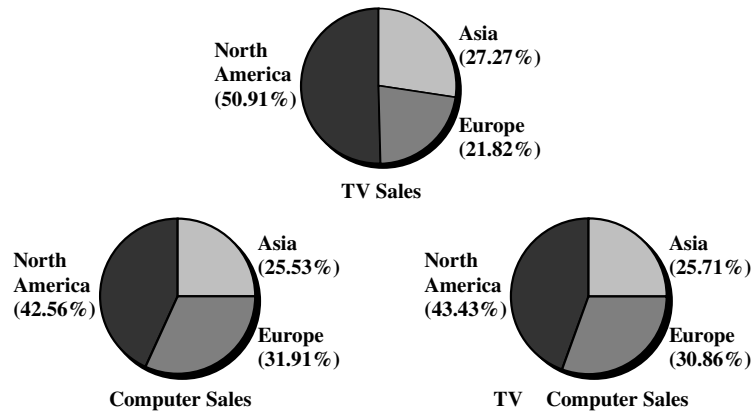


Figure 4.21: Pie chart representation of the sales in 2004.

**Example 4.24 Bar chart and pie chart.** The sales data of the crosstab shown in Table 4.15 can be transformed into the bar chart representation of Figure 4.20 and the pie chart representation of Figure 4.21. ■

Finally, a 3-D generalized relation or crosstab can be represented by a 3-D data cube, which is useful for browsing the data at different levels of generalization.

**Example 4.25 Cube view.** Consider the data cube shown in Figure 4.22 for the dimensions *item*, *location*, and *cost*. This is the same kind of data cube that we have seen so far, although it is presented in a slightly different way. Here, the *size* of a cell (displayed as a tiny cube) represents the *count* of the corresponding cell, while the *brightness* of the cell can be used to represent another measure of the cell, such as  $\text{sum}(\text{sales})$ . Pivoting, drilling, and slicing-and-dicing operations can be performed on the data cube browser by mouse clicking. ■

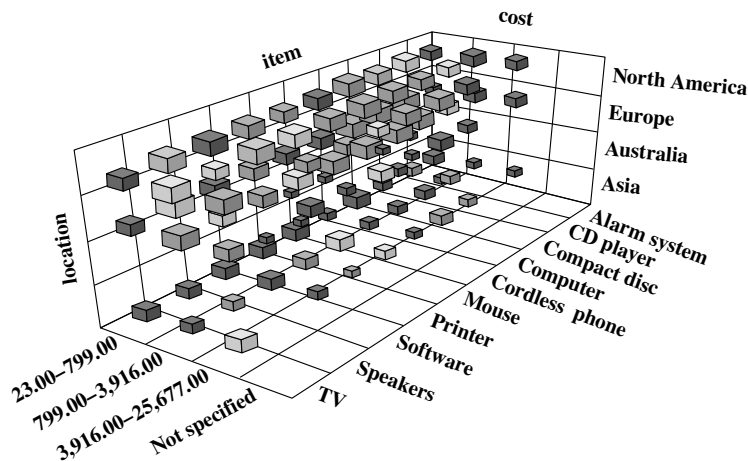


Figure 4.22: A 3-D cube view representation of the sales in 2004.

A generalized relation may also be represented in the form of logic rules. Typically, each generalized tuple represents a rule disjunct. Since data in a large database usually span a diverse range of distributions, a single generalized tuple is unlikely to *cover*, or represent, 100% of the initial working relation tuples, or *cases*. Thus quantitative information, such as the percentage of data tuples that satisfy the left- and right-hand side of the

rule, should be associated with each rule. A logic rule that is associated with quantitative information is called a **quantitative rule**.

To define a quantitative characteristic rule, we introduce the **t-weight** as an interestingness measure that describes the *typicality* of each *disjunct* in the rule, or of each *tuple* in the corresponding generalized relation. The measure is defined as follows. Let the class of objects that is to be characterized (or described by the rule) be called the *target class*. Let  $\mathbf{q}_a$  be a generalized tuple describing the target class. The **t-weight** for  $\mathbf{q}_a$  is the percentage of tuples of the target class from the initial working relation that are covered by  $\mathbf{q}_a$ . Formally, we have

$$t\_weight = \text{count}(\mathbf{q}_a) / \sum_{i=1}^n \text{count}(\mathbf{q}_i), \quad (4.1)$$

where  $n$  is the number of tuples for the target class in the generalized relation;  $\mathbf{q}_1, \dots, \mathbf{q}_n$  are tuples for the target class in the generalized relation; and  $\mathbf{q}_a$  is in  $\mathbf{q}_1, \dots, \mathbf{q}_n$ . Obviously, the range for the t-weight is  $[0.0, 1.0]$  or  $[0\%, 100\%]$ .

A **quantitative characteristic rule** can then be represented either (1) in logic form by associating the corresponding t-weight value with each disjunct covering the target class, or (2) in the relational table or crosstab form by changing the count values in these tables for tuples of the target class to the corresponding t-weight values.

Each disjunct of a quantitative characteristic rule represents a condition. In general, the disjunction of these conditions forms a *necessary* condition of the target class, since the condition is derived based on all of the cases of the target class; that is, all tuples of the target class must satisfy this condition. However, the rule may not be a *sufficient* condition of the target class, since a tuple satisfying the same condition could belong to another class. Therefore, the rule should be expressed in the form

$$\forall \mathbf{X}, \text{target\_class}(\mathbf{X}) \Rightarrow \text{condition}_1(\mathbf{X})[t : w_1] \vee \dots \vee \text{condition}_m(\mathbf{X})[t : w_m]. \quad (4.2)$$

The rule indicates that if  $\mathbf{X}$  is in the *target class*, there is a probability of  $w_i$  that  $\mathbf{X}$  satisfies *condition<sub>i</sub>*, where  $w_i$  is the t-weight value for condition or disjunct  $i$ , and  $i$  is in  $\{1, \dots, m\}$ .

**Example 4.26 Quantitative characteristic rule.** The crosstab shown in Table 4.15 can be transformed into logic rule form. Let the target class be the set of computer items. The corresponding characteristic rule, in logic form, is

$$\begin{aligned} \forall \mathbf{X}, \text{item}(\mathbf{X}) = \text{"computer"} \Rightarrow \\ (\text{location}(\mathbf{X}) = \text{"Asia"}) [t : 25.00\%] \vee (\text{location}(\mathbf{X}) = \text{"Europe"}) [t : 30.00\%] \vee \\ (\text{location}(\mathbf{X}) = \text{"North\_America"}) [t : 45.00\%] \end{aligned}$$

Notice that the first t-weight value of 25.00% is obtained by 1000, the value corresponding to the count slot for “(Asia, computer)”, divided by 4000, the value corresponding to the count slot for “(all\_regions, computer)”. (That is, 4000 represents the total number of computer items sold.) The t-weights of the other two disjuncts were similarly derived. Quantitative characteristic rules for other target classes can be computed in a similar fashion. ■

“How can the t-weight and interestingness measures in general be used by the data mining system to display only the concept descriptions that it objectively evaluates as interesting?” A threshold can be set for this purpose. For example, if the t-weight of a generalized tuple is lower than the threshold, then the tuple is considered to represent only a negligible portion of the database and can therefore be ignored as uninteresting. Ignoring such negligible tuples does not mean that they should be removed from the intermediate results (i.e., the prime generalized relation, or the data cube, depending on the implementation) since they may contribute to subsequent further exploration of the data by the user via interactive rolling up or drilling down of other dimensions and levels of abstraction. Such a threshold may be referred to as a **significance threshold** or **support threshold**, where the latter term is commonly used in association rule mining.

### 4.3.4 Mining Class Comparisons: Discriminating between Different Classes

In many applications, users may not be interested in having a single class (or concept) described or characterized, but rather would prefer to mine a description that compares or distinguishes one class (or concept) from other comparable classes (or concepts). Class discrimination or comparison (hereafter referred to as **class comparison**) mines descriptions that distinguish a target class from its contrasting classes. Notice that the target and contrasting classes must be *comparable* in the sense that they share similar dimensions and attributes. For example, the three classes, *person*, *address*, and *item*, are not comparable. However, the sales in the last three years are comparable classes, and so are computer science students versus physics students.

Our discussions on class characterization in the previous sections handle multilevel data summarization and characterization in a single class. The techniques developed can be extended to handle class comparison across several comparable classes. For example, the attribute generalization process described for class characterization can be modified so that the generalization is performed *synchronously* among all the classes compared. This allows the attributes in all of the classes to be generalized to the *same* levels of abstraction. Suppose, for instance, that we are given the *AlIElectronics* data for sales in 2003 and sales in 2004 and would like to compare these two classes. Consider the dimension *location* with abstractions at the *city*, *province\_or\_state*, and *country* levels. Each class of data should be generalized to the same *location* level. That is, they are synchronously all generalized to either the *city* level, or the *province\_or\_state* level, or the *country* level. Ideally, this is more useful than comparing, say, the sales in Vancouver in 2003 with the sales in the United States in 2004 (i.e., where each set of sales data is generalized to a different level). The users, however, should have the option to overwrite such an automated, synchronous comparison with their own choices, when preferred.

“How is class comparison performed?” In general, the procedure is as follows:

1. **Data collection:** The set of relevant data in the database is collected by query processing and is partitioned respectively into a *target class* and one or a set of *contrasting class(es)*.
2. **Dimension relevance analysis:** If there are many dimensions, then dimension relevance analysis should be performed on these classes to select only the highly relevant dimensions for further analysis. Correlation or entropy-based measures can be used for this step (Chapter 2).
3. **Synchronous generalization:** Generalization is performed on the target class to the level controlled by a user- or expert-specified dimension threshold, which results in a **prime target class relation**. The concepts in the contrasting class(es) are generalized to the same level as those in the prime target class relation, forming the **prime contrasting class(es) relation**.
4. **Presentation of the derived comparison:** The resulting class comparison description can be visualized in the form of tables, graphs, and rules. This presentation usually includes a “contrasting” measure such as **count%** (percentage count) that reflects the comparison between the target and contrasting classes. The user can adjust the comparison description by applying drill-down, roll-up, and other OLAP operations to the target and contrasting classes, as desired.

The above discussion outlines a general algorithm for mining comparisons in databases. In comparison with characterization, the above algorithm involves synchronous generalization of the target class with the contrasting classes so that classes are simultaneously compared at the same levels of abstraction.

The following example mines a class comparison describing the graduate students and the undergraduate students at *Big-University*.

**Example 4.27 Mining a class comparison.** Suppose that you would like to compare the general properties between the graduate students and the undergraduate students at *Big-University*, given the attributes *name*, *gender*, *major*, *birth\_place*, *birth\_date*, *residence*, *phone#*, and *gpa*.

This data mining task can be expressed in DMQL as follows:

```

use Big.University.DB
mine comparison as "grad.vs.undergrad_students"
in relevance to name, gender, major, birth_place, birth_date, residence, phone#, gpa
for "graduate_students"
where status in "graduate"
versus "undergraduate_students"
where status in "undergraduate"
analyze count%
from student

```

Let's see how this typical example of a data mining query for mining comparison descriptions can be processed.

First, the query is transformed into two relational queries that collect two sets of task-relevant data: one for the *initial target class working relation*, and the other for the *initial contrasting class working relation*, as shown in Tables 4.16 and 4.17. This can also be viewed as the construction of a data cube, where the status {*graduate*, *undergraduate*} serves as one dimension, and the other attributes form the remaining dimensions.

<i>name</i>	<i>gender</i>	<i>major</i>	<i>birth_place</i>	<i>birth_date</i>	<i>residence</i>	<i>phone#</i>	<i>gpa</i>
Jim Woodman	M	CS	Vancouver, BC, Canada	8-12-76	3511 Main St., Richmond	687-4598	3.67
Scott Lachance	M	CS	Montreal, Que, Canada	28-7-75	345 1st Ave., Vancouver	253-9106	3.70
Laura Lee	F	Physics	Seattle, WA, USA	25-8-70	125 Austin Ave., Burnaby	420-5232	3.83
...	...	...	...	...	...	...	...

Table 4.16: Initial working relations: the *target class* (graduate students)

<i>name</i>	<i>gender</i>	<i>major</i>	<i>birth_place</i>	<i>birth_date</i>	<i>residence</i>	<i>phone#</i>	<i>gpa</i>
Bob Schumann	M	Chemistry	Calgary, Alt, Canada	10-1-78	2642 Halifax St., Burnaby	294-4291	2.96
Amy Eau	F	Biology	Golden, BC, Canada	30-3-76	463 Sunset Cres., Vancouver	681-5417	3.52
...	...	...	...	...	...	...	...

Table 4.17: Initial working relations: the *contrasting class* (undergraduate students)

Second, dimension relevance analysis can be performed, when necessary, on the two classes of data. After this analysis, irrelevant or weakly relevant dimensions, such as *name*, *gender*, *birth\_place*, *residence*, and *phone#*, are removed from the resulting classes. Only the highly relevant attributes are included in the subsequent analysis.

Third, synchronous generalization is performed: Generalization is performed on the target class to the levels controlled by user- or expert-specified dimension thresholds, forming the *prime target class relation*. The contrasting class is generalized to the same levels as those in the prime target class relation, forming the *prime contrasting class(es) relation*, as presented in Tables 4.18 and 4.19. In comparison with undergraduate students, graduate students tend to be older and have a higher GPA, in general.

Finally, the resulting class comparison is presented in the form of tables, graphs, and/or rules. This visualization includes a contrasting measure (such as *count%*) that compares between the target class and the contrasting class. For example, 5.02% of the graduate students majoring in Science are between 26 and 30 years of age and have a "good" GPA, while only 2.32% of undergraduates have these same characteristics. Drilling and other OLAP operations may be performed on the target and contrasting classes as deemed necessary by the user in order to adjust the abstraction levels of the final description. ■

*"How can class comparison descriptions be presented?"* As with class characterizations, class comparisons can be presented to the user in various forms, including generalized relations, crosstabs, bar charts, pie charts, curves, cubes, and rules. With the exception of logic rules, these forms are used in the same way for characterization as for comparison. In this section, we discuss the visualization of class comparisons in the form of discriminant rules.

As is similar with characterization descriptions, the discriminative features of the target and contrasting classes

Table 4.18: Prime generalized relation for the *target class* (graduate students)

<i>major</i>	<i>age_range</i>	<i>gpa</i>	<i>count%</i>
Science	21...25	good	5.53%
Science	26...30	good	5.02%
Science	over_30	very_good	5.86%
...	...	...	...
Business	over_30	excellent	4.68%

Table 4.19: Prime generalized relation for the *contrasting class* (undergraduate students)

<i>major</i>	<i>age_range</i>	<i>gpa</i>	<i>count%</i>
Science	16...20	fair	5.53%
Science	16...20	good	4.53%
...	...	...	...
Science	26...30	good	2.32%
...	...	...	...
Business	over_30	excellent	0.68%

of a comparison description can be described quantitatively by a *quantitative discriminant rule*, which associates a statistical interestingness measure, *d-weight*, with each generalized tuple in the description.

Let  $\mathbf{q}_a$  be a generalized tuple, and  $C_j$  be the target class, where  $\mathbf{q}_a$  covers some tuples of the target class. Note that it is possible that  $\mathbf{q}_a$  also covers some tuples of the contrasting classes, particularly since we are dealing with a comparison description. The **d-weight** for  $\mathbf{q}_a$  is the ratio of the number of tuples from the initial target class working relation that are covered by  $\mathbf{q}_a$  to the total number of tuples in both the initial target class and contrasting class working relations that are covered by  $\mathbf{q}_a$ . Formally, the d-weight of  $\mathbf{q}_a$  for the class  $C_j$  is defined as

$$d\_weight = \text{count}(\mathbf{q}_a \in C_j) / \sum_{i=1}^m \text{count}(\mathbf{q}_a \in C_i), \quad (4.3)$$

where  $m$  is the total number of the target and contrasting classes,  $C_j$  is in  $\{C_1, \dots, C_m\}$ , and  $\text{count}(\mathbf{q}_a \in C_i)$  is the number of tuples of class  $C_i$  that are covered by  $\mathbf{q}_a$ . The range for the d-weight is  $[0.0, 1.0]$  (or  $[0\%, 100\%]$ ).

A high d-weight in the target class indicates that the concept represented by the generalized tuple is primarily derived from the target class, whereas a low d-weight implies that the concept is primarily derived from the contrasting classes. A threshold can be set to control the display of interesting tuples based on the d-weight or other measures used, as described in Section 4.3.3.

**Example 4.28 Computing the d-weight measure.** In Example 4.27, suppose that the count distribution for the generalized tuple, *major* = “Science” AND *age\_range* = “21...25” AND *gpa* = “good”, from Tables 4.18 and 4.19 is as shown in Table 4.20.

Table 4.20: Count distribution between graduate and undergraduate students for a generalized tuple.

<i>status</i>	<i>major</i>	<i>age_range</i>	<i>gpa</i>	<i>count</i>
graduate	Science	21...25	good	90
undergraduate	Science	21...25	good	210

The d-weight for the given generalized tuple is  $90/(90 + 210) = 30\%$  with respect to the target class, and  $210/(90 + 210) = 70\%$  with respect to the contrasting class. That is, *if a student majoring in Science is 21 to 25*



years old and has a “good” gpa, then based on the data, there is a 30% probability that she is a graduate student, versus a 70% probability that she is an undergraduate student. Similarly, the d-weights for the other generalized tuples in Tables 4.18 and 4.19 can be derived. ■

A **quantitative discriminant rule** for the target class of a given comparison description is written in the form

$$\forall \mathbf{X}, \text{target\_class}(\mathbf{X}) \Leftarrow \text{condition}(\mathbf{X}) \quad [d : d\_weight], \quad (4.4)$$

where the condition is formed by a generalized tuple of the description. This is different from rules obtained in class characterization where the arrow of implication is from left to right.

**Example 4.29 Quantitative discriminant rule.** Based on the generalized tuple and count distribution in Example 4.28, a quantitative discriminant rule for the target class *graduate\_student* can be written as follows:

$$\begin{aligned} \forall \mathbf{X}, \text{status}(\mathbf{X}) = \text{“graduate\_student”} \Leftarrow \\ \text{major}(\mathbf{X}) = \text{“Science”} \wedge \text{age\_range}(\mathbf{X}) = \text{“21...25”} \wedge \text{gpa}(\mathbf{X}) = \text{“good”} [d : 30\%]. \end{aligned} \quad (4.5)$$

Notice that a discriminant rule provides a *sufficient* condition, but not a *necessary* one, for an object (or tuple) to be in the target class. For example, Rule (4.5) implies that if  $\mathbf{X}$  satisfies the condition, then the probability that  $\mathbf{X}$  is a graduate student is 30%. However, it does not imply the probability that  $\mathbf{X}$  meets the condition, given that  $\mathbf{X}$  is a graduate student. This is because although the tuples that meet the condition are in the target class, other tuples that do not necessarily satisfy this condition may also be in the target class, since the rule may not cover *all* of the examples of the target class in the database. Therefore, the condition is sufficient, but not necessary.

### 4.3.5 Class Description: Presentation of Both Characterization and Comparison

“Since class characterization and class comparison are two aspects forming a class description, can we present both in the same table or in the same rule?” Actually, as long as we have a clear understanding of the meaning of the t-weight and d-weight measures and can interpret them correctly, there is no additional difficulty in presenting both aspects in the same table. Let’s examine an example of expressing both class characterization and class comparison in the same crosstab.

**Example 4.30 Crosstab for class characterization and class comparison.** Let Table 4.21 be a crosstab showing the total number (in thousands) of TVs and computers sold at *AllElectronics* in 2004.

Table 4.21: A crosstab for the total number (count) of TVs and computers sold in thousands in 2004.

<i>location \ item</i>	TV	computer	<i>both_items</i>
Europe	80	240	320
North_America	120	560	680
<i>both_regions</i>	200	800	1000

Let *Europe* be the target class and *North\_America* be the contrasting class. The t-weights and d-weights of the sales distribution between the two classes are presented in Table 4.22. According to the table, the t-weight of a generalized tuple or object (e.g., *item* = “TV”) for a given class (e.g., the target class *Europe*) shows how typical the tuple is of the given class (e.g., what proportion of these sales in Europe are for TVs?). The d-weight of a

Table 4.22: The same crosstab as in Table 4.21, but here the t-weight and d-weight values associated with each class are shown.

<i>location \ item</i>	TV			computer			<i>both_items</i>		
	count	t-weight	d-weight	count	t-weight	d-weight	count	t-weight	d-weight
Europe	80	25%	40%	240	75%	30%	320	100%	32%
North_America	120	17.65%	60%	560	82.35%	70%	680	100%	68%
<i>both_regions</i>	200	20%	100%	800	80%	100%	1000	100%	100%

tuple shows how distinctive the tuple is in the given (target or contrasting) class in comparison with its rival class (e.g., how do the TV sales in Europe compare with those in North America?).

For example, the t-weight for “(*Europe*, *TV*)” is 25% because the number of TVs sold in Europe (80,000) represents only 25% of the European sales for both items (320,000). The d-weight for “(*Europe*, *TV*)” is 40% because the number of TVs sold in Europe (80,000) represents 40% of the number of TVs sold in both the target and the contrasting classes of Europe and North America, respectively (which is 200,000). ■

Notice that the **count** measure in the crosstab of Table 4.22 obeys the general property of a crosstab (i.e., the **count** values per row and per column, when totaled, match the corresponding totals in the *both\_items* and *both\_regions* slots, respectively). However, this property is not observed by the t-weight and d-weight measures. This is because the semantic meaning of each of these measures is different from that of **count**, as we explained in Example 4.30.

“Can a quantitative characteristic rule and a quantitative discriminant rule be expressed together in the form of one rule?” The answer is yes—a quantitative characteristic rule and a quantitative discriminant rule for the same class can be combined to form a *quantitative description rule* for the class, which displays the t-weights and d-weights associated with the corresponding characteristic and discriminant rules. To see how this is done, let’s quickly review how quantitative characteristic and discriminant rules are expressed.

As discussed in Section 4.3.3, a quantitative characteristic rule provides a necessary condition for the given target class since it presents a probability measurement for each property that can occur in the target class. Such a rule is of the form

$$\forall \mathbf{X}, \text{target\_class}(\mathbf{X}) \Rightarrow \text{condition}_1(\mathbf{X})[t : w_1] \vee \cdots \vee \text{condition}_m(\mathbf{X})[t : w_m], \quad (4.6)$$

where each condition represents a property of the target class. The rule indicates that if  $\mathbf{X}$  is in the *target\_class*, the probability that  $\mathbf{X}$  satisfies *condition<sub>i</sub>* is the value of the t-weight,  $w_i$ , where  $i$  is in  $\{1, \dots, m\}$ .

As previously discussed in Section 4.3.4, a quantitative discriminant rule provides a sufficient condition for the target class since it presents a quantitative measurement of the properties that occur in the target class versus those that occur in the contrasting classes. Such a rule is of the form

$$\forall \mathbf{X}, \text{target\_class}(\mathbf{X}) \Leftarrow \text{condition}_1(\mathbf{X})[d : w_1] \vee \cdots \vee \text{condition}_m(\mathbf{X})[d : w_m].$$

The rule indicates that if  $\mathbf{X}$  satisfies *condition<sub>i</sub>*, there is a probability of  $w_i$  (the d-weight value) that  $\mathbf{X}$  is in the *target\_class*, where  $i$  is in  $\{1, \dots, m\}$ .

A quantitative characteristic rule and a quantitative discriminant rule for a given class can be combined as follows to form a **quantitative description rule**: (1) For each condition, show both the associated t-weight and d-weight, and (2) a bidirectional arrow should be used between the given class and the conditions. That is, a quantitative description rule is of the form

$$\forall \mathbf{X}, \text{target\_class}(\mathbf{X}) \Leftrightarrow \text{condition}_1(\mathbf{X})[t : w_1, d : w'_1] \vee \cdots \vee \text{condition}_m(\mathbf{X})[t : w_m, d : w'_m]. \quad (4.7)$$

This form indicates that for  $i$  from 1 to  $m$ , if  $\mathbf{X}$  is in the *target\_class*, there is a probability of  $w_i$  that  $\mathbf{X}$  satisfies *condition<sub>i</sub>*; and if  $\mathbf{X}$  satisfies *condition<sub>i</sub>*, there is a probability of  $w'_i$  that  $\mathbf{X}$  is in the *target\_class*.

**Example 4.31 Quantitative description rule.** It is straightforward to transform the crosstab of Table 4.22 in Example 4.30 into a class description in the form of quantitative description rules. For example, the quantitative description rule for the target class, *Europe*, is

$$\forall X, \text{location}(X) = \text{"Europe"} \Leftrightarrow (\text{item}(X) = \text{"TV"}) [t : 25\%, d : 40\%] \vee (\text{item}(X) = \text{"computer"}) [t : 75\%, d : 30\%] \quad (4.8)$$

The rule states that for the sales of TVs and computers at *AllElectronics* in 2004, if the sale of one of these items occurred in Europe, then the probability of the item being a TV is 25%, while that of being a computer is 75%. On the other hand, if we compare the sales of these items in Europe and North America, then 40% of the TVs were sold in Europe (and therefore we can deduce that 60% of the TVs were sold in North America). Furthermore, regarding computer sales, 30% of these sales took place in Europe. ■

## 4.4 Summary

- **Data generalization** is a process that abstracts a large set of task-relevant data in a database from a relatively low conceptual level to higher conceptual levels. Data generalization approaches include data cube-based data aggregation and attribute-oriented induction.
- From a data analysis point of view, data generalization is a form of *descriptive data mining*. **Descriptive data mining** describes data in a concise and summarative manner and presents interesting general properties of the data. This is different from **predictive data mining**, which analyzes data in order to construct one or a set of models, and attempts to predict the behavior of new data sets. This chapter focussed on methods for descriptive data mining.
- A data cube consists of a **lattice of cuboids**. Each cuboid corresponds to a different degree of summarization of the given multidimensional data.
- **Full materialization** refers to the computation of all of the cuboids in a data cube lattice. **Partial materialization** refers to the selective computation of a subset of the cuboids cells in the lattice. Iceberg cubes and shell fragments are examples of partial materialization. An **iceberg cube** is a data cube that stores only those cube cells whose aggregate value (e.g., count) is above some minimum support threshold. For **shell fragments** of a data cube, only some cuboids involving a small number of dimensions are computed. Queries on additional combinations of the dimensions can be computed on the fly.
- There are several efficient **data cube computation methods**. In this chapter, we discussed in depth four cube computation methods: (1) **MultiWay** array aggregation for materializing full data cubes in sparse-array-based, bottom-up, shared computation; (2) **BUC** for computing iceberg cubes by exploring ordering and sorting for efficient top-down computation; (3) **Star-Cubing** for integration of top-down and bottom-up computation using a star-tree structure; and (4) high-dimensional OLAP by precomputing only the partitioned shell fragments (thus called *minimal cubing*).
- There are several method for effective and efficient exploration of data cubes, including *discovery-driven cube exploration*, *multifeature data cubes*, and *constrained cube gradient analysis*. **Discovery-driven exploration** of data cubes uses precomputed measures and visual cues to indicate data exceptions at all levels of aggregation, guiding the user in the data analysis process. **Multifeature cubes** compute complex queries involving multiple dependent aggregates at multiple granularity. **Constrained cube gradient analysis** explores *significant changes in measures* in a multidimensional space, based on a given set of probe cells, where changes in sector characteristics are expressed in terms of dimensions of the cube and are limited to *specialization* (drill-down), *generalization* (roll-up), and *mutation* (a change in one of the cube's dimensions).

- **Concept description** is the most basic form of descriptive data mining. It describes a given set of task-relevant data in a concise and summarative manner, presenting interesting general properties of the data. Concept (or class) description consists of **characterization** and **comparison** (or **discrimination**). The former summarizes and describes a collection of data, called the **target class**, whereas the latter summarizes and distinguishes one collection of data, called the **target class**, from other collection(s) of data, collectively called the **contrasting class(es)**.
- **Concept characterization** can be implemented using **data cube (OLAP-based) approaches** and the **attribute-oriented induction approach**. These are attribute- or dimension-based generalization approaches. The **attribute-oriented induction approach** consists of the following techniques: *data focusing*, *data generalization by attribute removal or attribute generalization*, *count and aggregate value accumulation*, *attribute generalization control*, and *generalization data visualization*.
- **Concept comparison** can be performed using the attribute-oriented induction or data cube approaches in a manner similar to concept characterization. Generalized tuples from the target and contrasting classes can be quantitatively compared and contrasted.
- Characterization and comparison descriptions (which form a concept description) can both be presented in the *same* generalized relation, crosstab, or quantitative rule form, although they are displayed with different interestingness measures. These measures include the **t-weight** (for tuple typicality) and **d-weight** (for tuple discriminability).

## 4.5 Exercises

1. Assume a base cuboid of 10 dimensions contains only three base cells: (1)  $(a_1, d_2, d_3, d_4, \dots, d_9, d_{10})$ , (2)  $(d_1, b_2, d_3, d_4, \dots, d_9, d_{10})$ , and (3)  $(d_1, d_2, c_3, d_4, \dots, d_9, d_{10})$ , where  $a_1 \neq d_1$ ,  $b_2 \neq d_2$ , and  $c_3 \neq d_3$ . The measure of the cube is *count*.
  - (a) How many *nonempty* cuboids will a full data cube contain?
  - (b) How many *nonempty* aggregate (i.e., non-base) cells will a full cube contain?
  - (c) How many *nonempty* aggregate cells will an iceberg cube contain if the condition of the iceberg cube is “*count*  $\geq 2$ ”?
  - (d) A cell,  $\mathbf{c}$ , is a *closed cell* if there exists no cell,  $\mathbf{d}$ , such that  $\mathbf{d}$  is a specialization of cell  $\mathbf{c}$  (i.e.,  $\mathbf{d}$  is obtained by replacing a  $*$  in  $\mathbf{c}$  by a non- $*$  value) and  $\mathbf{d}$  has the same measure value as  $\mathbf{c}$ . A *closed cube* is a data cube consisting of only closed cells. How many closed cells are in the full cube?
2. There are several typical cube computation methods, such as *multiway array computation* [ZDN97], *BUC* (bottom-up computation) [BR99], and *Star-Cubing* [XHLW03].  
Briefly describe these three methods (i.e., use one or two lines to outline the key points), and compare their feasibility and performance under the following conditions:
  - (a) computing a dense full cube of low dimensionality (e.g., less than 8 dimensions),
  - (b) computing an iceberg cube of around 10 dimensions with a highly skewed data distribution, and
  - (c) computing a sparse iceberg cube of high dimensionality (e.g., over 100 dimensions).
3. [Contributed by Chen Chen] Suppose a data cube has  $D$  dimensions, and the base cuboid contains  $k$  distinct tuples.
  - (a) Present a formula to calculate the minimum number of cells that the cube,  $C$ , may contain.
  - (b) Present a formula to calculate the maximum number of cells that  $C$  may contain.
  - (c) Answer parts (a) and (b) above as if the count in each cube cell must be no less than a threshold,  $v$ .

- (d) Answer parts (a) and (b) above as if only closed cells are considered (with the minimum count threshold,  $v$ ).
4. Suppose that a base cuboid has three dimensions  $A, B, C$ , with the following number of cells:  $|A| = 1,000,000$ ,  $|B| = 100$ , and  $|C| = 1000$ . Suppose that each dimension is evenly partitioned into 10 portions for *chunking*.
    - (a) Assuming each dimension has only one level, draw the complete lattice of the cube.
    - (b) If each cube cell stores one measure with 4 bytes, what is the total size of the computed cube if the cube is *dense*?
    - (c) State the order for computing the chunks in the cube that requires the least amount of space, and compute the total amount of main memory space required for computing the 2-D planes.
  5. When computing a cube of high dimensionality, we encounter the inherent *curse of dimensionality* problem: there exists a huge number of subsets of combinations of dimensions.
    - (a) Suppose that there are only two base cells,  $\{(a_1, a_2, a_3, \dots, a_{100}), (a_1, a_2, b_3, \dots, b_{100})\}$ , in a 100-dimensional base cuboid. Compute the number of nonempty aggregate cells. Comment on the storage space and time required to compute these cells.
    - (b) Suppose we are to compute an iceberg cube from the above. If the minimum support count in the iceberg condition is two, how many aggregate cells will there be in the iceberg cube? Show the cells.
    - (c) Introducing iceberg cubes will lessen the burden of computing trivial aggregate cells in a data cube. However, even with iceberg cubes, we could still end up having to compute a large number of trivial uninteresting cells (i.e., with small counts). Suppose that a database has 20 tuples that map to (or cover) the two following base cells in a 100-dimensional base cuboid, each with a cell count of 10:  $\{(a_1, a_2, a_3, \dots, a_{100}) : 10, (a_1, a_2, b_3, \dots, b_{100}) : 10\}$ .
      - i. Let the minimum support be 10. How many distinct aggregate cells will there be like the following:  $\{(a_1, a_2, a_3, a_4, \dots, a_{99}, *) : 10, \dots, (a_1, a_2, *, a_4, \dots, a_{99}, a_{100}) : 10, \dots, (a_1, a_2, a_3, *, \dots, *, *) : 10\}$ ?
      - ii. If we ignore all the aggregate cells that can be obtained by replacing some constants by  $*$ 's while keeping the same measure value, how many distinct cells are left? What are the cells?
  6. Propose an algorithm that computes *closed iceberg cubes* efficiently.
  7. Suppose that we would like to compute an iceberg cube for the dimensions,  $A, B, C, D$ , where we wish to materialize all cells that satisfy a minimum support count of at least  $v$ , and where  $\text{cardinality}(A) < \text{cardinality}(B) < \text{cardinality}(C) < \text{cardinality}(D)$ . Show the BUC processing tree (which shows the order in which the BUC algorithm explores the lattice of a data cube, starting from all) for the construction of the above iceberg cube.
  8. Discuss how you might extend the *Star-Cubing* algorithm to compute iceberg cubes where the iceberg condition tests for *avg* that is no bigger than some value,  $v$ .
  9. A flight data warehouse for a travel agent consists of six dimensions: *traveler*, *departure (city)*, *departure\_time*, *arrival*, *arrival\_time*, and *flight*; and two measures: *count*, and *avg\_fare*, where *avg\_fare* stores the concrete fare at the lowest level but average fare at other levels.
    - (a) Suppose the cube is fully materialized. Starting with the *base cuboid* [*traveller*, *departure*, *departure\_time*, *arrival*, *arrival\_time*, *flight*], what *specific OLAP operations* (e.g., roll-up *flight* to *AA*) should one perform in order to list the average fare per month for *each business traveler* who flies American Airlines (*AA*) from L.A. in the year 2004?
    - (b) Suppose one wants to compute a data cube where the condition is that the minimum number of records is 10 and the average fare is over \$500. Outline an efficient cube computation method (based on commonsense about flight data distribution).
  10. (An implementation project) There are four typical data cube computation methods: multiway array aggregation [ZDN97], BUC [BR99], H-cubing [HPDW01], and Star-cubing [XHLW03].

- (a) Implement any one of these cube computation algorithms and describe your implementation, experimentation, and performance. Find another student who has implemented a different algorithm on the same platform (e.g., C++ on Linux) and compare your algorithm performance with his/hers.

Input:

- i. An  $n$ -dimensional base cuboid table (for  $n < 20$ ), which is essentially a relational table with  $n$  attributes;
- ii. An iceberg condition:  $\text{count}(C) \geq k$  where  $k$  is a positive integer as a parameter.

Output

- i. The set of computed cuboids that satisfy the iceberg condition, in the order of your output generation;
- ii. Summary of the set of cuboids in the form of “*cuboid ID*: the number of nonempty cells”, sorted in alphabetical order of cuboids, e.g.,  $A:155$ ,  $AB: 120$ ,  $ABC: 22$ ,  $ABCD: 4$ ,  $ABCE: 6$ ,  $ABD: 36$ , (where the number after “:” represents the number of nonempty cells).—This is used to quickly check the correctness of your results.

- (b) Based on your implementation, discuss the following:

- i. What challenging computation problems are encountered as the number of dimensions grows large?
- ii. How can iceberg cubing solve the problems of part (a) for some data sets (and characterize such data sets)?
- iii. Give one simple example to show that sometimes iceberg cubes cannot provide a good solution.

- (c) Instead of computing a data cube of high dimensionality, we may choose to materialize the cuboids having only a small number of dimension combinations. For example, for a 30-dimensional data cube, we may only compute all cuboids with five dimensions, for every possible 5-dimensional combination. The resulting cuboids form a *shell cube*. Discuss how easy or hard it is to modify your cube computation algorithm to facilitate such computation.

11. Consider the following *multifeature cube* query: Grouping by all subsets of  $\{\text{item}, \text{region}, \text{month}\}$ , find the minimum shelf life in 2004 for each group, and the fraction of the total sales due to tuples whose price is less than \$100, and whose shelf life is between 1.25 and 1.5 of the minimum shelf life.

(a) Draw the multifeature cube graph for the query.

(b) Express the query in extended SQL.

(c) Is this a *distributive* multifeature cube? Why or why not?

12. For *class characterization*, what are the major differences between a data cube-based implementation and a relational implementation such as attribute-oriented induction? Discuss which method is most efficient and under what conditions this is so.

13. Suppose that the following table is derived by *attribute-oriented induction*.

<i>class</i>	<i>birth_place</i>	<i>count</i>
Programmer	USA	180
	others	120
DBA	USA	20
	others	80

(a) Transform the table into a crosstab showing the associated t-weights and d-weights.

(b) Map the class *Programmer* into a (bidirectional) *quantitative descriptive rule*, for example,

$$\forall \mathbf{X}, \text{Programmer}(\mathbf{X}) \Leftrightarrow (\text{birth\_place}(\mathbf{X}) = \text{“USA”} \wedge \dots)[t : x\%, d : y\%] \dots \vee (\dots)[t : w\%, d : z\%].$$

14. Discuss why *relevance analysis* is beneficial and how it can be performed and integrated into the characterization process. Compare the result of two induction methods: (1) with relevance analysis and (2) without relevance analysis.

15. Given a generalized relation,  $R$ , derived from a database,  $DB$ , suppose that a set,  $\Delta DB$ , of tuples needs to be deleted from  $DB$ . Outline an *incremental* updating procedure for applying the necessary deletions to  $R$ .
16. Outline a data cube-based *incremental* algorithm for mining class comparisons.

## 4.6 Bibliographic Notes

Gray et al. [GCB<sup>+</sup>97] proposed the data cube as a relational aggregation operator generalizing group-by, crosstabs, and subtotals. Harinarayan, Rajaraman, and Ullman [HRU96] proposed a greedy algorithm for the partial materialization of cuboids in the computation of a data cube. Sarawagi and Stonebraker [SS94] developed a chunk-based computation technique for the efficient organization of large multidimensional arrays. Agarwal, Agrawal, Deshpande, et al. [AAD<sup>+</sup>96] proposed several methods for the efficient computation of multidimensional aggregates for ROLAP servers. The chunk-based MultiWay array aggregation method for data cube computation in MOLAP was proposed in Zhao, Deshpande, and Naughton [ZDN97]. Ross and Srivastava [RS97] developed a method for computing sparse data cubes. Iceberg queries are first described in Fang, Shivakumar, Garcia-Molina, et al. [FSGM<sup>+</sup>98]. BUC, a scalable method that computes iceberg cubes from the apex cuboid, downwards, was introduced by Beyer and Ramakrishnan [BR99]. Han, Pei, Dong, Wang [HPDW01] introduced an H-Cubing method for computing iceberg cubes with complex measures using an H-tree structure. The Star-Cubing method for computing iceberg cubes with a dynamic star-tree structure was introduced by Xin, Han, Li, and Wah [XHLW03]. MMCubing, an efficient iceberg cube computation method that factorizes the lattice space was developed by Shao, Han, and Xin [SHX04]. The shell-fragment-based minimal cubing approach for efficient high-dimensional OLAP introduced in this chapter was proposed by Li, Han and Gonzalez [LHG04].

Aside from computing iceberg cubes, another way to reduce data cube computation is to materialize condensed, dwarf, or quotient cubes, which are variants of closed cubes. Wang, Feng, Lu and Yu proposed computing a reduced data cube, called a *condensed cube* [WLFY02], Sismanis, Deligiannakis, Roussopoulos and Kotidis proposed computing a compressed data cube, called a *dwarf cube*. Lakeshmanan, Pei, and Han proposed a *quotient cube* structure to summarize the semantics of a data cube [LPH02], which has been further extended to a *qc-tree structure* by Lakshmanan, Pei, and Zhao [LPZ03].

There are also various studies on the computation of compressed data cubes by approximation, such as quasi-cubes by Barbara and Sullivan [BS97a], wavelet cubes by Vitter, Wang, and Iyer [VWI98], compressed cubes for query approximation on continuous dimensions by Shanmugasundaram, Fayyad, and Bradley [SFB99], and using log-linear models to compress data cubes by Barbara and Wu [BW00]. Computation of stream data “cubes” for multidimensional regression analysis has been studied by Chen, Dong, Han, et al. [CDH<sup>+</sup>02].

For works regarding the selection of materialized cuboids for efficient OLAP query processing, see Chaudhuri and Dayal [CD97], Harinarayan, Rajaraman, and Ullman [HRU96], and Sristava, Dar, Jagadish, and Levy [SDJL96], Gupta [Gup97], Baralis, Paraboschi, and Teniente [BPT97], and Shukla, Deshpande, and Naughton [SDN98]. Methods for cube size estimation can be found in Deshpande, Naughton, Ramasamy, et al. [DNR<sup>+</sup>97], Ross and Srivastava [RS97], and Beyer and Ramakrishnan [BR99]. Agrawal, Gupta, and Sarawagi [AGS97] proposed operations for modeling multidimensional databases.

The discovery-driven exploration of OLAP data cubes was proposed by Sarawagi, Agrawal, and Megiddo [SAM98]. Further studies on the integration of OLAP with data mining capabilities include the proposal of DIFF and RELAX operators for intelligent exploration of multidimensional OLAP data by Sarawagi and Sathe [SS00, SS01]. The construction of multifeature data cubes is described in Ross, Srivastava, and Chatziantoniou [RSC98]. Methods for answering queries quickly by on-line aggregation are described in Hellerstein, Haas, and Wang [HHW97] and Hellerstein, Avnur, Chou, et al. [HAC<sup>+</sup>99]. A cube-gradient analysis problem, called *cubegrade*, was first proposed by Imielinski, Khachiyan, and Abdulghani [IKA02]. An efficient method for multidimensional constrained gradient analysis in data cubes was studied by Dong, Han, Lam, et al. [DHL<sup>+</sup>01].

Generalization and concept description methods have been studied in the statistics literature long before the onset of computers. Good summaries of statistical descriptive data mining methods include Cleveland [Cle93] and Devore [Dev95]. Generalization-based induction techniques, such as learning from examples, were proposed

and studied in the machine learning literature before data mining became active. A theory and methodology of inductive learning was proposed by Michalski [Mic83]. The learning-from-examples method was proposed by Michalski [Mic83]. Version space was proposed by Mitchell [Mit77, Mit82]. The method of factoring the version space was presented by Subramanian and Feigenbaum [SF86b]. Overviews of machine learning techniques can be found in Dietterich and Michalski [DM83], Michalski, Carbonell, and Mitchell [MCM86], and Mitchell [Mit97].

Database-oriented methods for concept description explore scalable and efficient techniques for describing large sets of data. The attribute-oriented induction method described in this chapter was first proposed by Cai, Cercone, and Han [CCH91] and further extended by Han, Cai, and Cercone [HCC93], Han and Fu [HF96], Carter and Hamilton [CH98], and Han, Nishio, Kawano, and Wang [HNKW98].



## Chapter 5

# Mining Frequent Patterns, Associations, and Correlations

**Frequent patterns** are patterns (such as itemsets, subsequences, or substructures) that appear in a data set frequently. For example, a set of items, such as milk and bread, that appear frequently together in a transaction data set is a *frequent itemset*. A subsequence, such as buying first a PC, then a digital camera, and then a memory card, if it occurs frequently in a shopping history database, is a (*frequent*) *sequential pattern*. A *substructure* can refer to different structural forms, such as subgraphs, subtrees, or sublattices, which may be combined with itemsets or subsequences. If a substructure occurs frequently, it is called a (*frequent*) *structured pattern*. Finding such frequent patterns plays an essential role in mining associations, correlations, and many other interesting relationships among data. Moreover, it helps in data classification, clustering, and other data mining tasks as well. Thus, frequent pattern mining has become an important data mining task and a focused theme in data mining research.

In this chapter, we introduce the concepts of frequent patterns, associations and correlations, and study how they can be mined efficiently. The topic of frequent pattern mining is indeed rich. This chapter is dedicated to methods of *frequent itemset mining*. We delve into the following questions: How can we find frequent itemsets from large amounts of data, where the data are either transactional or relational? How can we mine association rules in multilevel and multidimensional space? Which association rules are the most interesting? How can we help or guide the mining procedure to discover interesting associations or correlations? How can we take advantage of user preferences or constraints to speed up the mining process? The techniques learned in this chapter may also be extended for more advanced forms of frequent pattern mining, such as from sequential and structured data sets, as we will study in later chapters.

### 5.1 Basic Concepts and a Road Map

Frequent pattern mining searches for recurring relationships in a given data set. This section introduces the basic concepts of frequent pattern mining for the discovery of interesting associations and correlations between itemsets in transactional and relational databases. We begin in Section 5.1.1 by presenting an example of market basket analysis, the earliest form of frequent pattern mining for association rules. The basic concepts of mining frequent patterns and associations are given in Section 5.1.2. Section 5.1.3 presents a road map to the different kinds of frequent patterns, association rules, and correlation rules that can be mined.

### 5.1.1 Market Basket Analysis: A Motivating Example

Frequent itemset mining leads to the discovery of associations and correlations among items in large transactional or relational data sets. With massive amounts of data continuously being collected and stored, many industries are becoming interested in mining such patterns from their databases. The discovery of interesting correlation relationships among huge amounts of business transaction records can help in many business decision making processes, such as catalog design, cross-marketing, and customer shopping behavior analysis.

A typical example of frequent itemset mining is **market basket analysis**. This process analyzes customer buying habits by finding associations between the different items that customers place in their “shopping baskets” (Figure 5.1). The discovery of such associations can help retailers develop marketing strategies by gaining insight into which items are frequently purchased together by customers. For instance, if customers are buying milk, how likely are they to also buy bread (and what kind of bread) on the same trip to the supermarket? Such information can lead to increased sales by helping retailers do selective marketing and plan their shelf space.

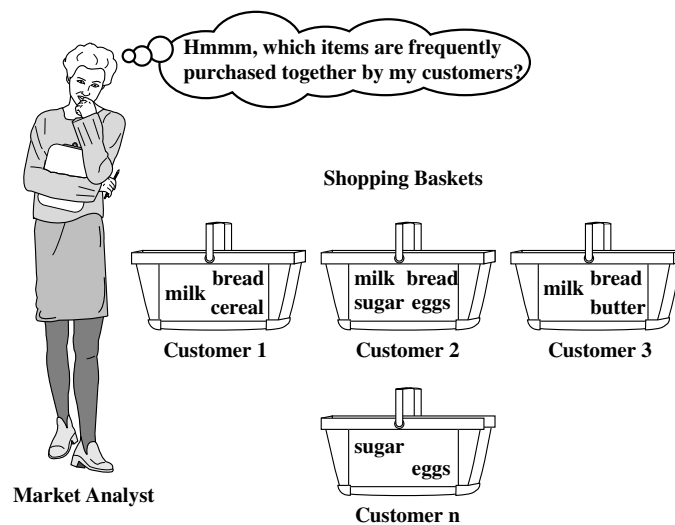


Figure 5.1: Market basket analysis.

Let’s look at an example of how market basket analysis can be useful.

**Example 5.1 Market basket analysis.** Suppose, as manager of an *AllElectronics* branch, you would like to learn more about the buying habits of your customers. Specifically, you wonder, “Which groups or sets of items are customers likely to purchase on a given trip to the store?” To answer your question, market basket analysis may be performed on the retail data of customer transactions at your store. You can then make use of the results to plan marketing or advertising strategies, or in the design of a new catalog. For instance, market basket analysis may help you design different store layouts. In one strategy, items that are frequently purchased together can be placed in close proximity in order to further encourage the sale of such items together. If customers who purchase computers also tend to buy antivirus software at the same time, then placing the hardware display close to the software display may help to increase the sales of both of these items. In an alternative strategy, placing hardware and software at opposite ends of the store may entice customers who purchase such items to pick up other items along the way. For instance, after deciding on an expensive computer, a customer may observe security systems for sale while heading towards the software display to purchase antivirus software and may decide to purchase a home security system as well. Market basket analysis can also help retailers to plan which items to put on sale at reduced prices. If customers tend to purchase computers and printers together, then having a sale on printers may encourage the sale of printers *as well as* computers. ■

If we think of the universe as the set of items available at the store, then each item has a Boolean variable

representing the presence or absence of that item. Each basket can then be represented by a Boolean vector of values assigned to these variables. The Boolean vectors can be analyzed for buying patterns that reflect items that are frequently *associated* or purchased together. These patterns can be represented in the form of **association rules**. For example, the information that customers who purchase computers also tend to buy antivirus software at the same time is represented in Association Rule (5.1) below:

$$\text{computer} \Rightarrow \text{antivirus\_software} \quad [\text{support} = 2\%, \text{confidence} = 60\%] \quad (5.1)$$

Rule **support** and **confidence** are two measures of rule interestingness. They respectively reflect the usefulness and certainty of discovered rules. A support of 2% for Association Rule (5.1) means that 2% of all the transactions under analysis show that computer and antivirus software are purchased together. A confidence of 60% means that 60% of the customers who purchased a computer also bought the software. Typically, association rules are considered interesting if they satisfy both a **minimum support threshold** and a **minimum confidence threshold**. Such thresholds can be set by users or domain experts. Additional analysis can be performed to uncover interesting statistical correlations between associated items.

### 5.1.2 Frequent Itemsets, Closed Itemsets, and Association Rules

Let  $\mathcal{I} = \{I_1, I_2, \dots, I_m\}$  be a set of items. Let  $D$ , the task-relevant data, be a set of database transactions where each transaction  $T$  is a set of items such that  $T \subseteq \mathcal{I}$ . Each transaction is associated with an identifier, called TID. Let  $A$  be a set of items. A transaction  $T$  is said to contain  $A$  if and only if  $A \subseteq T$ . An association rule is an implication of the form  $A \Rightarrow B$ , where  $A \subset \mathcal{I}$ ,  $B \subset \mathcal{I}$ , and  $A \cap B = \phi$ . The rule  $A \Rightarrow B$  holds in the transaction set  $D$  with **support**  $s$ , where  $s$  is the percentage of transactions in  $D$  that contain  $A \cup B$  (i.e., the *union* of sets  $A$  and  $B$ , or say, both  $A$  and  $B$ ). This is taken to be the probability,  $P(A \cup B)$ .<sup>1</sup> The rule  $A \Rightarrow B$  has **confidence**  $c$  in the transaction set  $D$ , where  $c$  is the percentage of transactions in  $D$  containing  $A$  that also contain  $B$ . This is taken to be the conditional probability,  $P(B|A)$ . That is,

$$\text{support}(A \Rightarrow B) = P(A \cup B) \quad (5.2)$$

$$\text{confidence}(A \Rightarrow B) = P(B|A). \quad (5.3)$$

Rules that satisfy both a minimum support threshold (*min\_sup*) and a minimum confidence threshold (*min\_conf*) are called **strong**. By convention, we write support and confidence values so as to occur between 0% and 100%, rather than 0 to 1.0.

A set of items is referred to as an **itemset**.<sup>2</sup> An itemset that contains  $k$  items is a  **$k$ -itemset**. The set  $\{\text{computer}, \text{antivirus\_software}\}$  is a 2-itemset. The **occurrence frequency of an itemset** is the number of transactions that contain the itemset. This is also known, simply, as the **frequency**, **support count**, or **count** of the itemset. Note that the itemset support defined in Equation (5.2) is sometimes referred to as *relative support*, whereas the occurrence frequency is called the **absolute support**. If the relative support of an itemset  $I$  satisfies a pre-specified **minimum support threshold** (i.e., the absolute support of  $I$  satisfies the corresponding **minimum support count threshold**), then  $I$  is a **frequent** itemset.<sup>3</sup> The set of frequent  $k$ -itemsets is commonly denoted by  $L_k$ .<sup>4</sup>

From Equation (5.3), we have

$$\text{confidence}(A \Rightarrow B) = P(B|A) = \frac{\text{support}(A \cup B)}{\text{support}(A)} = \frac{\text{support\_count}(A \cup B)}{\text{support\_count}(A)}. \quad (5.4)$$

<sup>1</sup>Notice that the notation  $P(A \cup B)$  indicates the probability that a transaction contains the *union* of set  $A$  and set  $B$ , i.e., it contains every item in  $A$  and in  $B$ . This should not be confused with  $P(A \text{ or } B)$ , which indicates the probability that a transaction contains either  $A$  or  $B$ .

<sup>2</sup>In the data mining research literature, “itemset” is more commonly used than “item set.”

<sup>3</sup>In early work, itemsets satisfying minimum support were referred to as **large**. This term, however, is somewhat confusing as it has connotations to the number of items in an itemset rather than the frequency of occurrence of the set. Hence, we use the more recent term **frequent**.

<sup>4</sup>Although the term **frequent** is preferred over **large**, for historical reasons frequent  $k$ -itemsets are still denoted as  $L_k$ .

Equation (5.4) shows that the confidence of rule  $A \Rightarrow B$  can be easily derived from the support counts of  $A$  and  $A \cup B$ . That is, once the support counts of  $A$ ,  $B$  and  $A \cup B$  are found, it is straightforward to derive the corresponding association rules  $A \Rightarrow B$  and  $B \Rightarrow A$  and check whether they are strong. Thus the problem of mining association rules can be reduced to that of mining frequent itemsets.

In general, association rule mining can be viewed as a two-step process:

1. **Find all frequent itemsets:** By definition, each of these itemsets will occur at least as frequently as a pre-determined minimum support count,  $min\_sup$ .
2. **Generate strong association rules from the frequent itemsets:** By definition, these rules must satisfy minimum support and minimum confidence.

Additional interestingness measures can be applied for the discovery of correlation relationships between associated items, as will be discussed in Section 5.4. Since the second step is much less costly than the first, the overall performance of mining association rules is determined by the first step.

A major challenge in mining frequent itemsets from a large data set is the fact that such mining often generates a huge number of itemsets satisfying the minimum support ( $min\_sup$ ) threshold, especially when  $min\_sup$  is set low. This is because if an itemset is frequent, each of its subsets is frequent as well. A long itemset will contain a combinatorial number of shorter, frequent sub-itemsets. For example, a frequent itemset of length 100, such as  $\{a_1, a_2, \dots, a_{100}\}$ , contains  $\binom{100}{1} = 100$  frequent 1-itemsets:  $a_1, a_2, \dots, a_{100}$ ,  $\binom{100}{2}$  frequent 2-itemsets:  $(a_1, a_2), (a_1, a_3) \dots, (a_{99}, a_{100})$ , and so on. The total number of frequent itemsets that it contains is thus,

$$\binom{100}{1} + \binom{100}{2} + \dots + \binom{100}{100} = 2^{100} - 1 \approx 1.27 \times 10^{30}. \quad (5.5)$$

This is too huge a number of itemsets for any computer to compute or store. To overcome this difficulty, we introduce the concepts of *closed frequent itemset* and *maximal frequent itemset*.

An itemset  $X$  is **closed** in a data set  $S$  if there exists no proper super-itemset<sup>5</sup>  $Y$  such that  $Y$  has the same support count as  $X$  in  $S$ . An itemset  $X$  is a **closed frequent itemset** in set  $S$  if  $X$  is both closed and frequent in set  $S$ . An itemset  $X$  is a **maximal frequent itemset** (or **max-itemset**) in set  $S$  if  $X$  is frequent, and there exists no super-itemset  $Y$  such that  $X \subset Y$  and  $Y$  is frequent in  $S$ .

Let  $\mathcal{C}$  be the set of closed frequent itemsets for a data set  $S$  satisfying a minimum support threshold,  $min\_sup$ . Let  $\mathcal{M}$  be the set of maximal frequent itemsets for  $S$  satisfying  $min\_sup$ . Suppose that we have the support count of each itemset in  $\mathcal{C}$  and  $\mathcal{M}$ . Notice that  $\mathcal{C}$  and its count information can be used to derive the whole set of frequent itemsets. Thus we say that  $\mathcal{C}$  contains complete information regarding its corresponding frequent itemsets. On the other hand,  $\mathcal{M}$ , registers only the support of the maximal itemsets. It usually does not contain the complete support information regarding its corresponding frequent itemsets. We illustrate these concepts with the following example.

**Example 5.2 Closed and maximal frequent itemsets.** Suppose that a transaction database has only two transactions:  $\{\langle a_1, a_2, \dots, a_{100} \rangle; \langle a_1, a_2, \dots, a_{50} \rangle\}$ . Let the minimum support count threshold be  $min\_sup = 1$ . We find two closed frequent itemsets and their support counts, that is  $\mathcal{C} = \{\{a_1, a_2, \dots, a_{100}\} : 1; \{a_1, a_2, \dots, a_{50}\} : 2\}$ . There is one maximal frequent itemset:  $\mathcal{M} = \{\{a_1, a_2, \dots, a_{100}\} : 1\}$ . (We cannot include  $\{a_1, a_2, \dots, a_{50}\}$  as a maximal frequent itemset since it has a frequent super-set,  $\{a_1, a_2, \dots, a_{100}\}$ .) Compare this to the above, where we determined that there are  $2^{100} - 1$  frequent itemsets, which is too huge a set to be enumerated anywhere!

The set of closed frequent itemsets contains complete information regarding the frequent itemsets. For example, from  $\mathcal{C}$ , we can derive, say, (1)  $\{a_2, a_{45} : 2\}$  since  $\{a_2, a_{45}\}$  is a sub-itemset of the itemset  $\{a_1, a_2, \dots, a_{50} : 2\}$ ; and

<sup>5</sup> $Y$  is a proper super-itemset of  $X$  if  $X$  is a proper sub-itemset of  $Y$ , that is, if  $X \subset Y$ . In other words, every item of  $X$  is contained in  $Y$  but there is at least one item of  $Y$  that is not in  $X$ .

(2)  $\{a_8, a_{55} : 1\}$  since  $\{a_8, a_{55}\}$  is not a sub-itemset of the previous itemset but of the itemset  $\{a_1, a_2, \dots, a_{100} : 1\}$ . However, from the maximal frequent itemset, we can only assert that both itemsets ( $\{a_2, a_{45}\}$  and  $\{a_8, a_{55}\}$ ) are frequent, but we cannot assert their actual support counts. ■

### 5.1.3 Frequent Pattern Mining: A Road Map

Market basket analysis is just one form of frequent pattern mining. In fact, there are many kinds of frequent patterns, association rules and correlation relationships. Frequent pattern mining can be classified in various ways, based on the following criteria:

- **Based on the *completeness* of patterns to be mined:** As we discussed in the previous subsection, we can mine the **complete set of frequent itemsets**, the **closed frequent itemsets**, and the **maximal frequent itemsets**, given a minimum support threshold. We can also mine **constrained frequent itemsets** (i.e., those that satisfy a set of user-defined constraints), **approximate frequent itemsets** (i.e., those that derive only approximate support counts for the mined frequent itemsets), **near-match frequent itemsets** (i.e., those that tally the support count of the near or almost matching itemsets), **top- $k$  frequent itemsets** (i.e., the  $k$  most frequent itemsets for a user-specified value,  $k$ ), and so on.

Different applications may have different requirements regarding the completeness of the patterns to be mined, which in turn, can lead to different evaluation and optimization methods. In this chapter, our study of mining methods focuses on mining the *complete set of frequent itemsets*, *closed frequent itemsets*, and *constrained frequent itemsets*. We leave the mining of frequent itemsets under other completeness requirements as an exercise.

- **Based on the *levels of abstraction* involved in the rule set:** Some methods for association rule mining can find rules at differing levels of abstraction. For example, suppose that a set of association rules mined includes the following rules where  $X$  is a variable representing a customer:

$$\text{buys}(X, \text{"computer"}) \Rightarrow \text{buys}(X, \text{"HP\_printer"}) \quad (5.6)$$

$$\text{buys}(X, \text{"laptop\_computer"}) \Rightarrow \text{buys}(X, \text{"HP\_printer"}) \quad (5.7)$$

In Rules (5.6) and (5.7), the items bought are referenced at different levels of abstraction (e.g., “computer” is a higher-level abstraction of “laptop computer”.) We refer to the rule set mined as consisting of **multilevel association rules**. If, instead, the rules within a given set do not reference items or attributes at different levels of abstraction, then the set contains **single-level association rules**.

- **Based on the *number of data dimensions* involved in the rule:** If the items or attributes in an association rule reference only one dimension, then it is a **single-dimensional association rule**. Note that Rule (5.1), for example, could be rewritten as Rule (5.8):

$$\text{buys}(X, \text{"computer"}) \Rightarrow \text{buys}(X, \text{"antivirus\_software"}) \quad (5.8)$$

Rules (5.6), (5.7), and (5.8) are single-dimensional association rules since they each refer to only one dimension, *buys*.<sup>6</sup>

If a rule references two or more dimensions, such as the dimensions *age*, *income*, and *buys*, then it is a **multidimensional association rule**. The following rule is an example of a multidimensional rule.

$$\text{age}(X, \text{"30...39"}) \wedge \text{income}(X, \text{"42K...48K"}) \Rightarrow \text{buys}(X, \text{"high resolution TV"}) \quad (5.9)$$

<sup>6</sup>Following the terminology used in multidimensional databases, we refer to each distinct predicate in a rule as a *dimension*.

- **Based on the *types of values* handled in the rule:** If a rule involves associations between the presence or absence of items, it is a **Boolean association rule**. For example, Rules (5.1), (5.6), and (5.7) are Boolean association rules obtained from market basket analysis.

If a rule describes associations between quantitative items or attributes, then it is a **quantitative association rule**. In these rules, quantitative values for items or attributes are partitioned into intervals. Rule (5.9) above is also considered a quantitative association rule. Note that the quantitative attributes, *age* and *income*, have been discretized.

- **Based on the *kinds of rules* to be mined:** Frequent pattern analysis can generate various kinds of rules and other interesting relationships. **Association rules** are the most popular kind of rules generated from frequent patterns. Typically, such mining can generate a large number of rules, many of which are redundant or do not indicate a correlation relationship among itemsets. Thus, the discovered associations can be further analyzed to uncover statistical correlations, leading to **correlation rules**.

We can also mine **strong gradient relationships** among itemsets, where a gradient is the ratio of the measure of an item when compared with that of its parent (a generalized itemset), its child (a specialized itemset) or its sibling (a comparable itemset). One such example could be: The average sales from Sony\_Digital\_Camera increases over 16% when sold together with Sony\_Laptop\_Computer: both are siblings of the parent itemset, Sony.

- **Based on the *kinds of patterns* to be mined:** There are many kinds of frequent patterns that can be mined from different kinds of data sets. For this chapter, our focus is on **frequent itemset mining**, that is, the mining of frequent itemsets (sets of items) from transactional or relational data sets. However, other kinds of frequent patterns can be found from other kinds of data sets. **Sequential pattern mining** searches for frequent *subsequences* in a *sequence data set*, where a sequence records an ordering of events. For example, with sequential pattern mining, we can study the order in which items are frequently purchased. For instance, customers may tend to first buy a PC, followed by a digital camera, and then a memory card. **Structured pattern mining** searches for frequent *substructures* in a *structured data set*. Notice that *structure* is a general concept that covers many different kinds of structural forms, such as graphs, lattices, trees, sequences, sets, single items, or combinations of such structures. Single items are the simplest form of structure. Each element of an itemset may contain a subsequence, a subtree, etc., and such containment relationships can be defined recursively. Therefore, structured pattern mining can be considered as the most general form of frequent pattern mining.

In the next section, we will study efficient methods for mining the basic (i.e., single-level, single-dimensional, Boolean) frequent itemsets from transactional databases, and show how to generate association rules from such itemsets. The extension of this scope of mining to multi-level, multi-dimensional, and quantitative rules is discussed in Section 5.3. The mining of strong-correlation relationships is studied in Section 5.4. Constraint-based mining is studied in Section 5.5. We address the more advanced topic of mining sequence and structured patterns in later chapters. Nevertheless, most of the methods studied here can be easily extended for the mining of more complex kinds of patterns.

## 5.2 Efficient and Scalable Frequent Itemset Mining Methods

In this section, you will learn methods for mining the simplest form of frequent patterns—*single-dimensional, single-level, Boolean frequent itemsets*, such as those discussed for market basket analysis in Section 5.1.1. We begin by presenting **Apriori**, a basic algorithm for finding frequent itemsets (Section 5.2.1). In Section 5.2.2, we look at how to generate strong association rules from frequent itemsets. Section 5.2.3 describes several variations to the Apriori algorithm for improved efficiency and scalability. Section 5.2.4 presents methods for mining frequent itemsets that, unlike Apriori, do not involve the generation of “candidate” frequent itemsets. Section 5.2.5 presents methods for mining frequent itemsets that take advantage of vertical data format. Methods for mining closed frequent itemsets are discussed in Section 5.2.6.

### 5.2.1 The Apriori Algorithm: Finding Frequent Itemsets Using Candidate Generation

**Apriori** is a seminal algorithm proposed by R. Agrawal and R. Srikant in 1994 for mining frequent itemsets for Boolean association rules. The name of the algorithm is based on the fact that the algorithm uses *prior knowledge* of frequent itemset properties, as we shall see below. Apriori employs an iterative approach known as a *level-wise* search, where  $k$ -itemsets are used to explore  $(k + 1)$ -itemsets. First, the set of frequent 1-itemsets is found by scanning the database to accumulate the count for each item, and collecting those items that satisfy minimum support. The resulting set is denoted  $L_1$ . Next,  $L_1$  is used to find  $L_2$ , the set of frequent 2-itemsets, which is used to find  $L_3$ , and so on, until no more frequent  $k$ -itemsets can be found. The finding of each  $L_k$  requires one full scan of the database.

To improve the efficiency of the level-wise generation of frequent itemsets, an important property called the **Apriori property**, presented below, is used to reduce the search space. We will first describe this property, and then show an example illustrating its use.

[TO EDITOR Please Note: There were some errors/misunderstandings regarding the typesetting of this property in edition 1! We would like it to be one line, on its own (e.g., treated as a paragraph), not joined with the paragraph following it. We would like the words ‘Apriori property’ to be in boldface and the sentence following it to be italicized. Thank you!]

**Apriori property:** *All nonempty subsets of a frequent itemset must also be frequent.*

The Apriori property is based on the following observation. By definition, if an itemset  $I$  does not satisfy the minimum support threshold,  $\text{min\_sup}$ , then  $I$  is not frequent, that is,  $P(I) < \text{min\_sup}$ . If an item  $A$  is added to the itemset  $I$ , then the resulting itemset (i.e.,  $I \cup A$ ) cannot occur more frequently than  $I$ . Therefore,  $I \cup A$  is not frequent either, that is,  $P(I \cup A) < \text{min\_sup}$ .

This property belongs to a special category of properties called **antimonotone** in the sense that *if a set cannot pass a test, all of its supersets will fail the same test as well*. It is called *antimonotone* because the property is monotonic in the context of failing a test.<sup>7</sup>

“How is the Apriori property used in the algorithm?” To understand this, let us look at how  $L_{k-1}$  is used to find  $L_k$  for  $k \geq 2$ . A two-step process is followed, consisting of **join** and **prune** actions.

1. **The join step:** To find  $L_k$ , a set of **candidate**  $k$ -itemsets is generated by joining  $L_{k-1}$  with itself. This set of candidates is denoted  $C_k$ . Let  $l_1$  and  $l_2$  be itemsets in  $L_{k-1}$ . The notation  $l_i[j]$  refers to the  $j$ th item in  $l_i$  (e.g.,  $l_1[k-2]$  refers to the second to the last item in  $l_1$ ). By convention, Apriori assumes that items within a transaction or itemset are sorted in lexicographic order. For the  $(k-1)$ -itemset,  $l_i$ , this means that the items are sorted such that  $l_i[1] < l_i[2] < \dots < l_i[k-1]$ . The join,  $L_{k-1} \bowtie L_{k-1}$ , is performed, where members of  $L_{k-1}$  are joinable if their first  $(k-2)$  items are in common. That is, members  $l_1$  and  $l_2$  of  $L_{k-1}$  are joined if  $(l_1[1] = l_2[1]) \wedge (l_1[2] = l_2[2]) \wedge \dots \wedge (l_1[k-2] = l_2[k-2]) \wedge (l_1[k-1] < l_2[k-1])$ . The condition  $l_1[k-1] < l_2[k-1]$  simply ensures that no duplicates are generated. The resulting itemset formed by joining  $l_1$  and  $l_2$  is  $l_1[1]l_1[2] \dots l_1[k-2]l_2[k-1]$ .
2. **The prune step:**  $C_k$  is a superset of  $L_k$ , that is, its members may or may not be frequent, but all of the frequent  $k$ -itemsets are included in  $C_k$ . A scan of the database to determine the count of each candidate in  $C_k$  would result in the determination of  $L_k$  (i.e., all candidates having a count no less than the minimum support count are frequent by definition, and therefore belong to  $L_k$ ).  $C_k$ , however, can be huge, and so this could involve heavy computation. To reduce the size of  $C_k$ , the Apriori property is used as follows. Any  $(k-1)$ -itemset that is not frequent cannot be a subset of a frequent  $k$ -itemset. Hence, if any  $(k-1)$ -subset of a candidate  $k$ -itemset is not in  $L_{k-1}$ , then the candidate cannot be frequent either and so can be removed from  $C_k$ . This **subset testing** can be done quickly by maintaining a hash tree of all frequent itemsets.

<sup>7</sup>The Apriori property has many applications. It can also be used to prune search during data cube computation (Chapter 4).

Table 5.1: Transactional data for an *AllElectronics* branch.

<i>TID</i>	<i>List of item-IDs</i>
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

**Example 5.3 Apriori.** Let's look at a concrete example, based on the *AllElectronics* transaction database,  $D$ , of Table 5.1. There are nine transactions in this database, that is,  $|D| = 9$ . We use Figure 5.2 to illustrate the Apriori algorithm for finding frequent itemsets in  $D$ .

1. In the first iteration of the algorithm, each item is a member of the set of candidate 1-itemsets,  $C_1$ . The algorithm simply scans all of the transactions in order to count the number of occurrences of each item.
2. Suppose that the minimum support count required is 2 (i.e.,  $\text{min\_sup\_count} = 2$ , or  $\text{min\_sup} = 2/9 = 22\%$ ). The set of frequent 1-itemsets,  $L_1$ , can then be determined. It consists of the candidate 1-itemsets satisfying minimum support. In our example, all of the candidates in  $C_1$  satisfy minimum support.
3. To discover the set of frequent 2-itemsets,  $L_2$ , the algorithm uses the join  $L_1 \bowtie L_1$  to generate a candidate set of 2-itemsets,  $C_2$ .<sup>8</sup>  $C_2$  consists of  $\binom{|L_1|}{2}$  2-itemsets. Note that no candidates are removed from  $C_2$  during the prune step since each subset of the candidates is also frequent.
4. Next, the transactions in  $D$  are scanned and the support count of each candidate itemset in  $C_2$  is accumulated, as shown in the middle table of the second row in Figure 5.2.
5. The set of frequent 2-itemsets,  $L_2$ , is then determined, consisting of those candidate 2-itemsets in  $C_2$  having minimum support.
6. The generation of the set of candidate 3-itemsets,  $C_3$ , is detailed in Figure 5.3. From the join step, we first get  $C_3 = L_2 \bowtie L_2 = \{\{I1, I2, I3\}, \{I1, I2, I5\}, \{I1, I3, I5\}, \{I2, I3, I4\}, \{I2, I3, I5\}, \{I2, I4, I5\}\}$ . Based on the Apriori property that all subsets of a frequent itemset must also be frequent, we can determine that the four latter candidates cannot possibly be frequent. We therefore remove them from  $C_3$ , thereby saving the effort of unnecessarily obtaining their counts during the subsequent scan of  $D$  to determine  $L_3$ . Note that when given a candidate  $k$ -itemset, we only need to check if its  $(k - 1)$ -subsets are frequent since the Apriori algorithm uses a level-wise search strategy. The resulting pruned version of  $C_3$  is shown in the first table of the bottom row of Figure 5.2.
7. The transactions in  $D$  are scanned in order to determine  $L_3$ , consisting of those candidate 3-itemsets in  $C_3$  having minimum support (Figure 5.2).
8. The algorithm uses  $L_3 \bowtie L_3$  to generate a candidate set of 4-itemsets,  $C_4$ . Although the join results in  $\{\{I1, I2, I3, I5\}\}$ , this itemset is pruned since its subset  $\{\{I2, I3, I5\}\}$  is not frequent. Thus,  $C_4 = \phi$ , and the algorithm terminates, having found all of the frequent itemsets. ■

Figure 5.4 shows pseudocode for the Apriori algorithm and its related procedures. Step 1 of Apriori finds the frequent 1-itemsets,  $L_1$ . In steps 2–10,  $L_{k-1}$  is used to generate candidates  $C_k$  in order to find  $L_k$  for  $k \geq 2$ . The `apriori_gen` procedure generates the candidates and then uses the Apriori property to eliminate those having a subset

<sup>8</sup> $L_1 \bowtie L_1$  is equivalent to  $L_1 \times L_1$  since the definition of  $L_k \bowtie L_k$  requires the two joining itemsets to share  $k - 1 = 0$  items.



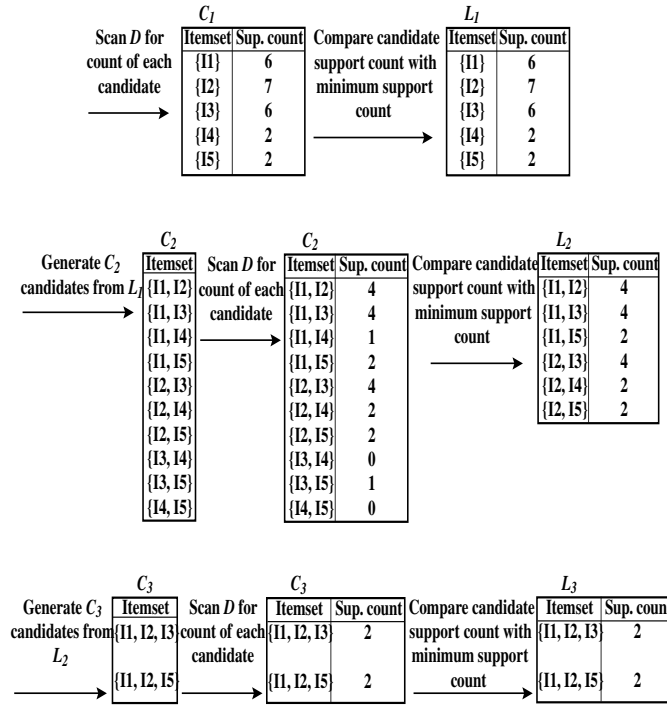


Figure 5.2: Generation of candidate itemsets and frequent itemsets, where the minimum support count is 2.

that is not frequent (step 3). This procedure is described below. Once all the candidates have been generated, the database is scanned (step 4). For each transaction, a **subset** function is used to find all subsets of the transaction that are candidates (step 5), and the count for each of these candidates is accumulated (steps 6 and 7). Finally, all those candidates satisfying minimum support form the set of frequent itemsets,  $L$ . A procedure can then be called to generate association rules from the frequent itemsets. Such a procedure is described in Section 5.2.2.

The **apriori\_gen** procedure performs two kinds of actions, namely, **join** and **prune**, as described above. In the join component,  $L_{k-1}$  is joined with  $L_{k-1}$  to generate potential candidates (steps 1–4). The prune component (steps 5–7) employs the Apriori property to remove candidates that have a subset that is not frequent. The test for infrequent subsets is shown in procedure **has\_infrequent\_subset**.

### 5.2.2 Generating Association Rules from Frequent Itemsets

Once the frequent itemsets from transactions in a database  $D$  have been found, it is straightforward to generate strong association rules from them (where *strong* association rules satisfy both minimum support and minimum confidence). This can be done using Equation (5.4) for confidence, which we show again here for completeness:

$$\text{confidence}(A \Rightarrow B) = P(B|A) = \frac{\text{support\_count}(A \cup B)}{\text{support\_count}(A)},$$

The conditional probability is expressed in terms of itemset support count, where  $\text{support\_count}(A \cup B)$  is the number of transactions containing the itemsets  $A \cup B$ , and  $\text{support\_count}(A)$  is the number of transactions containing the itemset  $A$ . Based on this equation, association rules can be generated as follows:

- For each frequent itemset  $l$ , generate all nonempty subsets of  $l$ .
- For every nonempty subset  $s$  of  $l$ , output the rule “ $s \Rightarrow (l - s)$ ” if  $\frac{\text{support\_count}(l)}{\text{support\_count}(s)} \geq \text{min\_conf}$ , where  $\text{min\_conf}$  is the minimum confidence threshold.

- (a) Join:  $C_3 = L_2 \bowtie L_2 = \{\{I1, I2\}, \{I1, I3\}, \{I1, I5\}, \{I2, I3\}, \{I2, I4\}, \{I2, I5\}\} \bowtie \{\{I1, I2\}, \{I1, I3\}, \{I1, I5\}, \{I2, I3\}, \{I2, I4\}, \{I2, I5\}\} = \{\{I1, I2, I3\}, \{I1, I2, I5\}, \{I1, I3, I5\}, \{I2, I3, I4\}, \{I2, I3, I5\}, \{I2, I4, I5\}\}.$
- (b) Prune using the Apriori property: All nonempty subsets of a frequent itemset must also be frequent. Do any of the candidates have a subset that is not frequent?
- The 2-item subsets of  $\{I1, I2, I3\}$  are  $\{I1, I2\}$ ,  $\{I1, I3\}$ , and  $\{I2, I3\}$ . All 2-item subsets of  $\{I1, I2, I3\}$  are members of  $L_2$ . Therefore, keep  $\{I1, I2, I3\}$  in  $C_3$ .
  - The 2-item subsets of  $\{I1, I2, I5\}$  are  $\{I1, I2\}$ ,  $\{I1, I5\}$ , and  $\{I2, I5\}$ . All 2-item subsets of  $\{I1, I2, I5\}$  are members of  $L_2$ . Therefore, keep  $\{I1, I2, I5\}$  in  $C_3$ .
  - The 2-item subsets of  $\{I1, I3, I5\}$  are  $\{I1, I3\}$ ,  $\{I1, I5\}$ , and  $\{I3, I5\}$ .  $\{I3, I5\}$  is not a member of  $L_2$ , and so it is not frequent. Therefore, remove  $\{I1, I3, I5\}$  from  $C_3$ .
  - The 2-item subsets of  $\{I2, I3, I4\}$  are  $\{I2, I3\}$ ,  $\{I2, I4\}$ , and  $\{I3, I4\}$ .  $\{I3, I4\}$  is not a member of  $L_2$ , and so it is not frequent. Therefore, remove  $\{I2, I3, I4\}$  from  $C_3$ .
  - The 2-item subsets of  $\{I2, I3, I5\}$  are  $\{I2, I3\}$ ,  $\{I2, I5\}$ , and  $\{I3, I5\}$ .  $\{I3, I5\}$  is not a member of  $L_2$ , and so it is not frequent. Therefore, remove  $\{I2, I3, I5\}$  from  $C_3$ .
  - The 2-item subsets of  $\{I2, I4, I5\}$  are  $\{I2, I4\}$ ,  $\{I2, I5\}$ , and  $\{I4, I5\}$ .  $\{I4, I5\}$  is not a member of  $L_2$ , and so it is not frequent. Therefore, remove  $\{I2, I4, I5\}$  from  $C_3$ .
- (c) Therefore,  $C_3 = \{\{I1, I2, I3\}, \{I1, I2, I5\}\}$  after pruning.

Figure 5.3: Generation and pruning of candidate 3-itemsets,  $C_3$ , from  $L_2$  using the Apriori property.

Since the rules are generated from frequent itemsets, each one automatically satisfies minimum support. Frequent itemsets can be stored ahead of time in hash tables along with their counts so that they can be accessed quickly.

**Example 5.4 Generating associations rules.** Let's try an example based on the transactional data for *All-Electronics* shown in Table 5.1. Suppose the data contain the frequent itemset  $l = \{I1, I2, I5\}$ . What are the association rules that can be generated from  $l$ ? The nonempty subsets of  $l$  are  $\{I1, I2\}$ ,  $\{I1, I5\}$ ,  $\{I2, I5\}$ ,  $\{I1\}$ ,  $\{I2\}$ , and  $\{I5\}$ . The resulting association rules are as shown below, each listed with its confidence:

$I1 \wedge I2 \Rightarrow I5,$	$confidence = 2/4 = 50\%$
$I1 \wedge I5 \Rightarrow I2,$	$confidence = 2/2 = 100\%$
$I2 \wedge I5 \Rightarrow I1,$	$confidence = 2/2 = 100\%$
$I1 \Rightarrow I2 \wedge I5,$	$confidence = 2/6 = 33\%$
$I2 \Rightarrow I1 \wedge I5,$	$confidence = 2/7 = 29\%$
$I5 \Rightarrow I1 \wedge I2,$	$confidence = 2/2 = 100\%$

If the minimum confidence threshold is, say, 70%, then only the second, third, and last rules above are output, since these are the only ones generated that are strong. Note that, unlike conventional classification rules, association rules can contain more than one conjunct in the right hand side of the rule. ■

### 5.2.3 Improving the Efficiency of Apriori

*“How can we further improve the efficiency of Apriori-based mining?”* Many variations of the Apriori algorithm have been proposed that focus on improving the efficiency of the original algorithm. Several of these variations are summarized below.

**Hash-based technique** (hashing itemsets into corresponding buckets): A hash-based technique can be used to reduce the size of the candidate  $k$ -itemsets,  $C_k$ , for  $k > 1$ . For example, when scanning each transaction in the database to generate the frequent 1-itemsets,  $L_1$ , from the candidate 1-itemsets in  $C_1$ , we can generate all of

**Algorithm: Apriori.** Find frequent itemsets using an iterative level-wise approach based on candidate generation.

**Input:**

- $D$ , a database of transactions;
- $min\_sup$ , the minimum support count threshold.

**Output:**  $L$ , frequent itemsets in  $D$ .

**Method:**

```

(1)   $L_1 = \text{find\_frequent\_1-itemsets}(D)$ ;
(2)  for  $(k = 2; L_{k-1} \neq \phi; k++)$  {
(3)     $C_k = \text{apriori\_gen}(L_{k-1})$ ;
(4)    for each transaction  $t \in D$  { // scan  $D$  for counts
(5)       $C_t = \text{subset}(C_k, t)$ ; // get the subsets of  $t$  that are candidates
(6)      for each candidate  $c \in C_t$ 
(7)         $c.\text{count}++$ ;
(8)    }
(9)     $L_k = \{c \in C_k | c.\text{count} \geq min\_sup\}$ 
(10) }
(11) return  $L = \cup_k L_k$ ;

procedure apriori_gen( $L_{k-1}$ :frequent  $(k-1)$ -itemsets)
(1)  for each itemset  $l_1 \in L_{k-1}$ 
(2)    for each itemset  $l_2 \in L_{k-1}$ 
(3)      if  $(l_1[1] = l_2[1]) \wedge (l_1[2] = l_2[2]) \wedge \dots \wedge (l_1[k-2] = l_2[k-2]) \wedge (l_1[k-1] < l_2[k-1])$  then {
(4)         $c = l_1 \bowtie l_2$ ; // join step: generate candidates
(5)        if  $\text{has\_infrequent\_subset}(c, L_{k-1})$  then
(6)          delete  $c$ ; // prune step: remove unfruitful candidate
(7)        else add  $c$  to  $C_k$ ;
(8)      }
(9)  return  $C_k$ ;

procedure has_infrequent_subset( $c$ : candidate  $k$ -itemset;
                                $L_{k-1}$ : frequent  $(k-1)$ -itemsets); // use prior knowledge
(1)  for each  $(k-1)$ -subset  $s$  of  $c$ 
(2)    if  $s \notin L_{k-1}$  then
(3)      return TRUE;
(4)  return FALSE;

```

Figure 5.4: The Apriori algorithm for discovering frequent itemsets for mining Boolean association rules.

the 2-itemsets for each transaction, hash (i.e., map) them into the different *buckets* of a *hash table* structure, and increase the corresponding bucket counts (Figure 5.5). A 2-itemset whose corresponding bucket count in the hash table is below the support threshold cannot be frequent and thus should be removed from the candidate set. Such a hash-based technique may substantially reduce the number of the candidate  $k$ -itemsets examined (especially when  $k = 2$ ).

**Transaction reduction** (reducing the number of transactions scanned in future iterations): A transaction that does not contain any frequent  $k$ -itemsets cannot contain any frequent  $(k+1)$ -itemsets. Therefore, such a transaction can be marked or removed from further consideration since subsequent scans of the database for  $j$ -itemsets, where  $j > k$ , will not require it.

**Partitioning** (partitioning the data to find candidate itemsets): A partitioning technique can be used that requires just two database scans to mine the frequent itemsets (Figure 5.6). It consists of two phases. In Phase I, the algorithm subdivides the transactions of  $D$  into  $n$  nonoverlapping partitions. If the minimum support threshold for transactions in  $D$  is  $min\_sup$ , then the minimum support count for a partition is  $min\_sup \times \text{the number of transactions in that partition}$ . For each partition, all frequent itemsets within the partition are found. These are referred to as **local frequent itemsets**. The procedure employs a special data structure that, for each itemset, records the TIDs of the transactions containing the items in the itemset. This allows it to find all of the local frequent  $k$ -itemsets, for  $k = 1, 2, \dots$ , in just one scan of the database.

Create hash table  $H_2$  using hash function  $h(x, y) = ((\text{order of } x) \cdot 10 + (\text{order of } y)) \bmod 7$

$H_2$							
bucket address	0	1	2	3	4	5	6
bucket count	2	2	4	2	2	4	4
bucket contents	{I1, I4} {I3, I5}	{I1, I5} {I1, I5}	{I2, I3} {I2, I3} {I2, I3}	{I2, I4} {I2, I4}	{I2, I5} {I2, I5}	{I1, I2} {I1, I2}	{I1, I3} {I1, I3}

Figure 5.5: Hash table,  $H_2$ , for candidate 2-itemsets: This hash table was generated by scanning the transactions of Table 5.1 while determining  $L_1$  from  $C_1$ . If the minimum support count is, say, 3, then the itemsets in buckets 0, 1, 3, and 4 cannot be frequent and so they should not be included in  $C_2$ .

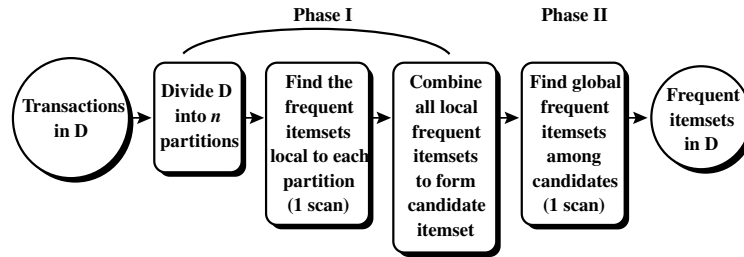


Figure 5.6: Mining by partitioning the data.

A local frequent itemset may or may not be frequent with respect to the entire database,  $D$ . Any itemset that is potentially frequent with respect to  $D$  must occur as a frequent itemset in at least one of the partitions. Therefore, all local frequent itemsets are candidate itemsets with respect to  $D$ . The collection of frequent itemsets from all partitions forms the **global candidate itemsets** with respect to  $D$ . In Phase II, a second scan of  $D$  is conducted in which the actual support of each candidate is assessed in order to determine the global frequent itemsets. Partition size and the number of partitions are set so that each partition can fit into main memory and therefore be read only once in each phase.

**Sampling** (mining on a subset of the given data): The basic idea of the sampling approach is to pick a random sample  $S$  of the given data  $D$ , and then search for frequent itemsets in  $S$  instead of  $D$ . In this way, we trade off some degree of accuracy against efficiency. The sample size of  $S$  is such that the search for frequent itemsets in  $S$  can be done in main memory, and so only one scan of the transactions in  $S$  is required overall. Because we are searching for frequent itemsets in  $S$  rather than in  $D$ , it is possible that we will miss some of the global frequent itemsets. To lessen this possibility, we use a lower support threshold than minimum support to find the frequent itemsets local to  $S$  (denoted  $L^S$ ). The rest of the database is then used to compute the actual frequencies of each itemset in  $L^S$ . A mechanism is used to determine whether all of the global frequent itemsets are included in  $L^S$ . If  $L^S$  actually contains all of the frequent itemsets in  $D$ , then only one scan of  $D$  is required. Otherwise, a second pass can be done in order to find the frequent itemsets that were missed in the first pass. The sampling approach is especially beneficial when efficiency is of utmost importance, such as in computationally intensive applications that must be run on a very frequent basis.

**Dynamic itemset counting** (adding candidate itemsets at different points during a scan): A dynamic itemset counting technique was proposed in which the database is partitioned into blocks marked by start points. In this variation, new candidate itemsets can be added at any start point, unlike in Apriori, which determines new candidate itemsets only immediately prior to each complete database scan. The technique is dynamic in that it estimates the support of all of the itemsets that have been counted so far, adding new candidate itemsets if all of their subsets are estimated to be frequent. The resulting algorithm requires fewer database scans than Apriori.

Other variations involving the mining of multilevel and multidimensional association rules are discussed in the rest of this chapter. The mining of associations related to spatial data, time-series data, and multimedia data are discussed in Chapter 9.

### 5.2.4 Mining Frequent Itemsets without Candidate Generation

As we have seen, in many cases the Apriori candidate generate-and-test method significantly reduces the size of candidate sets, leading to good performance gain. However, it can suffer from two nontrivial costs.

- *It may need to generate a huge number of candidate sets.* For example, if there are  $10^4$  frequent 1-itemsets, the Apriori algorithm will need to generate more than  $10^7$  candidate 2-itemsets. Moreover, to discover a frequent pattern of size 100, such as  $\{a_1, \dots, a_{100}\}$ , it has to generate at least  $2^{100} - 1 \approx 10^{30}$  candidates in total.
- *It may need to repeatedly scan the database and check a large set of candidates by pattern matching.* It is costly to go over each transaction in the database to determine the support of the candidate itemsets.

“Can we design a method that mines the complete set of frequent itemsets without candidate generation?” An interesting method in this attempt is called **frequent-pattern growth**, or simply **FP-growth**, which adopts a *divide-and-conquer* strategy as follows. First, it compresses the database representing frequent items into a **frequent-pattern tree**, or **FP-tree**, which retains the itemset association information. It then divides the compressed database into a set of *conditional databases* (a special kind of projected database), each associated with one frequent item or “pattern fragment”, and mines each such database separately. You’ll see how it works with the following example.

**Example 5.5 FP-growth (finding frequent itemsets without candidate generation).** We reexamine the mining of transaction database,  $D$ , of Table 5.1 in Example 5.3 using the frequent-pattern growth approach.

The first scan of the database is the same as Apriori, which derives the set of frequent items (1-itemsets) and their support counts (frequencies). Let the minimum support count be 2. The set of frequent items is sorted in the order of descending support count. This resulting set or *list* is denoted  $L$ . Thus, we have  $L = \{\{I2: 7\}, \{I1: 6\}, \{I3: 6\}, \{I4: 2\}, \{I5: 2\}\}$ .

An FP-tree is then constructed as follows. First, create the root of the tree, labeled with “null”. Scan database  $D$  a second time. The items in each transaction are processed in  $L$  order (i.e., sorted according to descending support count) and a branch is created for each transaction. For example, the scan of the first transaction, “T100: I1, I2, I5”, which contains three items (I2, I1, I5 in  $L$  order), leads to the construction of the first branch of the tree with three nodes,  $\langle I2: 1 \rangle$ ,  $\langle I1: 1 \rangle$ , and  $\langle I5: 1 \rangle$ , where I2 is linked as a child of the root, I1 is linked to I2, and I5 is linked to I1. The second transaction, T200, contains the items I2 and I4 in  $L$  order, which would result in a branch where I2 is linked to the root and I4 is linked to I2. However, this branch would share a common **prefix**, I2, with the existing path for T100. Therefore, we instead increment the count of the I2 node by 1, and create a new node,  $\langle I4: 1 \rangle$ , which is linked as a child of  $\langle I2: 2 \rangle$ . In general, when considering the branch to be added for a transaction, the count of each node along a common prefix is incremented by 1, and nodes for the items following the prefix are created and linked accordingly.

To facilitate tree traversal, an item header table is built so that each item points to its occurrences in the tree via a chain of **node-links**. The tree obtained after scanning all of the transactions is shown in Figure 5.7 with the associated node-links. In this way, the problem of mining frequent patterns in databases is transformed to that of mining the FP-tree.

The FP-tree is mined as follows. Start from each frequent length-1 pattern (as an initial **suffix pattern**), construct its **conditional pattern base** (a “subdatabase”, which consists of the set of *prefix paths* in the FP-tree co-occurring with the suffix pattern), then construct its (*conditional*) FP-tree, and perform mining recursively on such a tree. The pattern growth is achieved by the concatenation of the suffix pattern with the frequent patterns generated from a conditional FP-tree.

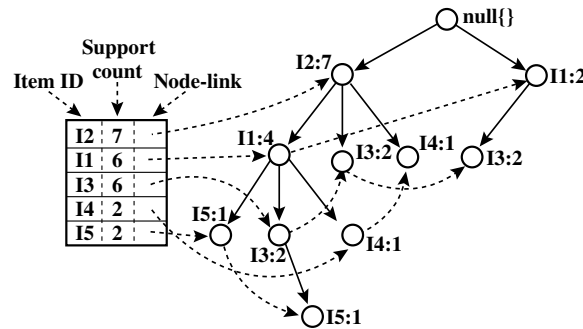


Figure 5.7: An FP-tree registers compressed, frequent pattern information.

Mining of the FP-tree is summarized in Table 5.2 and detailed as follows. We first consider I5 which is the last item in  $L$ , rather than the first. The reason for starting at the end of the list will become apparent as we explain the FP-tree mining process. I5 occurs in two branches of the FP-tree of Figure 5.7. (The occurrences of I5 can easily be found by following its chain of node-links.) The paths formed by these branches are  $\langle I2, I1, I5: 1 \rangle$  and  $\langle I2, I1, I3, I5: 1 \rangle$ . Therefore, considering I5 as a suffix, its corresponding two prefix paths are  $\langle I2, I1: 1 \rangle$  and  $\langle I2, I1, I3: 1 \rangle$ , which form its conditional pattern base. Its conditional FP-tree contains only a single path,  $\langle I2: 2, I1: 2 \rangle$ ; I3 is not included because its support count of 1 is less than the minimum support count. The single path generates all the combinations of frequent patterns:  $\{I2, I5: 2\}$ ,  $\{I1, I5: 2\}$ ,  $\{I2, I1, I5: 2\}$ .

Table 5.2: Mining the FP-tree by creating conditional (sub)-pattern bases.

item	conditional pattern base	conditional FP-tree	frequent patterns generated
I5	$\{\langle I2, I1: 1 \rangle, \langle I2, I1, I3: 1 \rangle\}$	$\langle I2: 2, I1: 2 \rangle$	$\{I2, I5: 2\}, \{I1, I5: 2\}, \{I2, I1, I5: 2\}$
I4	$\{\langle I2, I1: 1 \rangle, \langle I2: 1 \rangle\}$	$\langle I2: 2 \rangle$	$\{I2, I4: 2\}$
I3	$\{\langle I2, I1: 2 \rangle, \langle I2: 2 \rangle, \langle I1: 2 \rangle\}$	$\langle I2: 4, I1: 2 \rangle, \langle I1: 2 \rangle$	$\{I2, I3: 4\}, \{I1, I3: 4\}, \{I2, I1, I3: 2\}$
I1	$\{\langle I2: 4 \rangle\}$	$\langle I2: 4 \rangle$	$\{I2, I1: 4\}$

For I4, its two prefix paths form the conditional pattern base,  $\{\langle I2, I1: 1 \rangle, \langle I2: 1 \rangle\}$ , which generates a single-node conditional FP-tree,  $\langle I2: 2 \rangle$ , and derives one frequent pattern,  $\{I2, I1: 4\}$ . Notice that although I5 follows I4 in the first branch, there is no need to include I5 in the analysis here since any frequent pattern involving I5 is analyzed in the examination of I5.

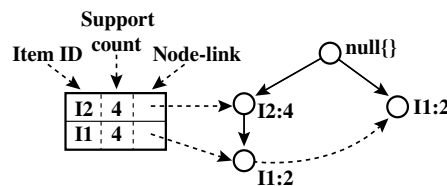


Figure 5.8: The conditional FP-tree associated with the conditional node I3.

Similar to the above analysis, I3's conditional pattern base is  $\{\langle I2, I1: 2 \rangle, \langle I2: 2 \rangle, \langle I1: 2 \rangle\}$ . Its conditional FP-tree has two branches,  $\langle I2: 4, I1: 2 \rangle$  and  $\langle I1: 2 \rangle$ , as shown in Figure 5.8, which generates the set of patterns,  $\{\langle I2, I3: 4 \rangle, \langle I1, I3: 4 \rangle, \langle I2, I1, I3: 2 \rangle\}$ . Finally, I1's conditional pattern base is  $\{\langle I2: 4 \rangle\}$ , whose FP-tree contains only one node,  $\langle I2: 4 \rangle$ , which generates one frequent pattern,  $\{I2, I1: 4\}$ . This mining process is summarized in Figure 5.9. ■

The FP-growth method transforms the problem of finding long frequent patterns to searching for shorter ones recursively and then concatenating the suffix. It uses the least frequent items as a suffix, offering good selectivity.

**Algorithm: FP\_growth.** Mine frequent itemsets using an FP-tree by pattern fragment growth.

**Input:**

- $D$ , a transaction database;
- $min\_sup$ , the minimum support count threshold.

**Output:** The complete set of frequent patterns.

**Method:**

1. The FP-tree is constructed in the following steps.
  - (a) Scan the transaction database  $D$  once. Collect  $F$ , the set of frequent items, and their support counts. Sort  $F$  in support count descending order as  $L$ , the *list* of frequent items.
  - (b) Create the root of an FP-tree, and label it as “null”. For each transaction  $Trans$  in  $D$  do the following.  
 Select and sort the frequent items in  $Trans$  according to the order of  $L$ . Let the sorted frequent item list in  $Trans$  be  $[p|P]$ , where  $p$  is the first element and  $P$  is the remaining list. Call `insert_tree([p|P], T)`, which is performed as follows. If  $T$  has a child  $N$  such that  $N.item\_name = p.item\_name$ , then increment  $N$ ’s count by 1; else create a new node  $N$ , and let its count be 1, its parent link be linked to  $T$ , and its node-link to the nodes with the same *item-name* via the node-link structure. If  $P$  is nonempty, call `insert_tree(P, N)` recursively.
2. The FP-tree is mined by calling `FP_growth(FP_tree, null)`, which is implemented as follows.

```

procedure FP_growth(Tree,  $\alpha$ )
(1)  if Tree contains a single path  $P$  then
(2)    for each combination (denoted as  $\beta$ ) of the nodes in the path  $P$ 
(3)      generate pattern  $\beta \cup \alpha$  with support_count = minimum support count of nodes in  $\beta$ ;
(4)  else for each  $a_i$  in the header of Tree {
(5)    generate pattern  $\beta = a_i \cup \alpha$  with support_count =  $a_i.support\_count$ ;
(6)    construct  $\beta$ ’s conditional pattern base and then  $\beta$ ’s conditional FP-tree  $Tree_\beta$ ;
(7)    if  $Tree_\beta \neq \emptyset$  then
(8)      call FP_growth(Tree $_\beta$ ,  $\beta$ ); }

```

Figure 5.9: The FP-growth algorithm for discovering frequent itemsets without candidate generation.

The method substantially reduces the search costs.

When the database is large, it is sometimes unrealistic to construct a main memory-based FP-tree. An interesting alternative is to first partition the database into a set of projected databases, and then construct an FP-tree and mine it in each projected database. Such a process can be recursively applied to any projected database if its FP-tree still cannot fit in main memory.

A study on the performance of the FP-growth method shows that it is efficient and scalable for mining both long and short frequent patterns, and is about an order of magnitude faster than the Apriori algorithm. It is also faster than a Tree-Projection algorithm, which recursively projects a database into a tree of projected databases.

### 5.2.5 Mining Frequent Itemsets Using Vertical Data Format

Both the Apriori and FP-growth methods mine frequent patterns from a set of transactions in *TID-itemset* format (that is,  $\{TID : itemset\}$ ), where  $TID$  is a transaction-id and *itemset* is the set of items bought in transaction  $TID$ . This data format is known as **horizontal data format**. Alternatively, data can also be presented in *item-TID\_set* format (that is,  $\{item : TID\_set\}$ ), where *item* is an item name, and  $TID\_set$  is the set of transaction identifiers containing the item. This format is known as **vertical data format**.

In this section, we look at how frequent itemsets can also be mined efficiently using vertical data format, which is the essence of the Eclat algorithm developed by Zaki [Zak00].

**Example 5.6 Mining frequent itemsets using vertical data format.** Consider the horizontal data format of the transaction database,  $D$ , of Table 5.1 in Example 5.3. This can be transformed into the vertical data format shown in Table 5.3 by scanning the data set once.

Table 5.3: The vertical data format of the transaction data set  $D$  of Table 5.1.

<i>itemset</i>	<i>TID_set</i>
I1	{T100, T400, T500, T700, T800, T900}
I2	{T100, T200, T300, T400, T600, T800, T900}
I3	{T300, T500, T600, T700, T800, T900}
I4	{T200, T400}
I5	{T100, T800}

Mining can be performed on this data set by intersecting the *TID\_set*s of every pair of frequent single-items. The minimum support count is 2. Since every single item is frequent in Table 5.3, there are 10 intersections performed in total, which lead to 8 nonempty 2-itemsets as shown in Table 5.4. Notice that since the itemsets {I1, I4} and {I3, I5} each contain only one transaction, they do not belong to the set of frequent 2-itemsets.

Table 5.4: The 2-itemsets in vertical data format.

<i>itemset</i>	<i>TID_set</i>
{I1,I2}	{T100, T400, T800, T900}
{I1,I3}	{T500, T700, T800, T900}
{I1,I4}	{T400}
{I1,I5}	{T100, T800}
{I2,I3}	{T300, T600, T800, T900}
{I2,I4}	{T200, T400}
{I2,I5}	{T100, T800}
{I3,I5}	{T800}

Table 5.5: The 3-itemsets in vertical data format.

<i>itemset</i>	<i>TID_set</i>
{I1,I2,I3}	{T800, T900}
{I1,I2,I5}	{T100, T800}

Based on the Apriori property, a given 3-itemset is a candidate 3-itemset only if every one of its 2-itemset subsets is frequent. The candidate generation process here will generate only two 3-itemsets: {I1, I2, I3} and {I1, I2, I5}. By intersecting the *TID\_set*s of any two corresponding 2-itemsets of these candidate 3-itemsets, it derives Table 5.5, where there are only two frequent 3-itemsets: {I1, I2, I3: 2} and {I1, I2, I5: 2}. ■

Example 5.6 illustrates the process of mining frequent itemsets by exploring the vertical data format. First, we transform the horizontally formatted data to the vertical format by scanning the data set once. The support count of an itemset is simply the length of the *TID\_set* of the itemset. Starting with  $k = 1$ , the frequent  $k$ -itemsets can be used to construct the candidate  $(k + 1)$ -itemsets based on the Apriori property. The computation is done by intersection of the *TID\_set*s of the frequent  $k$ -itemsets to compute the *TID\_set*s of the corresponding  $(k + 1)$ -itemsets. This process repeats, with  $k$  incremented by 1 each time, until no frequent itemsets or no candidate itemsets can be found.

Besides taking advantage of the Apriori property in the generation of candidate  $(k + 1)$ -itemset from frequent  $k$ -itemsets, another merit of this method is that there is no need to scan the database to find the support of  $(k + 1)$  itemsets (for  $k \geq 1$ ). This is because the *TID\_set* of each  $k$ -itemset carries the complete information required for counting such support. However, the *TID\_set*s can be quite long, taking substantial memory space as well as computation time for intersecting the long sets.

To further reduce the cost of registering long *TID\_set*s as well as the subsequent costs of intersections, we can use a technique called *diffset*, which keeps track of only the differences of the *TID\_set*s of a  $(k + 1)$ -itemset and a corresponding  $k$ -itemset. For instance, in Example 5.6 we have {I1} = {T100, T400, T500, T700, T800,



T900}, and  $\{I1, I2\} = \{T100, T400, T800, T900\}$ . The *diffset* between the two, that is,  $\text{diffset}(\{I1, I2\}, \{I1\}) = \{T500, T700\}$ . Thus, rather than recording the four TIDs that make up the intersection of  $\{I1\}$  and  $\{I2\}$ , we can instead use *diffset* to record just two TIDs indicating the difference between  $\{I1\}$  and  $\{I1, I2\}$ . Experiments show that in certain situations, such as when the data set contains many dense and long patterns, this technique can substantially reduce the total cost of vertical format mining of frequent itemsets.

### 5.2.6 Mining Closed Frequent Itemsets

In Section 5.1.2 we saw how frequent itemset mining may generate a huge number of frequent itemsets, especially when the *min\_sup* threshold is set low or when there exist long patterns in the data set. Example 5.2 showed that closed frequent itemsets<sup>9</sup> can substantially reduce the number of patterns generated in frequent itemset mining while preserving the complete information regarding the set of frequent itemsets. That is, from the set of closed frequent itemsets, we can easily derive the set of frequent itemsets and their support. Thus in practice, it is more desirable to mine the set of closed frequent itemsets rather than the set of all frequent itemsets in most cases.

*“How can we mine closed frequent itemsets?”* A naïve approach would be to first mine the complete set of frequent itemsets and then remove every frequent itemset that is a proper subset of, and carries the same support as, an existing frequent itemset. However, this is quite costly. As shown in Example 5.2, this method would have to first derive  $2^{100} - 1$  frequent itemsets in order to obtain a length-100 frequent itemset, all before it could begin to eliminate redundant itemsets. This is prohibitively expensive. In fact, there exist only a very small number of closed frequent itemsets in the data set of Example 5.2.

A recommended methodology is to search for closed frequent itemsets directly during the mining process. This requires us to prune the search space as soon as we can identify the case of closed itemsets during mining. Pruning strategies include the following:

**Item merging:** *If every transaction containing a frequent itemset  $X$  also contains an itemset  $Y$  but not any proper superset of  $Y$ , then  $X \cup Y$  forms a frequent closed itemset and there is no need to search for any itemset containing  $X$  but no  $Y$ .*

For example, in Table 5.2 of Example 5.5, the projected conditional database for prefix itemset  $\{I5:2\}$  is  $\{\{I2, I1\}, \{I2, I1, I3\}\}$  from which we can see that each of its transactions contains itemset  $\{I2, I1\}$  but no proper superset of  $\{I2, I1\}$ . Itemset  $\{I2, I1\}$  can be merged with  $\{I5\}$  to form the closed itemset,  $\{I5, I2, I1: 2\}$ , and we do not need to mine for closed itemsets that contain  $I5$  but not  $\{I2, I1\}$ .

**Sub-itemset pruning:** *If a frequent itemset  $X$  is a proper subset of an already found frequent closed itemset  $Y$  and  $\text{support\_count}(X) = \text{support\_count}(Y)$ , then  $X$  and all of  $X$ 's descendants in the set enumeration tree cannot be frequent closed itemsets and thus can be pruned.*

Similar to Example 5.2, suppose a transaction database has only two transactions:  $\{\langle a_1, a_2, \dots, a_{100} \rangle, \langle a_1, a_2, \dots, a_{50} \rangle\}$ , and the minimum support count is *min\_sup* = 2. The projection on the first item,  $a_1$ , derives the frequent itemset,  $\{a_1, a_2, \dots, a_{50} : 2\}$ , based on the *itemset merging* optimization. Since  $\text{support}(\{a_2\}) = \text{support}(\{a_1, a_2, \dots, a_{50}\}) = 2$ , and  $\{a_2\}$  is a proper subset of  $\{a_1, a_2, \dots, a_{50}\}$ , there is no need to examine  $a_2$  and its projected database. Similar pruning can be done for  $a_3, \dots, a_{50}$  as well. Thus the mining of closed frequent itemsets in this data set terminates after mining  $a_1$ 's projected database.

**Item skipping:** *In the depth-first mining of closed itemsets, at each level, there will be a prefix itemset  $X$  associated with a header table and a projected database. If a local frequent item  $p$  has the same support in several header tables at different levels, we can safely prune  $p$  from the header tables at higher levels.*

Consider, for example, the transaction database above having only two transactions:  $\{\langle a_1, a_2, \dots, a_{100} \rangle, \langle a_1, a_2, \dots, a_{50} \rangle\}$ , where *min\_sup* = 2. Since  $a_2$  in  $a_1$ 's projected database has the same support as  $a_2$  in the global header table,  $a_2$  can be pruned from the global header table. Similar pruning can be done for  $a_3, \dots, a_{50}$ . There is no need to mine anything more after mining  $a_1$ 's projected database.

<sup>9</sup>Remember that  $X$  is a *closed frequent itemset* in a data set  $S$  if there exists no proper super-itemset  $Y$  such that  $Y$  has the same support count as  $X$  in  $S$ , and  $X$  satisfies minimum support.

Besides pruning the search space in the closed itemset mining process, another important optimization is to perform efficient checking of a newly derived frequent itemset to see whether it is closed since the mining process itself cannot ensure that every generated frequent itemset is closed.

When a new frequent itemset is derived, it is necessary to perform two kinds of closure checking: (1) *superset checking*, which checks if this new frequent itemset is a superset of some already found closed itemsets with the same support, and (2) *subset checking*, which checks if the newly found itemset is a subset of an already found closed itemset with the same support.

If we adopt the *item merging* pruning method under a divide-and-conquer framework, then the superset checking is actually built-in and there is no need to explicitly perform superset checking. This is because if a frequent itemset  $X \cup Y$  is found later than itemset  $X$ , and carries the same support as  $X$ , it must be in  $X$ 's projected database and must have been generated during itemset merging.

To assist in subset checking, a compressed **pattern-tree** can be constructed to maintain the set of closed itemsets mined so far. The pattern-tree is similar in structure to the FP-tree except that all of the closed itemsets found are stored explicitly in the corresponding tree branches. For efficient subset checking, we can use the following property: *If the current itemset  $S_c$  can be subsumed by another already found closed itemset  $S_a$ , then (1)  $S_c$  and  $S_a$  have the same support, (2) the length of  $S_c$  is smaller than that of  $S_a$ , and (3) all of the items in  $S_c$  are contained in  $S_a$ .* Based on this property, a **two-level hash index structure** can be built for fast accessing of the pattern-tree: The first level uses the identifier of the last item in  $S_c$  as a hash key (since this identifier must be within the branch of  $S_c$ ), and the second level uses the support of  $S_c$  as a hash key (since  $S_c$  and  $S_a$  have the same support). This will substantially speed up the subset checking process.

The above discussion illustrates methods for efficient mining of closed frequent itemsets. “Can we extend these methods for efficient mining of maximal frequent itemsets?” Since maximal frequent itemsets share many similarities with closed frequent itemsets, many of the optimization techniques developed here can be extended to mining maximal frequent itemsets. However, we leave this as an exercise for interested readers.

## 5.3 Mining Various Kinds of Association Rules

We have studied efficient methods for mining frequent itemsets and association rules. In this section, we consider additional application requirements by extending our scope to include mining multilevel association rules, multidimensional association rules, and quantitative association rules in transactional and/or relational databases and data warehouses. *Multilevel association rules* involve concepts at different levels of abstraction. *Multidimensional association rules* involve more than one dimension or predicate (e.g., rules relating what a customer *buys* as well as the customer's *age*.) *Quantitative association rules* involve numeric attributes that have an implicit ordering among values (e.g., *age*).

### 5.3.1 Mining Multilevel Association Rules

For many applications, it is difficult to find strong associations among data items at low or primitive levels of abstraction due to the sparsity of data at those levels. Strong associations discovered at high levels of abstraction may represent common sense knowledge. Moreover, what may represent common sense to one user may seem novel to another. Therefore, data mining systems should provide capabilities for mining association rules at multiple levels of abstraction, with sufficient flexibility for easy traversal among different abstraction spaces.

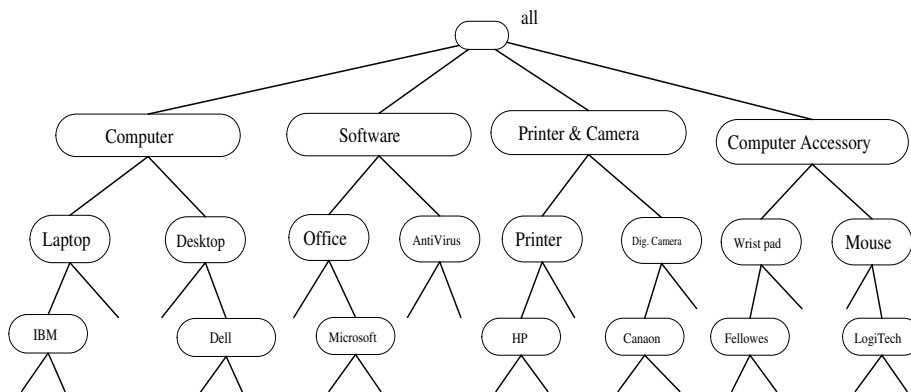
Let's examine the following example.

**Example 5.7 Mining multilevel association rules.** Suppose we are given the task-relevant set of transactional data in Table 5.6 for sales in an *AllElectronics* store, showing the items purchased for each transaction. The concept hierarchy for the items is shown in Figure 5.10. A concept hierarchy defines a sequence of mappings from a set of low-level concepts to higher-level, more general concepts. Data can be generalized by replacing low-level concepts within the data by their higher-level concepts, or *ancestors*, from a concept hierarchy. The concept hierarchy of

Figure 5.10 has five levels, respectively referred to as levels 0 to 4, starting with level 0 at the root node for *all* (the most general abstraction level). Here, level 1 includes *computer*, *software*, *printer&camera*, and *computer accessory*, level 2 includes *laptop computer*, *desktop computer*, *office software*, *antivirus software*, ..., and level 3 includes *IBM desktop computer*, ..., *Microsoft office software*, and so on. Level 4 represents the most specific abstraction level of this hierarchy. Concept hierarchies for categorical attributes are often implicit within the database schema, in which case they may be automatically generated using methods such as those described in Chapter 2. For our example, the concept hierarchy of Figure 5.10 was generated from data on product specifications. Concept hierarchies for numerical attributes can be generated using discretization techniques, many of which were introduced in Chapter 2. Alternatively, concept hierarchies may be specified by users familiar with the data, such as store managers in the case of our example.

Table 5.6: Task-relevant data, *D*.

<i>TID</i>	<i>items purchased</i>
T100	IBM-ThinkPad-T40/2373, HP-Photosmart-7660
T200	Microsoft-Office-Professional-2003, Microsoft-Plus!-Digital-Media
T300	Logitech-MX700-Cordless-Mouse, Fellowes-Wrist-Rest
T400	Dell-Dimension-XPS, Canon-PowerShot-S400
T500	IBM-ThinkPad-R40/P4M, Symantec-Norton-Antivirus-2003
...	...

Figure 5.10: A concept hierarchy for *AllElectronics* computer items.

The items in Table 5.6 are at the lowest level of the concept hierarchy of Figure 5.10. It is difficult to find interesting purchase patterns at such raw or primitive-level data. For instance, if “*IBM-ThinkPad-R40/P4M*” or “*Symantec-Norton-Antivirus-2003*” each occurs in a very small fraction of the transactions, then it can be difficult to find strong associations involving these specific items. Few people may buy these items together, making it unlikely that the itemset will satisfy minimum support. However, we would expect that it is easier to find strong associations between generalized abstractions of these items, such as between “*IBM laptop computer*” and “*antivirus software*”. ■

Association rules generated from mining data at multiple levels of abstraction are called **multiple-level** or **multilevel association rules**. Multilevel association rules can be mined efficiently using concept hierarchies under a support-confidence framework. In general, a top-down strategy is employed, where counts are accumulated for the calculation of frequent itemsets at each concept level, starting at the concept level 1 and working downwards in the hierarchy, towards the more specific concept levels, until no more frequent itemsets can be found. For each level, any algorithm for discovering frequent itemsets may be used, such as Apriori or its variations. A number of

variations to this approach are described below, where each variation involves “playing” with the support threshold in a slightly different way. The variations are illustrated in Figures 5.11 and 5.12, where nodes indicate an item or itemset that has been examined, and nodes with thick borders indicate that an examined item or itemset is frequent.

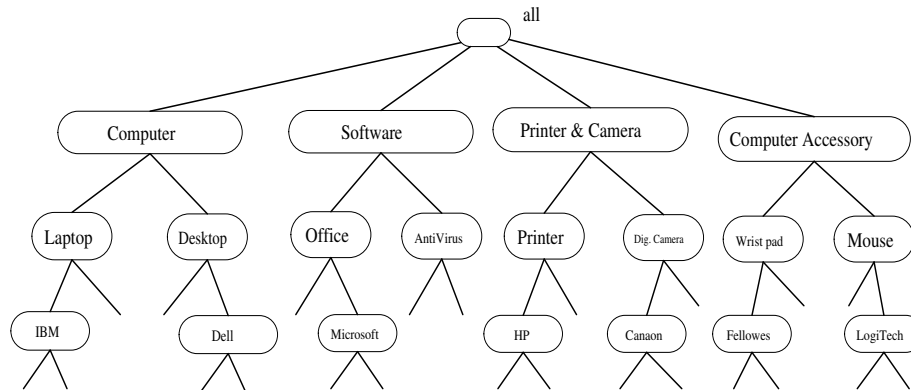


Figure 5.11: Multilevel mining with uniform support.

- **Using uniform minimum support for all levels** (referred to as **uniform support**): The same minimum support threshold is used when mining at each level of abstraction. For example, in Figure 5.11, a minimum support threshold of 5% is used throughout (e.g., for mining from “computer” down to “laptop computer”). Both “computer” and “laptop computer” are found to be frequent, while “desktop computer” is not.

When a uniform minimum support threshold is used, the search procedure is simplified. The method is also simple in that users are required to specify only one minimum support threshold. An Apriori-like optimization technique can be adopted, based on the knowledge that an ancestor is a superset of its descendants: The search avoids examining itemsets containing any item whose ancestors do not have minimum support.

The uniform support approach, however, has some difficulties. It is unlikely that items at lower levels of abstraction will occur as frequently as those at higher levels of abstraction. If the minimum support threshold is set too high, it could miss some meaningful associations occurring at low abstraction levels. If the threshold is set too low, it may generate many uninteresting associations occurring at high abstraction levels. This provides the motivation for the following approach.

- **Using reduced minimum support at lower levels** (referred to as **reduced support**): Each level of abstraction has its own minimum support threshold. The deeper the level of abstraction, the smaller the corresponding threshold is. For example, in Figure 5.12, the minimum support thresholds for levels 1 and 2 are 5% and 3%, respectively. In this way, “computer”, “laptop computer”, and “desktop computer” are all considered frequent.
- **Using item or group-based minimum support** (referred to as **group-based support**): Since users or experts often have insight as to which groups are more important than others, it is sometimes more desirable to set up user-specific, item or group-based minimal support thresholds when mining multilevel rules. For example, a user could set up the minimum support thresholds based on product price, or on items of interest, such as by setting particularly low support thresholds for *laptop computers* and *flash drives* in order to pay particular attention to the association patterns containing items in these categories.

Notice that the Apriori property may not always hold uniformly across all the items when mining under reduced support and group-based support. However, efficient methods can be developed based on the extension of the property. The details are left as an exercise for interested readers.

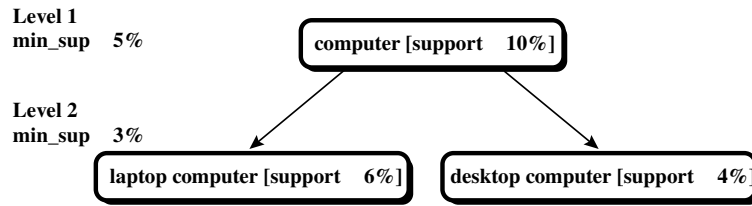


Figure 5.12: Multilevel mining with reduced support.

A serious side-effect of mining multilevel association rules is its generation of many redundant rules across multiple levels of abstraction due to the “ancestor” relationships among items. For example, consider the following rules where “*laptop computer*” is an ancestor of “*IBM laptop computer*” based on the concept hierarchy of Figure 5.10, and where  $X$  is a variable representing customers who purchased items in *AllElectronics* transactions.

$$\text{buys}(X, \text{“laptop computer”}) \Rightarrow \text{buys}(X, \text{“HP printer”}) \quad [\text{support} = 8\%, \text{confidence} = 70\%] \quad (5.10)$$

$$\text{buys}(X, \text{“IBM laptop computer”}) \Rightarrow \text{buys}(X, \text{“HP printer”}) \quad [\text{support} = 2\%, \text{confidence} = 72\%] \quad (5.11)$$

“If Rules (5.10) and (5.11) are both mined, then how useful is the latter rule?” you may wonder. “Does it really provide any novel information?” If the latter, less general rule does not provide new information, it should be removed. Let’s have a look at how this may be determined. A rule  $R1$  is an **ancestor** of a rule  $R2$ , if  $R1$  can be obtained by replacing the items in  $R2$  by their ancestors in a concept hierarchy. For example, Rule (5.10) is an ancestor of Rule (5.11) since “*laptop computer*” is an ancestor of “*IBM laptop computer*”. Based on this definition, a rule can be considered redundant if its support and confidence are close to their “expected” values, based on an ancestor of the rule. As an illustration, suppose that Rule (5.10) has a 70% confidence and 8% support, and that about one quarter of all “*laptop computer*” sales are for “*IBM laptop computers*”. We may expect Rule (5.11) to have a confidence of around 70% (since all data samples of “*IBM laptop computer*” are also samples of “*laptop computer*”) and a support of around 2% (i.e.,  $8\% \times \frac{1}{4}$ ). If this is indeed the case, then Rule (5.11) is not interesting since it does not offer any additional information and is less general than Rule (5.10).

### 5.3.2 Mining Multidimensional Association Rules from Relational Databases and Data Warehouses

So far in this chapter, we have studied association rules that imply a single predicate, that is, the predicate *buys*. For instance, in mining our *AllElectronics* database, we may discover the Boolean association rule

$$\text{buys}(X, \text{“digital camera”}) \Rightarrow \text{buys}(X, \text{“HP printer”}) \quad (5.12)$$

Following the terminology used in multidimensional databases, we refer to each distinct predicate in a rule as a dimension. Hence, we can refer to Rule (5.12) as a **single-dimensional** or **intradimensional association rule** since it contains a single distinct predicate (e.g., *buys*) with multiple occurrences (i.e., the predicate occurs more than once within the rule). As we have seen in the previous sections of this chapter, such rules are commonly mined from transactional data.

Suppose, however, that rather than using a transactional database, sales and related information are stored in a relational database or data warehouse. Such data stores are multidimensional, by definition. For instance,

in addition to keeping track of the items purchased in sales transactions, a relational database may record other attributes associated with the items, such as the quantity purchased or the price, or the branch location of the sale. Additional relational information regarding the customers who purchased the items, such as customer age, occupation, credit rating, income, and address, may also be stored. Considering each database attribute or warehouse dimension as a predicate, we can therefore mine association rules containing *multiple* predicates, such as

$$age(X, "20...29") \wedge occupation(X, "student") \Rightarrow buys(X, "laptop") \quad (5.13)$$

Association rules that involve two or more dimensions or predicates can be referred to as **multidimensional association rules**. Rule (5.13) contains three predicates (*age*, *occupation*, and *buys*), each of which occurs *only once* in the rule. Hence, we say that it has **no repeated predicates**. Multidimensional association rules with no repeated predicates are called **interdimensional association rules**. We can also mine multidimensional association rules with repeated predicates, which contain multiple occurrences of some predicates. These rules are called **hybrid-dimensional association rules**. An example of such a rule is the following, where the predicate *buys* is repeated:

$$age(X, "20...29") \wedge buys(X, "laptop") \Rightarrow buys(X, "HP printer") \quad (5.14)$$

Note that database attributes can be categorical or quantitative. **Categorical** attributes have a finite number of possible values, with no ordering among the values (e.g., *occupation*, *brand*, *color*). Categorical attributes are also called **nominal** attributes, since their values are “names of things.” **Quantitative** attributes are numeric and have an implicit ordering among values (e.g., *age*, *income*, *price*). Techniques for mining multidimensional association rules can be categorized into two basic approaches regarding the treatment of quantitative attributes.

In the first approach, *quantitative attributes are discretized using predefined concept hierarchies*. This discretization occurs prior to mining. For instance, a concept hierarchy for *income* may be used to replace the original numeric values of this attribute by interval labels, such as “0...20K”, “21K...30K”, “31K...40K”, and so on. Here, discretization is *static* and predetermined. Chapter 2 on data preprocessing gave several techniques for discretizing numeric attributes. The discretized numeric attributes, with their interval labels, can then be treated as categorical attributes (where each interval is considered a category). We refer to this as **mining multidimensional association rules using static discretization of quantitative attributes**.

In the second approach, *quantitative attributes are discretized or clustered into “bins” based on the distribution of the data*. These bins may be further combined during the mining process. The discretization process is *dynamic* and established so as to satisfy some mining criteria, such as maximizing the confidence of the rules mined. Because this strategy treats the numeric attribute values as quantities rather than as predefined ranges or categories, association rules mined from this approach are also referred to as **(dynamic) quantitative association rules**.

Let’s study each of these approaches for mining multidimensional association rules. For simplicity, we confine our discussion to interdimensional association rules. Note that rather than searching for frequent itemsets (as is done for single-dimensional association rule mining), in multidimensional association rule mining we search for frequent *predicate sets*. A **k-predicate set** is a set containing *k* conjunctive predicates. For instance, the set of predicates {*age*, *occupation*, *buys*} from Rule (5.13) is a 3-predicate set. Similar to the notation used for itemsets, we use the notation  $L_k$  to refer to the set of frequent *k*-predicate sets.

## 1. Mining Multidimensional Association Rules Using Static Discretization of Quantitative Attributes

Quantitative attributes, in this case, are discretized prior to mining using predefined concept hierarchies or data discretization techniques, where numeric values are replaced by interval labels. Categorical attributes may also

be generalized to higher conceptual levels if desired. If the resulting task-relevant data are stored in a relational table, then any of the frequent itemset mining algorithms we have discussed can easily be modified so as to find all frequent predicate sets rather than frequent itemsets. In particular, instead of searching on only one attribute like *buys*, we need to search through all of the relevant attributes, treating each attribute-value pair as an itemset.

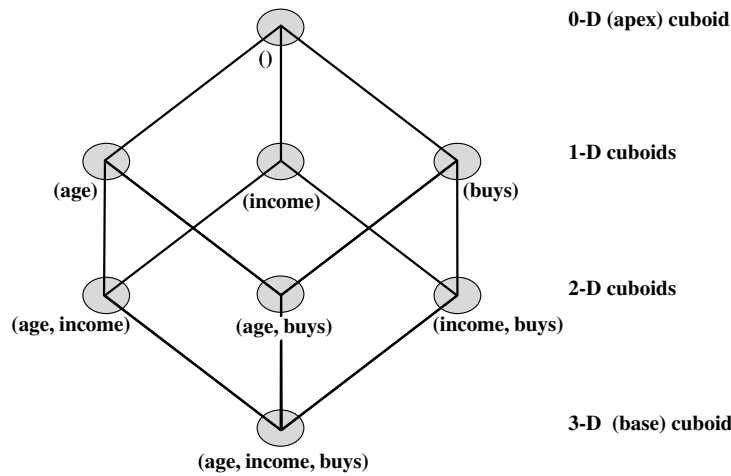


Figure 5.13: Lattice of cuboids, making up a 3-D data cube. Each cuboid represents a different group-by. The base cuboid contains the three predicates *age*, *income*, and *buys*. [TO EDITOR For consistency with rest of book, please kindly italicize all instances of *age*, *income*, and *buys*.]

Alternatively, the transformed multidimensional data may be used to construct a *data cube*. Data cubes are well suited for the mining of multidimensional association rules: They store aggregates (such as counts), in multidimensional space, which is essential for computing the support and confidence of multidimensional association rules. An overview of data cube technology was presented in Chapter 3. Detailed algorithms for data cube computation were given in Chapter 4. Figure 5.13 shows the lattice of cuboids defining a data cube for the dimensions *age*, *income*, and *buys*. The cells of an  $n$ -dimensional cuboid can be used to store the support counts of the corresponding  $n$ -predicate sets. The base cuboid aggregates the task-relevant data by *age*, *income*, and *buys*; the 2-D cuboid, *(age, income)*, aggregates by *age* and *income*; the 0-D (apex) cuboid contains the total number of transactions in the task-relevant data, and so on.

Due to the ever-increasing use of data warehouse and OLAP technology, it is possible that a data cube containing the dimensions that are of interest to the user may already exist, fully materialized. If this is the case, we can simply fetch the corresponding aggregate values and return the rules needed using a rule generation algorithm (Section 5.2.2). Notice that even in this case, the Apriori property can still be used to prune the search space. If a given  $k$ -predicate set has support *sup* which does not satisfy minimum support, then further exploration of this set should be terminated. This is because any more specialized version of the  $k$ -itemset will have support no greater than *sup* and, therefore, will not satisfy minimum support either. In cases where no relevant data cube exists for the mining task, we must create one on the fly. This becomes an iceberg cube computation problem, where the minimum support threshold is taken as the iceberg condition (Chapter 4).

## 2. Mining Quantitative Association Rules

Quantitative association rules are multidimensional association rules in which the numeric attributes are *dynamically* discretized during the mining process so as to satisfy some mining criteria, such as maximizing the confidence or compactness of the rules mined. In this section, we will focus specifically on how to mine quantitative association rules having two quantitative attributes on the left-hand side of the rule, and one categorical attribute on the right-hand side of the rule. That is,

$$A_{quan1} \wedge A_{quan2} \Rightarrow A_{cat}$$

where  $A_{quan1}$  and  $A_{quan2}$  are tests on quantitative attribute intervals (where the intervals are dynamically determined), and  $A_{cat}$  tests a categorical attribute from the task-relevant data. Such rules have been referred to as **two-dimensional quantitative association rules**, since they contain two quantitative dimensions. For instance, suppose you are curious about the association relationship between pairs of quantitative attributes, like customer age and income, and the type of television (such as *high definition TV*, i.e., *HDTV*) that customers like to buy. An example of such a 2-D quantitative association rule is

$$age(X, "30...39") \wedge income(X, "42K...48K") \Rightarrow buys(X, "HDTV") \quad (5.15)$$

*“How can we find such rules?”* Let’s look at an approach used in a system called **ARCS** (Association Rule Clustering System), which borrows ideas from image processing. Essentially, this approach maps pairs of quantitative attributes onto a 2-D grid for tuples satisfying a given categorical attribute condition. The grid is then searched for clusters of points, from which the association rules are generated. The following steps are involved in ARCS:

**Binning:** Quantitative attributes can have a very wide range of values defining their domain. Just think about how big a 2-D grid would be if we plotted *age* and *income* as axes, where each possible value of *age* was assigned a unique position on one axis, and similarly, each possible value of *income* was assigned a unique position on the other axis! To keep grids down to a manageable size, we instead partition the ranges of quantitative attributes into intervals. These intervals are dynamic in that they may later be further combined during the mining process. The partitioning process is referred to as **binning**, that is, where the intervals are considered “bins.” Three common binning strategies are

- **equal-width binning**, where the interval size of each bin is the same,
- **equal-frequency binning**, where each bin has approximately the same number of tuples assigned to it, and
- **clustering-based binning**, where clustering is performed on the quantitative attribute to group *neighboring points* (judged based on various distance measures) into the same bin.

ARCS uses equal-width binning, where the bin size for each quantitative attribute is input by the user. A 2-D array for each possible bin combination involving both quantitative attributes is created. Each array cell holds the corresponding count distribution for each possible class of the categorical attribute of the rule right-hand side. By creating this data structure, the task-relevant data need only be scanned once. The same 2-D array can be used to generate rules for any value of the categorical attribute, based on the same two quantitative attributes. Binning is also discussed in Chapter 2.

**Finding frequent predicate sets:** Once the 2-D array containing the count distribution for each category is set up, this can be scanned in order to find the frequent predicate sets (those satisfying minimum support) that also satisfy minimum confidence. Strong association rules can then be generated from these predicate sets, using a rule generation algorithm like that described in Section 5.2.2.

**Clustering the association rules:** The strong association rules obtained in the previous step are then mapped to a 2-D grid. Figure 5.14 shows a 2-D grid for 2-D quantitative association rules predicting the condition  $buys(X, "HDTV")$  on the rule right-hand side, given the quantitative attributes *age* and *income*. The four Xs correspond to the rules

$$age(X, 34) \wedge income(X, "31K...40K") \Rightarrow buys(X, "HDTV") \quad (5.16)$$

$$age(X, 35) \wedge income(X, "31K...40K") \Rightarrow buys(X, "HDTV") \quad (5.17)$$

$$age(X, 34) \wedge income(X, "41K...50K") \Rightarrow buys(X, "HDTV") \quad (5.18)$$

$$age(X, 35) \wedge income(X, "41K...50K") \Rightarrow buys(X, "HDTV") \quad (5.19)$$



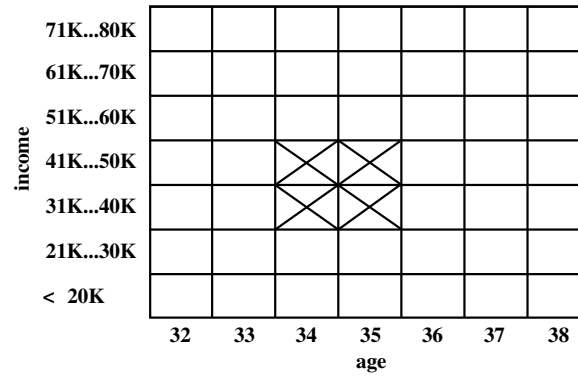


Figure 5.14: A 2-D grid for tuples representing customers who purchase high-definition TVs.

“Can we find a simpler rule to replace the above four rules?” Notice that these rules are quite “close” to one another, forming a rule cluster on the grid. Indeed, the four rules can be combined or “clustered” together to form the following simpler rule, which subsumes and replaces the above four rules:

$$age(X, "34...35") \wedge income(X, "31K...50K") \Rightarrow buys(X, "HDTV") \quad (5.20)$$

ARCS employs a clustering algorithm for this purpose. The algorithm scans the grid, searching for rectangular clusters of rules. In this way, bins of the quantitative attributes occurring within a rule cluster may be further combined, and hence, further dynamic discretization of the quantitative attributes occurs.

The grid-based technique described here assumes that the initial association rules can be clustered into rectangular regions. Prior to performing the clustering, smoothing techniques can be used to help remove noise and outliers from the data. Rectangular clusters may oversimplify the data. Alternative approaches have been proposed, based on other shapes of regions that tend to better fit the data, yet require greater computation effort.

A non-grid-based technique has been proposed to find quantitative association rules that are more general, where any number of quantitative and categorical attributes can appear on either side of the rules. In this technique, quantitative attributes are dynamically partitioned using equal-frequency binning, and the partitions are combined based on a measure of *partial completeness*, which quantifies the information lost due to partitioning. For references on these alternatives to ARCS, see the bibliographic notes.

## 5.4 From Association Mining to Correlation Analysis

Most association rule mining algorithms employ a support-confidence framework. Often, many interesting rules can be found using low support thresholds. Although minimum support and confidence thresholds *help* weed out or exclude the exploration of a good number of uninteresting rules, many rules so generated are still not interesting to the users. Unfortunately, this is especially true *when mining at low support thresholds or mining for long patterns*. This has been one of the major bottlenecks for successful application of association rule mining.

In this section, we first look at how even strong association rules can be uninteresting and misleading. We then discuss how the support-confidence framework can be supplemented with additional interestingness measures based on statistical significance and correlation analysis.

### 5.4.1 Strong Rules Are Not Necessarily Interesting: An Example

Whether or not a rule is interesting can be assessed either subjectively or objectively. Ultimately, only the user can judge if a given rule is interesting, and this judgment, being subjective, may differ from one user to another.



“lifts” the occurrence of the other. For example, if  $A$  corresponds to the sale of computer games and  $B$  corresponds to the sale of videos, then given the current market conditions, the sale of games is said to increase or “lift” the likelihood of the sale of videos by a factor of the value returned by Equation (5.23).

Let’s go back to the computer game and video data of Example 5.8.

Table 5.7: A  $2 \times 2$  contingency table summarizing the transactions with respect to game and video purchases.

	<i>game</i>	$\overline{game}$	$\Sigma_{row}$
<i>video</i>	4,000	3,500	7,500
$\overline{video}$	2,000	500	2,500
$\Sigma_{col}$	6,000	4,000	10,000

**Example 5.9 Correlation analysis using lift.** To help filter out misleading “strong” associations of the form  $A \Rightarrow B$  from the data of Example 5.8, we need to study how the two itemsets,  $A$  and  $B$ , are correlated. Let  $\overline{game}$  refer to the transactions of Example 5.8 that do not contain computer games, and  $\overline{video}$  refer to those that do not contain videos. The transactions can be summarized in a *contingency table*, as shown in Table 5.7. From the table, we can see that the probability of purchasing a computer game is  $P(\{game\}) = 0.60$ , the probability of purchasing a video is  $P(\{video\}) = 0.75$ , and the probability of purchasing both is  $P(\{game, video\}) = 0.40$ . By Equation (5.23), the lift of Rule (5.21) is  $P(\{game, video\}) / (P(\{game\}) \times P(\{video\})) = 0.40 / (0.60 \times 0.75) = 0.89$ . Since this value is less than 1, there is a negative correlation between the occurrence of  $\{game\}$  and  $\{video\}$ . The numerator is the likelihood of a customer purchasing both, while the denominator is what the likelihood would have been if the two purchases were completely independent. Such a negative correlation cannot be identified by a support-confidence framework. ■

The second correlation measure that we study is the  $\chi^2$  measure, which was introduced in Chapter 2 (Equation 2.9). To compute the  $\chi^2$  value, we take the squared difference between the observed and expected value for a slot ( $A$  and  $B$  pair) in the contingency table, divided by the expected value. This amount is summed for all slots of the contingency table. Let’s perform a  $\chi^2$  analysis of the above example.

Table 5.8: The above contingency table, now shown with the expected values.

	<i>game</i>	$\overline{game}$	$\Sigma_{row}$
<i>video</i>	4,000 (4,500)	3,500 (3,000)	7,500
$\overline{video}$	2,000 (1,500)	500 (1,000)	2,500
$\Sigma_{col}$	6,000	4,000	10,000

**Example 5.10 Correlation analysis using  $\chi^2$ .** To compute the correlation using  $\chi^2$  analysis, we need the observed value and expected value (displayed in parenthesis) for each slot of the contingency table, as shown in Table 5.8. From the table, we can compute the  $\chi^2$  value as follows:

$$\chi^2 = \sum \frac{(\text{observed} - \text{expected})^2}{\text{expected}} = \frac{(4000 - 4500)^2}{4500} + \frac{(3500 - 3000)^2}{3000} + \frac{(2000 - 1500)^2}{1500} + \frac{(500 - 1000)^2}{1000} = 555.6.$$

Since the  $\chi^2$  value is greater than one, and the observed value of the slot  $(game, video) = 4000$  which is less than the expected value 4500, *buying game* and *buying video* are *negatively correlated*, which is consistent with the conclusion derived from the analysis of the *lift* measure in Example 5.9. ■

Let’s examine two other correlation measures, *all-confidence* and *cosine*, as defined below.

Given an itemset  $X = \{i_1, i_2, \dots, i_k\}$ , the **all\_confidence** of  $X$  is defined as,

$$all\_conf(X) = \frac{sup(X)}{max\_item\_sup(X)} = \frac{sup(X)}{max\{sup(i_j) | \forall i_j \in X\}} \quad (5.24)$$

where  $max\{sup(i_j) | \forall i_j \in X\}$  is the maximum (single) item support of all the items in  $X$ , and hence is called the **max\_item\_sup** of the itemset  $X$ . The *all\_confidence* of  $X$  is the minimal confidence among the set of rules  $i_j \rightarrow X - i_j$ , where  $i_j \in X$ .

Given two itemsets  $A$  and  $B$ , the **cosine** measure of  $A$  and  $B$  is defined as,

$$cosine(A, B) = \frac{P(A \cup B)}{\sqrt{P(A) \times P(B)}} = \frac{sup(A \cup B)}{\sqrt{sup(A) \times sup(B)}} \quad (5.25)$$

The *cosine* measure can be viewed as a harmonized *lift* measure: the two formulae are similar except that for cosine, the *square root* is taken on the product of the probabilities of  $A$  and  $B$ . This is an important difference, however, because by taking the square root, the cosine value is only influenced by the supports of  $A$ ,  $B$  and  $A \cup B$ , and not by the total number of transactions.

“Are these two measures better than lift and  $\chi^2$  in assessing the correlation relationship?” To answer this question, we first look at some other “typical” data sets before returning to our running example.

**Example 5.11 Comparison of four correlation measures on typical data sets.** The correlation relationships between the purchases of two items, *milk* and *coffee*, can be examined by summarizing their purchase history in the form of Table 5.9, a  $2 \times 2$  contingency table, where an entry such as *mc* represents the number of transactions containing both milk and coffee. For the derivation of *all\_confidence*, we let itemset  $X = \{m, c\}$  so that  $sup(X) = mc$  in Equation (5.24).

Table 5.9: A  $2 \times 2$  contingency table for two items.

	<i>milk</i>	$\overline{milk}$	$\Sigma_{row}$
<i>coffee</i>	<i>mc</i>	$\overline{mc}$	<i>c</i>
$\overline{coffee}$	$m\overline{c}$	$\overline{m}\overline{c}$	$\overline{c}$
$\Sigma_{col}$	<i>m</i>	$\overline{m}$	$\Sigma$

Table 5.10: Comparison of four correlation measures using contingency tables for different data sets.

Data Set	<i>mc</i>	$\overline{mc}$	$m\overline{c}$	$\overline{m}\overline{c}$	<i>all_conf.</i>	<i>cosine</i>	<i>lift</i>	$\chi^2$
$A_1$	1,000	100	100	100,000	0.91	0.91	83.64	83,452.6
$A_2$	1,000	100	100	10,000	0.91	0.91	9.26	9,055.7
$A_3$	1,000	100	100	1,000	0.91	0.91	1.82	1,472.7
$A_4$	1,000	100	100	0	0.91	0.91	0.99	9.9
$B_1$	1,000	1,000	1,000	1,000	0.50	0.50	1.00	0.0
$C_1$	100	1,000	1,000	100,000	0.09	0.09	8.44	670.0
$C_2$	1,000	100	10,000	100,000	0.09	0.29	9.18	8,172.8
$C_3$	1	1	100	10,000	0.01	0.07	50.0	48.5

Table 5.10 shows a set of transactional data sets with their corresponding contingency tables and values for each of the four correlation measures. From the table, we see that  $m$  and  $c$  are positively correlated in  $A_1$  through  $A_4$ , independent in  $B_1$ , and negatively correlated in  $C_1$  through  $C_3$ . All four measures are good indicators for the independent case,  $B_1$ . *Lift* and  $\chi^2$  are poor indicators of the other relationships, whereas *all\_confidence* and *cosine* are good indicators. Another interesting fact is that between *all\_confidence* and *cosine*, *cosine* is the better

indicator when  $\overline{mc}$  and  $m\overline{c}$  are not balanced. This is because *cosine* considers the supports of both  $A$  and  $B$  whereas *all\_confidence* considers only the maximal support. Such a difference can be seen by comparing  $C_1$  and  $C_2$ .  $C_1$  should be more negatively correlated for  $m$  and  $c$  than  $C_2$  since  $mc$  is the smallest among the three counts,  $mc$ ,  $\overline{mc}$  and  $m\overline{c}$ , in  $C_1$ . However, this can only be seen by checking the *cosine* measure since the *all\_confidence* values are identical in  $C_1$  and  $C_2$ .

“Why are lift and  $\chi^2$  so poor at distinguishing correlation relationships in the above transactional data sets?” To answer this, we have to consider the *null-transactions*. A **null-transaction** is a transaction that does not contain any of the itemsets being examined. In our example,  $\overline{mc}$  represents the number of null-transactions. Lift and  $\chi^2$  have difficulty distinguishing correlation relationships because they are both strongly influenced by  $\overline{mc}$ . Typically, the number of null-transactions can outweigh the number of individual purchases, since many people may buy neither milk nor coffee. On the other hand, *all\_confidence* and *cosine* values are good indicators of correlation because their definitions remove the influence of  $\overline{mc}$ , i.e., they are not influenced by the number of null-transactions. ■

A measure is **null-invariant** if its value is free from the influence of null-transactions. Null-invariance is an important property for measuring correlations in large transaction databases. Among the four above measures, *all\_confidence* and *cosine* are null-invariant measures.

“Are *all\_confidence* and *cosine* the best at assessing correlation in all cases?” Let’s examine the game-and-video examples again.

**Example 5.12 Comparison of four correlation measures on game-and-video data.** We revisit Examples 5.8 to 5.10. Let  $D_1$  be the original game ( $g$ ) and video ( $v$ ) data set from Table 5.7. We add two more data sets,  $D_0$  and  $D_2$ , where  $D_0$  has zero null-transactions, and  $D_2$  has 10,000 null-transactions (instead of only 500 as in  $D_1$ ). The values of all four correlation measures are shown in Table 5.11.

Table 5.11: Comparison of the four correlation measures for game-and-video data sets.

Data Set	$gv$	$\overline{gv}$	$g\overline{v}$	$\overline{g}\overline{v}$	<i>all_conf.</i>	<i>cosine</i>	<i>lift</i>	$\chi^2$
$D_0$	4,000	3,500	2,000	0	0.53	0.60	0.84	1,477.8
$D_1$	4,000	3,500	2,000	500	0.53	0.60	0.89	555.6
$D_2$	4,000	3,500	2,000	10,000	0.53	0.60	1.73	2,913.0

In Table 5.11,  $gv$ ,  $\overline{gv}$ , and  $g\overline{v}$  remain the same in  $D_0$ ,  $D_1$  and  $D_2$ . However, lift and  $\chi^2$  change from rather negative to rather positive correlations, whereas *all\_confidence* and *cosine* have the nice null-invariant property, and their values remain the same in all cases. Unfortunately, we cannot precisely assert that a set of items are positively or negatively correlated when the value of *all\_confidence* or *cosine* is around 0.5. Strictly based on whether the value is greater than 0.5, we will claim that  $g$  and  $v$  are positively correlated in  $D_1$ , however, it has been shown that they are negatively correlated by the lift and  $\chi^2$  analysis. Therefore, a good strategy is to perform the *all\_confidence* or *cosine* analysis first and when the result shows that they are *weakly* positively/negatively correlated, other analyses can be performed to assist in obtaining a more complete picture. ■

Besides null-invariance, another nice feature of the *all\_confidence* measure is that it has the Apriori-like *downward closure* property. That is, if a pattern is *all-confident* (i.e., passing a minimal *all\_confidence* threshold), so is every one of its subpatterns. In other words, if a pattern is not all-confident, further growth (or specialization) of this pattern will never satisfy the minimal *all\_confidence* threshold. This is obvious since according to Equation (5.24), adding any item into an itemset  $X$  will never increase  $\text{sup}(X)$ , never decrease  $\text{max\_item\_sup}(X)$ , and thus never increase *all\_conf*( $X$ ). This property makes Apriori-like pruning possible: we can prune any patterns that cannot satisfy the minimal *all\_confidence* threshold during the growth of all-confident patterns in mining.

In summary, the use of only support and confidence measures to mine associations results in the generation of a large number of rules, most of which are uninteresting to the user. Instead, we can augment the support-confidence

framework with a correlation measure, resulting in the mining of *correlation rules*. The added measure substantially reduces the number of rules generated, and leads to the discovery of more meaningful rules. However, there seems to be no single correlation measure that works well for all cases. Besides those introduced in this section, many other interestingness measures have been studied in the literature. Unfortunately, most such measures do not have the null-invariance property. Since large data sets typically have many null-transactions, it is important to consider the null-invariance property when selecting appropriate interestingness measures in the correlation analysis. Our analysis shows that both *all\_confidence* and *cosine* are good correlation measures for large applications, although it is wise to augment them with additional tests, such as *lift*, when the test result is not quite conclusive.

## 5.5 Constraint-Based Association Mining

A data mining process may uncover thousands of rules from a given set of data, most of which end up being unrelated or uninteresting to the users. Often, users have a good sense of which “direction” of mining may lead to interesting patterns and the “form” of the patterns or rules they would like to find. Thus, a good heuristic is to have the users specify such intuition or expectations as *constraints* to confine the search space. This strategy is known as **constraint-based mining**. The constraints can include the following:

- **Knowledge type constraints:** These specify the type of knowledge to be mined, such as association or correlation.
- **Data constraints:** These specify the set of task-relevant data.
- **Dimension/level constraints:** These specify the desired dimensions (or attributes) of the data, or levels of the concept hierarchies, to be used in mining.
- **Interestingness constraints:** These specify thresholds on statistical measures of rule interestingness, such as support, confidence, and correlation.
- **Rule constraints:** These specify the form of rules to be mined. Such constraints may be expressed as metarules (rule templates), as the maximum or minimum number of predicates that can occur in the rule antecedent or consequent, or as relationships among attributes, attribute values, and/or aggregates.

The above constraints can be specified using a high-level declarative data mining query language and user interface.

The first four of the above types of constraints have already been addressed in earlier parts of this book and chapter. In this section, we discuss the use of *rule constraints* to focus the mining task. This form of constraint-based mining allows users to describe the rules that they would like to uncover, thereby making the data mining process more *effective*. In addition, a sophisticated mining query optimizer can be used to exploit the constraints specified by the user, thereby making the mining process more *efficient*. Constraint-based mining encourages interactive exploratory mining and analysis. In Section 5.5.1, you will study metarule-guided mining, where syntactic rule constraints are specified in the form of rule templates. Section 5.5.2 discusses the use of additional rule constraints, specifying set/subset relationships, constant initiation of variables, and aggregate functions. For ease of discussion, we assume that the user is searching for association rules. The procedures presented can easily be extended to the mining of correlation rules by adding a correlation measure of interestingness to the support-confidence framework, as described in the previous section.

### 5.5.1 Metarule-Guided Mining of Association Rules

“How are metarules useful?” Metarules allow users to specify the syntactic form of rules that they are interested in mining. The rule forms can be used as constraints to help improve the efficiency of the mining process. Metarules may be based on the analyst’s experience, expectations, or intuition regarding the data, or automatically generated based on the database schema.

**Example 5.13 Metarule-guided mining.** Suppose that as a market analyst for *AllElectronics*, you have access to the data describing customers (such as customer age, address, and credit rating) as well as the list of customer transactions. You are interested in finding associations between customer traits and the items that customers buy. However, rather than finding *all* of the association rules reflecting these relationships, you are particularly interested only in determining which pairs of customer traits promote the sale of office software. A metarule can be used to specify this information describing the form of rules you are interested in finding. An example of such a metarule is

$$P_1(X, Y) \wedge P_2(X, W) \Rightarrow \text{buys}(X, \text{"office software"}) \quad (5.26)$$

where  $P_1$  and  $P_2$  are **predicate variables** that are instantiated to attributes from the given database during the mining process,  $X$  is a variable representing a customer, and  $Y$  and  $W$  take on values of the attributes assigned to  $P_1$  and  $P_2$ , respectively. Typically, a user will specify a list of attributes to be considered for instantiation with  $P_1$  and  $P_2$ . Otherwise, a default set may be used.

In general, a metarule forms a hypothesis regarding the relationships that the user is interested in probing or confirming. The data mining system can then search for rules that match the given metarule. For instance, Rule (5.27) matches or **complies with** Metarule (5.26).

$$\text{age}(X, \text{"30...39"}) \wedge \text{income}(X, \text{"41K...60K"}) \Rightarrow \text{buys}(X, \text{"office software"}) \quad (5.27)$$

■

*"How can metarules be used to guide the mining process?"* Let's examine this problem closely. Suppose that we wish to mine interdimensional association rules, such as in the example above. A metarule is a rule template of the form

$$P_1 \wedge P_2 \wedge \cdots \wedge P_l \Rightarrow Q_1 \wedge Q_2 \wedge \cdots \wedge Q_r \quad (5.28)$$

where  $P_i$  ( $i = 1, \dots, l$ ) and  $Q_j$  ( $j = 1, \dots, r$ ) are either instantiated predicates or predicate variables. Let the number of predicates in the metarule be  $p = l + r$ . In order to find interdimensional association rules satisfying the template,

- We need to find all frequent  $p$ -predicate sets,  $L_p$ .
- We must also have the support or count of the  $l$ -predicate subsets of  $L_p$  in order to compute the confidence of rules derived from  $L_p$ .

This is a typical case of mining multidimensional association rules, which was discussed in Section 5.3.2. By extending such methods using techniques described in the following section, we can derive efficient methods for meta-rule guided mining.

### 5.5.2 Constraint Pushing: Mining Guided by Rule Constraints

Rule constraints specify expected set/subset relationships of the variables in the mined rules, constant initiation of variables, and aggregate functions. Users typically employ their knowledge of the application or data to specify rule constraints for the mining task. These rule constraints may be used together with, or as an alternative to, metarule-guided mining. In this section, we examine rule constraints as to how they can be used to make the mining process more efficient. Let's study an example where rule constraints are used to mine hybrid-dimensional association rules.

**Example 5.14 A closer look at mining guided by rule constraints.** Suppose that *AllElectronics* has a sales multidimensional database with the following interrelated relations:

- *sales(customer\_name, item\_name, TID)*
- *lives\_in(customer\_name, region, city)*
- *item(item\_name, group, price)*
- *transaction(TID, day, month, year)*

where *lives\_in*, *item*, and *transaction* are three dimension tables, linked to the fact table *sales* via three keys, *customer\_name*, *item\_name*, and *TID* (*transaction\_id*), respectively.

Our association mining query is to “Find the sales of what cheap items (where the sum of the prices is less than \$100) that may promote the sales of what expensive items (where the minimum price is \$500) of the same group for Chicago customers in 2004”. This can be expressed in the DMQL data mining query language as follows, where each line of the query has been enumerated to aid in our discussion.

- (1) mine associations as
- (2)  $lives\_in(C, \_, “Chicago”) \wedge sales^+(C, ?\{I\}, \{S\}) \Rightarrow sales^+(C, ?\{J\}, \{T\})$
- (3) from sales
- (4) where  $S.year = 2004$  and  $T.year = 2004$  and  $I.group = J.group$
- (5) group by  $C, I.group$
- (6) having  $sum(I.price) < 100$  and  $min(J.price) \geq 500$
- (7) with support threshold = 1%
- (8) with confidence threshold = 50%

Before we discuss the rule constraints, let us have a closer look at the above query. Line 1 is a knowledge type constraint, where association patterns are to be discovered. Line 2 specifies a metarule. This is an abbreviated form for the following metarule for hybrid-dimensional association rules (multidimensional association rules where the repeated predicate here is *sales*):

$$\begin{aligned}
 &lives\_in(C, \_, “Chicago”) \\
 &\wedge sales(C, ?I_1, S_1) \wedge \dots \wedge sales(C, ?I_k, S_k) \wedge I = \{I_1, \dots, I_k\} \wedge S = \{S_1, \dots, S_k\} \\
 &\Rightarrow sales(C, ?J_1, T_1) \wedge \dots \wedge sales(C, ?J_m, T_m) \wedge J = \{J_1, \dots, J_m\} \wedge T = \{T_1, \dots, T_m\}
 \end{aligned}$$

which means that one or more *sales* records in the form of “*sales(C, ?I<sub>1</sub>, S<sub>1</sub>)*  $\wedge$  ... *sales(C, ?I<sub>k</sub>, S<sub>k</sub>)*” will reside at the rule antecedent (left-hand side), and the question mark “?” means that only *item\_name*,  $I_1, \dots, I_k$  need be printed out. “ $I = \{I_1, \dots, I_k\}$ ” means that all the *I*s at the antecedent are taken from a set *I*, obtained from the SQL-like where clause of line 4. Similar notational conventions are used at the consequent (right-hand side).

The metarule may allow the generation of association rules like the following:

$$\begin{aligned}
 &lives\_in(C, \_, “Chicago”) \wedge sales(C, “Census\_CD”, \_) \wedge \\
 &sales(C, “MS/Office”, \_) \Rightarrow sales(C, “MS/SQLServer”, \_), \quad [1.5\%, 68\%]
 \end{aligned} \tag{5.29}$$

which means that if a customer in Chicago bought “Census.CD” and “MS/Office”, it is likely (with a probability of 68%) that the customer also bought “MS/SQLServer”, and 1.5% of all of the customers bought all three.

Data constraints are specified in the “*lives\_in(, , “Chicago”)*” portion of the metarule (i.e., all the customers who live in Chicago), and in line 3, which specifies that only the fact table, *sales*, need be explicitly referenced. In such a multidimensional database, variable reference is simplified. For example, “*S.year = 2004*” is equivalent to the SQL statement “from *sales S*, *transaction T* where *S.TID = T.TID* and *T.year = 2004*”. All three dimensions (*lives\_in*, *item*, and *transaction*) are used. Level constraints are as follows: for *lives\_in*, we consider just *customer\_name* since *region* is not referenced and *city* = “Chicago” is only used in the selection; for *item*, we consider the levels *item\_name* and *group* since they are used in the query; and for *transaction*, we are only concerned with *TID* since *day* and *month* are not referenced and *year* is used only in the selection.



Rule constraints include most portions of the **where** (line 4) and **having** (line 6) clauses, such as “ $S.year = 2004$ ”, “ $T.year = 2004$ ”, “ $I.group = J.group$ ”, “ $sum(I.price) \leq 100$ ”, and “ $min(J.price) \geq 500$ ”. Finally, lines 7 and 8 specify two interestingness constraints (i.e., thresholds), namely, a minimum support of 1% and a minimum confidence of 50%. ■

Dimension/level constraints and interestingness constraints can be applied after mining, to filter out discovered rules, although it is generally more efficient and less expensive to use them *during* mining, to help prune the search space. Dimension/level constraints were discussed in Section 5.3, and interestingness constraints have been discussed throughout this chapter. Let’s focus now on rule constraints.

*“How can we use rule constraints to prune the search space? More specifically, what kind of rule constraints can be ‘pushed’ deep into the mining process and still ensure the completeness of the answer returned for a mining query?”*

Rule constraints can be classified into the following five categories with respect to frequent itemset mining: (1) *antimonotonic*, (2) *monotonic*, (3) *succinct*, (4) *convertible*, and (5) *inconvertible*. For each category, we will use an example to show its characteristics and explain how such kinds of constraints can be used in the mining process.

The first category of constraints is **antimonotonic**. Consider the rule constraint “ $sum(I.price) \leq 100$ ” of Example 5.14. Suppose we are using the Apriori framework, which at each iteration  $k$ , explores itemsets of size  $k$ . If the price summation of the items in an itemset is no less than 100, this itemset can be pruned from the search space, since adding more items into the set will only make it more expensive and thus will never satisfy the constraint. In other words, if an itemset does not satisfy this rule constraint, none of its supersets can satisfy the constraint. If a rule constraint obeys this property, it is **antimonotonic**. Pruning by antimonotonic constraints can be applied at each iteration of Apriori-style algorithms to help improve the efficiency of the overall mining process while guaranteeing completeness of the data mining task.

The Apriori property, which states that all nonempty subsets of a frequent itemset must also be frequent, is antimonotonic. If a given itemset does not satisfy minimum support, none of its supersets can. This property is used at each iteration of the Apriori algorithm to reduce the number of candidate itemsets examined, thereby reducing the search space for association rules.

Other examples of antimonotonic constraints include “ $min(J.price) \geq 500$ ”, “ $count(I) \leq 10$ ”, and so on. Any itemset that violates either of these constraints can be discarded since adding more items to such itemsets can never satisfy the constraints. Note that a constraint such as “ $avg(I.price) \leq 100$ ” is not antimonotonic. For a given itemset that does not satisfy this constraint, a superset created by adding some (cheap) items may result in satisfying the constraint. Hence, pushing this constraint inside the mining process will not guarantee completeness of the data mining task. A list of SQL-primitives-based constraints is given in the first column of Table 5.12. The antimonotonicity of the constraints is indicated in the second column of the table. To simplify our discussion, only existence operators (e.g.,  $=$ ,  $\in$ , but not  $\neq$ ,  $\notin$ ) and comparison (or containment) operators with equality (e.g.,  $\leq$ ,  $\subseteq$ ) are given.

The second category of constraints is **monotonic**. If the rule constraint in Example 5.14 were “ $sum(I.price) \geq 100$ ”, the constraint-based processing method would be quite different. If an itemset  $I$  satisfies the constraint, that is, the sum of the prices in the set is no less than 100, further addition of more items to  $I$  will increase cost and will always satisfy the constraint. Therefore, further testing of this constraint on itemset  $I$  becomes redundant. In other words, if an itemset satisfies this rule constraint, so do all of its supersets. If a rule constraint obeys this property, it is **monotonic**. Similar rule monotonic constraints include “ $min(I.price) \leq 10$ ”, “ $count(I) \geq 10$ ”, and so on. The monotonicity of the list of SQL-primitives-based constraints is indicated in the third column of Table 5.12.

The third category is **succinct constraints**. For this category of constraints, we can *enumerate all and only those sets that are guaranteed to satisfy the constraint*. That is, if a rule constraint is **succinct**, we can directly generate precisely the sets that satisfy it, even before support counting begins. This avoids the substantial overhead of the generate-and-test paradigm. In other words, such constraints are *precounting prunable*. For example, the constraint “ $min(J.price) \geq 500$ ” in Example 5.14 is succinct. This is because we can explicitly and precisely

Table 5.12: Characterization of commonly used SQL-based constraints.

Constraint	Antimonotonic	Monotonic	Succinct
$v \in S$	no	yes	yes
$S \supseteq V$	no	yes	yes
$S \subseteq V$	yes	no	yes
$\min(S) \leq v$	no	yes	yes
$\min(S) \geq v$	yes	no	yes
$\max(S) \leq v$	yes	no	yes
$\max(S) \geq v$	no	yes	yes
$\text{count}(S) \leq v$	yes	no	weakly
$\text{count}(S) \geq v$	no	yes	weakly
$\text{sum}(S) \leq v \ (\forall a \in S, a \geq 0)$	yes	no	no
$\text{sum}(S) \geq v \ (\forall a \in S, a \geq 0)$	no	yes	no
$\text{range}(S) \leq v$	yes	no	no
$\text{range}(S) \geq v$	no	yes	no
$\text{avg}(S) \theta v, \theta \in \{\leq, \geq\}$	convertible	convertible	no
$\text{support}(S) \geq \xi$	yes	no	no
$\text{support}(S) \leq \xi$	no	yes	no
$\text{all\_confidence}(S) \geq \xi$	yes	no	no
$\text{all\_confidence}(S) \leq \xi$	no	yes	no

generate all the sets of items satisfying the constraint. Specifically, such a set must contain at least one item whose price is no less than \$500. It is of the form  $S_1 \cup S_2$ , where  $S_1 \neq \emptyset$  is a subset of the set of all those items with prices no less than \$500, and  $S_2$ , possibly empty, is a subset of the set of all those items with prices no greater than \$500. Because there is a precise “formula” for generating all of the sets satisfying a succinct constraint, there is no need to iteratively check the rule constraint during the mining process. The succinctness of the list of SQL-primitives-based constraints is indicated in the fourth column of Table 5.12.<sup>10</sup>

The fourth category is **convertible constraints**. Some constraints belong to none of the above three categories. However, if the items in the itemset are arranged in a particular order, the constraint may become monotonic or antimonotonic with regard to the frequent itemset mining process. For example, the constraint “ $\text{avg}(I.\text{price}) \leq 100$ ” is neither antimonotonic nor monotonic. However, if items in a transaction are added to an itemset in price-ascending order, the constraint becomes *antimonotonic*, because if an itemset  $I$  violates the constraint (i.e., with an average price greater than \$100), further addition of more expensive items into the itemset will never make it satisfy the constraint. Similarly, if items in a transaction are added to an itemset in price-descending order, it becomes *monotonic*, because if the itemset satisfies the constraint (i.e., with an average price no greater than \$100), adding cheaper items into the current itemset will still make the average price no greater than \$100. Aside from “ $\text{avg}(S) \leq v$ ,” and “ $\text{avg}(S) \geq v$ ,” given in Table 5.12, there are many other convertible constraints, such as “ $\text{variance}(S) \geq v$ ,” “ $\text{standard\_deviation}(S) \geq v$ ,” and so on.

Note that the above discussion does not imply that every constraint is convertible. For example, “ $\text{sum}(S) \theta v$ ,” where  $\theta \in \{\leq, \geq\}$  and each element in  $S$  could be of any real value, is not convertible. Therefore, there is yet a fifth category of constraints, called **inconvertible constraints**. The good news is that although there still exist some tough constraints that are not convertible, most simple SQL expressions with built-in SQL aggregates belong to one of the first four categories to which efficient constraint mining methods can be applied.

## 5.6 Summary

- The discovery of frequent patterns, association and correlation relationships among huge amounts of data is useful in selective marketing, decision analysis, and business management. A popular area of application

<sup>10</sup>For constraint  $\text{count}(S) \leq v$  (and similarly for  $\text{count}(S) \geq v$ ), we can have a member generation function based on a cardinality constraint, i.e.,  $\{X \mid X \subseteq \text{Itemset} \wedge |X| \leq v\}$ . Member generation in this manner takes a different flavor and thus is called *weakly succinct*.

is **market basket analysis**, which studies the buying habits of customers by searching for sets of items that are frequently purchased together (or in sequence). **Association rule mining** consists of first finding **frequent itemsets** (set of items, such as  $A$  and  $B$ , satisfying a *minimum support threshold*, or percentage of the task-relevant tuples), from which **strong** association rules in the form of  $A \Rightarrow B$  are generated. These rules also satisfy a *minimum confidence threshold* (a prespecified probability of satisfying  $B$  under the condition that  $A$  is satisfied). Associations can be further analyzed to uncover **correlation rules**, which convey statistical correlations between itemsets  $A$  and  $B$ .

- **Frequent pattern mining** can be categorized in many different ways according to various criteria, such as the following:
  1. Based on the **completeness** of patterns to be mined, categories of frequent pattern mining include mining the *complete set of frequent itemsets*, the *closed frequent itemsets*, the *maximal frequent itemsets*, *constrained frequent itemsets*, and so on.
  2. Based on the **levels** and **dimensions** of data involved in the rule, categories can include the mining of *single-level association rules*, *multilevel association rules*, *single-dimensional association rules*, and *multidimensional association rules*.
  3. Based on the **types of values** handled in the rule, the categories can include mining *Boolean association rules*, and *quantitative association rules*.
  4. Based on the **kinds of rules** to be mined, categories include mining *association rules*, and *correlation rules*.
  5. Based on the **kinds of patterns** to be mined, frequent pattern mining can be classified into *frequent itemset mining*, *sequential pattern mining*, *structured pattern mining*, and so on. This chapter has focused on frequent itemset mining.
- Many efficient and scalable algorithms have been developed for **frequent itemset mining**, from which association and correlation rules can be derived. These algorithms can be classified into three categories: (1) *Apriori-like algorithms*, (2) *frequent-pattern growth*-based algorithms, such as FP-growth, and (3) *algorithms that use the vertical data format*.
- The **Apriori algorithm** is a seminal algorithm for mining frequent itemsets for Boolean association rules. It explores the level-wise mining Apriori property that *all nonempty subsets of a frequent itemset must also be frequent*. At the  $k$ th iteration (for  $k \geq 2$ ), it forms frequent  $k$ -itemset candidates based on the frequent  $(k - 1)$ -itemsets, and scans the database once to find the *complete* set of frequent  $k$ -itemsets,  $L_k$ .  
 Variations involving hashing and transaction reduction can be used to make the procedure more efficient. Other variations include partitioning the data (mining on each partition and then combining the results), and sampling the data (mining on a subset of the data). These variations can reduce the number of data scans required to as little as two or one.
- **Frequent pattern growth (FP-growth)** is a method of mining frequent itemsets without candidate generation. It constructs a highly compact data structure (an *FP-tree*) to compress the original transaction database. Rather than employing the generate-and-test strategy of Apriori-like methods, it focuses on frequent pattern (fragment) growth, which avoids costly candidate generation, resulting in greater efficiency.
- **Mining frequent itemsets using vertical data format (Eclat)** is a method that transforms a given data set of transactions in the horizontal data format of *TID-itemset* into the vertical data format of *item-TID\_set*. It mines the transformed data set by TID\_set intersections based on the Apriori property and additional optimization techniques, such as *diffset*.
- Methods for mining frequent itemsets can be extended for the mining of **closed frequent itemsets** (from which the set of frequent itemsets can easily be derived). These incorporate additional optimization techniques, such as *item merging*, *sub-itemset pruning* and *item skipping*, as well as efficient *subset checking* of generated itemsets in a *pattern-tree*.

- Mining frequent itemsets and associations has been **extended in various ways** to include mining *multilevel association rules* and *multidimensional association rules*.
- **Multilevel association rules** can be mined using several strategies, based on how minimum support thresholds are defined at each level of abstraction, such as *uniform support*, *reduced support* and *group-based support*. Redundant multilevel (descendent) association rules can be eliminated if their support and confidence are close to their expected values, based on their corresponding ancestor rules.
- Techniques for mining **multidimensional association rules** can be categorized according to their treatment of quantitative attributes. First, quantitative attributes may be *discretized statically*, based on predefined concept hierarchies. Data cubes are well suited to this approach, since both the data cube and quantitative attributes can make use of concept hierarchies. Second, **quantitative association rules** can be mined where quantitative attributes are discretized dynamically based on binning and/or clustering, where “adjacent” association rules may be further combined by clustering to generate concise and meaningful rules.
- Not all strong association rules are interesting. It is more effective to mine items that are statistically correlated. Therefore, association rules should be augmented with a correlation measure to generate more meaningful **correlation rules**. There are several correlation measures to choose from, including **lift**,  $\chi^2$ , **all\_confidence**, and **cosine**. A measure is **null-invariant** if its value is free from the influence of **null-transactions** (i.e., *transactions that do not contain any of the itemsets being examined*). Since large databases typically have numerous null-transactions, a null-invariant correlation measure should be used, such as *all\_confidence* or *cosine*. When interpreting correlation measure values, it is important to understand their implications and limitations.
- **Constraint-based rule mining** allow users to focus the search for rules by providing metarules (i.e., pattern templates) and additional mining constraints. Such mining is facilitated with the use of a declarative data mining query language and user interface, and poses great challenges for mining query optimization. Rule constraints can be classified into five categories: **antimonotonic**, **monotonic**, **succinct**, **convertible**, and **inconvertible**. Constraints belonging to the first four of these categories can be used during frequent itemset mining to guide the process, leading to more efficient and effective mining.
- **Association rules should not be used directly for prediction** without further analysis or domain knowledge. They do not necessarily indicate causality. They are, however, a helpful starting point for further exploration, making them a popular tool for understanding data. The application of frequent patterns to classification, cluster analysis, and other data mining tasks will be discussed in subsequent chapters.

## 5.7 Exercises

1. The Apriori algorithm makes use of *prior knowledge* of subset support properties.
  - (a) Prove that all nonempty subsets of a frequent itemset must also be frequent.
  - (b) Prove that the support of any nonempty subset  $s'$  of itemset  $s$  must be at least as great as the support of  $s$ .
  - (c) Given frequent itemset  $l$  and subset  $s$  of  $l$ , prove that the confidence of the rule “ $s' \Rightarrow (l - s')$ ” cannot be more than the confidence of “ $s \Rightarrow (l - s)$ ”, where  $s'$  is a subset of  $s$ .
  - (d) A *partitioning* variation of Apriori subdivides the transactions of a database  $D$  into  $n$  nonoverlapping partitions. Prove that any itemset that is frequent in  $D$  must be frequent in at least one partition of  $D$ .
2. Section 5.2.2 describes a method for *generating association rules* from frequent itemsets. Propose a more efficient method. Explain why it is more efficient than the one proposed in Section 5.2.2. (*Hint*: Consider incorporating the properties of Exercise 5.1(b) and 5.1(c) into your design.)
3. A database has 5 transactions. Let  $\text{min\_sup} = 60\%$  and  $\text{min\_conf} = 80\%$ .

<i>TID</i>	<i>items_bought</i>
T100	{M, O, N, K, E, Y}
T200	{D, O, N, K, E, Y}
T300	{M, A, K, E}
T400	{M, U, C, K, Y}
T500	{C, O, O, K, I, E}

- (a) Find all frequent itemsets using Apriori and FP-growth, respectively. Compare the efficiency of the two mining processes.
- (b) List all of the *strong* association rules (with support  $s$  and confidence  $c$ ) matching the following metarule, where  $X$  is a variable representing customers, and  $item_i$  denotes variables representing items (e.g., “A”, “B”, etc.):

$$\forall x \in \text{transaction}, \text{buys}(X, item_1) \wedge \text{buys}(X, item_2) \Rightarrow \text{buys}(X, item_3) \quad [s, c]$$

4. (**Implementation project**) Implement three *frequent itemset mining* algorithms introduced in this chapter: (1) Apriori [AS94b], (2) FP-growth [HPY00], and (3) Eclat [Zak00] (mining using vertical data format), using a programming language that you are familiar with, such as C++ or Java. Compare the performance of each algorithm with various kinds of large data sets. Write a report to analyze the situations (such as data size, data distribution, minimal support threshold setting, and pattern density) where one algorithm may perform better than the others, and state why.
5. A database has four transactions. Let  $min\_sup = 60\%$  and  $min\_conf = 80\%$ .

<i>cust_ID</i>	<i>TID</i>	<i>items_bought</i> (in the form of <i>brand-item_category</i> )
01	T100	{King’s-Crab, Sunset-Milk, Dairyland-Cheese, Best-Bread}
02	T200	{Best-Cheese, Dairyland-Milk, Goldenfarm-Apple, Tasty-Pie, Wonder-Bread}
01	T300	{Westcoast-Apple, Dairyland-Milk, Wonder-Bread, Tasty-Pie}
03	T400	{Wonder-Bread, Sunset-Milk, Dairyland-Cheese}

- (a) At the granularity of *item\_category* (e.g.,  $item_i$  could be “Milk”), for the following rule template,

$$\forall X \in \text{transaction}, \text{buys}(X, item_1) \wedge \text{buys}(X, item_2) \Rightarrow \text{buys}(X, item_3) \quad [s, c]$$

list the frequent  $k$ -itemset for the largest  $k$ , and *all* of the *strong* association rules (with their support  $s$  and confidence  $c$ ) containing the frequent  $k$ -itemset for the largest  $k$ .

- (b) At the granularity of *brand-item\_category* (e.g.,  $item_i$  could be “Sunset-Milk”), for the following rule template,

$$\forall X \in \text{customer}, \text{buys}(X, item_1) \wedge \text{buys}(X, item_2) \Rightarrow \text{buys}(X, item_3)$$

list the frequent  $k$ -itemset for the largest  $k$  (but do not print any rules).

6. Suppose that a large store has a transaction database that is *distributed* among four locations. Transactions in each component database have the same format, namely  $T_j : \{i_1, \dots, i_m\}$ , where  $T_j$  is a transaction identifier, and  $i_k$  ( $1 \leq k \leq m$ ) is the identifier of an item purchased in the transaction. Propose an efficient algorithm to mine global association rules (without considering multilevel associations). You may present your algorithm in the form of an outline. Your algorithm should not require shipping all of the data to one site and should not cause excessive network communication overhead.
7. Suppose that frequent itemsets are saved for a large transaction database,  $DB$ . Discuss how to efficiently mine the (global) association rules under the same minimum support threshold, if a set of new transactions, denoted as  $\Delta DB$ , is (*incrementally*) added in?
8. [Contributed by Tao Cheng] Most frequent pattern mining algorithms consider only distinct items in a transaction. However, multiple occurrences of an item in the same shopping basket, such as four cakes and three jugs of milk, can be important in transaction data analysis. How can one mine frequent itemsets efficiently considering multiple occurrences of items? Propose modifications to the well-known algorithms, such as Apriori and FP-growth, to adapt to such a situation.

9. (**Implementation project**) Implement three *closed frequent itemset mining* methods (1) A-Close [PBT99] (based on an extension of Apriori [AS94b]), (2) CLOSET+ [WHP03] (based on an extension of FP-growth [HPY00]), and (3) CHARM [ZH02] (based on an extension of Eclat [Zak00]). Compare their performance with various kinds of large data sets. Write a report to answer the following questions:
- (a) Why is mining the set of closed frequent itemsets often more desirable than mining the complete set of frequent itemsets (based on your experiments on the same data set as Exercise 4)?
  - (b) Analyze at what situations (such as data size, data distribution, minimal support threshold setting, and pattern density) and why that one algorithm performs better than the others.
10. Suppose that a data relation describing students at *Big-University* has been generalized to the generalized relation  $R$  in Table 5.13.

Table 5.13: Generalized relation for Exercise 5.9.

major	status	age	nationality	gpa	count
French	M.A	over_30	Canada	2.8_3.2	3
cs	junior	16...20	Europe	3.2_3.6	29
physics	M.S	26...30	Latin_America	3.2_3.6	18
engineering	Ph.D	26...30	Asia	3.6_4.0	78
philosophy	Ph.D	26...30	Europe	3.2_3.6	5
French	senior	16...20	Canada	3.2_3.6	40
chemistry	junior	21...25	USA	3.6_4.0	25
cs	senior	16...20	Canada	3.2_3.6	70
philosophy	M.S	over_30	Canada	3.6_4.0	15
French	junior	16...20	USA	2.8_3.2	8
philosophy	junior	26...30	Canada	2.8_3.2	9
philosophy	M.S	26...30	Asia	3.2_3.6	9
French	junior	16...20	Canada	3.2_3.6	52
math	senior	16...20	USA	3.6_4.0	32
cs	junior	16...20	Canada	3.2_3.6	76
philosophy	Ph.D	26...30	Canada	3.6_4.0	14
philosophy	senior	26...30	Canada	2.8_3.2	19
French	Ph.D	over_30	Canada	2.8_3.2	1
engineering	junior	21...25	Europe	3.2_3.6	71
math	Ph.D	26...30	Latin_America	3.2_3.6	7
chemistry	junior	16...20	USA	3.6_4.0	46
engineering	junior	21...25	Canada	3.2_3.6	96
French	M.S	over_30	Latin_America	3.2_3.6	4
philosophy	junior	21...25	USA	2.8_3.2	8
math	junior	16...20	Canada	3.6_4.0	59

Let the concept hierarchies be as follows:

*status* :             $\{freshman, sophomore, junior, senior\} \in undergraduate.$   
                        $\{M.Sc., M.A., Ph.D.\} \in graduate.$   
*major* :             $\{physics, chemistry, math\} \in science.$   
                        $\{cs, engineering\} \in appl\_sciences.$   
                        $\{French, philosophy\} \in arts.$   
*age* :                 $\{16...20, 21...25\} \in young.$   
                        $\{26...30, over\_30\} \in old.$   
*nationality* :       $\{Asia, Europe, Latin\_America\} \in foreign.$

$$\{U.S.A., Canada\} \in North\_America.$$

Let the minimum support threshold be 20% and the minimum confidence threshold be 50% (at each of the levels).

- (a) Draw the concept hierarchies for *status*, *major*, *age*, and *nationality*.
- (b) Write a program to find the set of strong multilevel association rules in  $R$  using *uniform support* for all levels, for the following rule template,

$$\forall S \in R, P(S, x) \wedge Q(S, y) \Rightarrow gpa(S, z) \quad [s, c]$$

where  $P, Q \in \{\textit{status}, \textit{major}, \textit{age}, \textit{nationality}\}$ .

- (c) Use the program to find the set of strong multilevel association rules in  $R$  using *level-cross filtering by single items*. In this strategy, an item at the  $i$ th level is examined if and only if its parent node at the  $(i - 1)$ th level in the concept hierarchy is frequent. That is, if a node is frequent, its children will be examined; otherwise, its descendants are pruned from the search. Use a reduced support of 10% for the lowest abstraction level, for the preceding rule template.
11. Propose and outline a **level-shared mining** approach to mining multilevel association rules in which each item is encoded by its level position, and an initial scan of the database collects the count for each item *at each concept level*, identifying frequent and subfrequent items. Comment on the processing cost of mining multilevel associations with this method in comparison to mining single-level associations.
  12. (**Implementation project**) Many techniques have been proposed to further improve the performance of frequent-itemset mining algorithms. Taking FP-tree-based frequent pattern-growth algorithms, such as FP-growth, as an example, implement one of the following optimization techniques, and compare the performance of your new implementation with the one that does not incorporate such optimization.
    - (a) The previously proposed frequent pattern mining with FP-tree generates conditional pattern bases using a bottom-up projection technique, i.e., project on the prefix path of an item  $p$ . However, one can develop a **top-down projection** technique, i.e., project on the suffix path of an item  $p$  in the generation of a conditional pattern-base. Design and implement such a top-down FP-tree mining method and compare your performance with the bottom-up projection method.
    - (b) Nodes and pointers are used uniformly in FP-tree in the design of the FP-growth algorithm. However, such a structure may consume a lot of space when the data are sparse. One possible alternative design is to explore **array-and pointer- based hybrid implementation**, where a node may store multiple items when a node contains no splitting point to multiple sub-branches. Develop such an implementation and compare it with the original one.
    - (c) It is time- and space- consuming to generate numerous conditional pattern bases during pattern-growth mining. One interesting alternative is to **push right** the branches that have been mined for a particular item  $p$ , that is, to push them to the remaining branch(es) of the FP-tree. This is done so that fewer conditional pattern bases have to be generated and additional sharing can be explored when mining the remaining branches of the FPtree. Design and implement such a method and conduct a performance study on it.
  13. Give a short example to show that items in a strong association rule may actually be *negatively correlated*.
  14. The following contingency table summarizes supermarket transaction data, where *hot dogs* refers to the transactions containing hot dogs, *hotdogs* refers to the transactions that do not contain hot dogs, *hamburgers* refers to the transactions containing hamburgers, and *hamburgers* refers to the transactions that do not contain hamburgers.

	<i>hot dogs</i>	<i>hotdogs</i>	$\Sigma_{row}$
<i>hamburgers</i>	2000	500	2500
<i>hamburgers</i>	1000	1500	2500
$\Sigma_{col}$	3000	2000	5000

- (a) Suppose that the association rule “*hot dogs*  $\Rightarrow$  *hamburgers*” is mined. Given a minimum support threshold of 25% and a minimum confidence threshold of 50%, is this association rule strong?
- (b) Based on the given data, is the purchase of *hot dogs* independent of the purchase of *hamburgers*? If not, what kind of *correlation* relationship exists between the two?
15. In multidimensional data analysis, it is interesting to extract pairs of *similar* cell characteristics associated with substantial changes in measure in a data cube, where cells are considered *similar* if they are related by roll-up (i.e., *ancestors*), drill-down (i.e., *descendants*), or 1-dimensional mutation (i.e., *siblings*) operations. Such an analysis is called **cube gradient analysis**. Suppose the measure of the cube is *average*. A user poses a set of *probe cells* and would like to find their corresponding sets of *gradient cells* each of which satisfies a certain gradient threshold. For example, find the set of corresponding gradient cells whose average sale price is greater than 20% of that of the given probe cells. Develop an algorithm than mines the set of constrained gradient cells efficiently in a large data cube.
16. Association rule mining often generates a large number of rules. Discuss effective methods that can be used to reduce the number of rules generated while still preserving most of the interesting rules.
17. Sequential patterns can be mined in methods similar to the mining of association rules. Design an efficient algorithm to mine **multilevel sequential patterns** from a transaction database. An example of such a pattern is the following: “*A customer who buys a PC will buy Microsoft software within three months*”, on which one may drill down to find a more refined version of the pattern, such as “*A customer who buys a Pentium PC will buy Microsoft Office within three months*”.
18. Prove that each entry in the following table correctly characterizes its corresponding rule constraint for frequent itemset mining.

	Rule constraint	Antimonotonic	Monotonic	Succinct
a)	$v \in S$	no	yes	yes
b)	$S \subseteq V$	yes	no	yes
c)	$\min(S) \leq v$	no	yes	yes
d)	$\text{range}(S) \leq v$	yes	no	no
e)	$\text{variance}(S) \leq v$	convertible	convertible	no

19. The price of each item in a store is nonnegative. The store manager is only interested in rules of the form: “*one free item may trigger \$200 total purchases in the same transaction*”. State how to mine such rules *efficiently*.
20. The price of each item in a store is nonnegative. For each of the following cases, identify the kinds of constraint they represent and briefly discuss how to mine such association rules *efficiently*.
- (a) Containing at least one Nintendo game
- (b) Containing items whose sum of the prices is less than \$150
- (c) Containing one free item and other items whose sum of the prices is at least \$200
- (d) Where the average price of all the items is between \$100 and \$500



## 5.8 Bibliographic Notes

Association rule mining was first proposed by Agrawal, Imielinski, and Swami [AIS93]. The Apriori algorithm discussed in Section 5.2.1 for frequent itemset mining was presented in Agrawal and Srikant [AS94b]. A variation of the algorithm using a similar pruning heuristic was developed independently by Mannila, Tiovonen, and Verkamo [MTV94]. A joint publication combining these works later appeared in Agrawal, Mannila, Srikant, Toivonen, and Verkamo [AMS<sup>+</sup>96]. A method for generating association rules from frequent itemsets is described in Agrawal and Srikant [AS94a].

References for the variations of Apriori described in Section 5.2.3 include the following. The use of hash tables to improve association mining efficiency was studied by Park, Chen, and Yu [PCY95a]. Transaction reduction techniques are described in Agrawal and Srikant [AS94b], Han and Fu [HF95], and Park, Chen, and Yu [PCY95a]. The partitioning technique was proposed by Savasere, Omiecinski, and Navathe [SON95]. The sampling approach is discussed in Toivonen [Toi96]. A dynamic itemset counting approach is given in Brin, Motwani, Ullman, and Tsur [BMUT97]. An efficient incremental updating of mined association rules was proposed by Cheung, Han, Ng, and Wong [CHNW96]. Parallel and distributed association data mining under the Apriori framework was studied by Park, Chen, and Yu [PCY95b], Agrawal and Shafer [AS96], and Cheung, Han, Ng, et al. [CHN<sup>+</sup>96]. Another parallel association mining method, which explores itemset clustering using a vertical database layout, was proposed in Zaki, Parthasarathy, Ogihara, and Li [ZPOL97].

Other scalable frequent itemset mining methods have been proposed as alternatives to the Apriori-based approach. FP-growth, a pattern-growth approach for mining frequent itemsets without candidate generation, was proposed by Han, Pei, and Yin [HPY00] (Section 5.2.4). An exploration of hyper-structure mining of frequent patterns, called H-Mine, was proposed by Pei, Han, Lu, Nishio, Tang, and Yang [PHMA<sup>+</sup>01]. OP, a method that integrates top-down and bottom-up traversal of FP-trees in pattern-growth mining, was proposed by Liu, Pan, Wang, and Han [LPWH02]. An array-based implementation of prefix-tree-structure for efficient pattern growth mining was proposed by Grahne and Zhu [GZ03b]. Eclat, an approach for mining frequent itemsets by exploring the vertical data format, was proposed by Zaki [Zak00]. A depth-first generation of frequent itemsets was proposed by Agarwal, Aggarwal, and Prasad [AAP01].

The mining of frequent closed itemsets was proposed in Pasquier, Bastide, Taouil, and Lakhal [PBTL99], where an Apriori-based algorithm called A-Close for such mining was presented. CLOSET, an efficient closed itemset mining algorithm based on the frequent pattern growth method, was proposed by Pei, Han, and Mao [PHM00], and further refined as CLOSET+ in Wang, Han and Pei [WHP03]. A prefix-tree-based algorithm, called FPClose, for mining closed itemsets using pattern growth approach, was proposed by Grahne and Zhu [GZ03b]. An extension for mining closed frequent itemsets with the vertical data format, called CHARM, was proposed by Zaki and Hsiao [ZH02]. Mining max-patterns was first studied by Bayardo [Bay98]. Another efficient method for mining maximal frequent itemsets using vertical data format, called MAFIA, was proposed by Burdick, Calimlim, and Gehrke [BCG01]. AFOPT, a method that explores a *right push* operation on FP-trees during the mining process, was proposed by Liu, Lu, Lou and Yu [LLLY03]. Pan, Cong, Tung, et al. [PCT<sup>+</sup>03] proposed CARPENTER, a method for finding closed patterns in long biological datasets, which integrates the advantages of vertical data formats and pattern growth methods. A FIMI (Frequent Itemset Mining Implementation) workshop dedicated to the implementation methods of frequent itemset mining was reported by Goethals and Zaki [GZ03a].

Frequent itemset mining has various extensions, including sequential pattern mining (Agrawal and Srikant [AS95]), episodes mining (Mannila, Toivonen, and Verkamo [MTV97]), spatial association rule mining (Koperski and Han [KH95]), cyclic association rule mining (Özden, Ramaswamy, and Silberschatz [ORS98]), negative association rule mining (Savasere, Omiecinski and Navathe [SON98]), intertransaction association rule mining (Lu, Han, and Feng [LHF98]), and calendric market basket analysis (Ramaswamy, Mahajan, and Silberschatz [RMS98]). Multilevel association mining was studied in Han and Fu [HF95], and Srikant and Agrawal [SA95]. In Srikant and Agrawal [SA95], such mining was studied in the context of *generalized association rules*, and an R-interest measure was proposed for removing redundant rules. A non-grid-based technique for mining quantitative association rules, which uses a measure of partial completeness, was proposed by Srikant and Agrawal [SA96]. The ARCS system for mining quantitative association rules based on rule clustering was proposed by Lent, Swami, and Widom [LSW97]. Techniques for mining quantitative rules based on x-monotone and rectilinear regions were presented by Fukuda,

Morimoto, Morishita, and Tokuyama [FMMT96], and Yoda, Fukuda, Morimoto, et al. [YFM<sup>+</sup>97]. Mining multi-dimensional association rules using static discretization of quantitative attributes and data cubes was studied by Kamber, Han, and Chiang [KHC97]. Mining (distance-based) association rules over interval data was proposed by Miller and Yang [MY97]. Mining quantitative association rules based on a statistical theory to present only those that deviate substantially from normal data was studied by Aumann and Lindell [AL99].

The problem of mining interesting rules has been studied by many researchers. The statistical independence of rules in data mining was studied by Piatetski-Shapiro [PS91b]. The interestingness problem of strong association rules is discussed in Chen, Han, and Yu [CHY96], Brin, Motwani, and Silverstein [BMS97], and Aggarwal and Yu [AY99], which cover several interestingness measures including *lift*. An efficient method for generalizing associations to correlations is given in Brin, Motwani, and Silverstein [BMS97]. Other alternatives to the support-confidence framework for assessing the interestingness of association rules are proposed in Brin, Motwani, Ullman, and Tsur [BMUT97] and Ahmed, El-Makky, and Taha [AEMT00]. A method for mining strong gradient relationships among itemsets was proposed by Imielinski, Khachiyan, and Abdulghani [IKA02]. Silverstein, Brin, Motwani, and Ullman [SBMU98] studied the problem of mining causal structures over transaction databases. Some comparative studies of different interestingness measures were done by Hilderman and Hamilton [HH01] and by Tan, Kumar and Srivastava [TKS02]. The use of *all\_confidence* as a correlation measure for generating interesting association rules was studied by Omiecinski [Omi03] and by Lee, Kim, Cai and Han [LKCH03].

To reduce the huge set of frequent patterns generated in data mining, recent studies have been working on mining compressed set of frequent patterns. Mining closed patterns can be viewed as lossless compression of frequent patterns. Lossy compression of patterns include maximal patterns by Bayardo [Bay98]), top-*k* patterns by Wang, Han, Lu, and Tsvetkov [WHLT05], and error-tolerant patterns by Yang, Fayyad, and Bradley [YFB01]. Afrati, Gionis, and Mannila [AGM04] proposed to use *K* itemsets to cover a collection of frequent itemsets. Yan, Cheng, Xin and Han proposed a profile-based approach [YCXH05] and Xin, Han, Yan, and Cheng proposed a clustering-based approach [XHYC05] for frequent itemset compression.

The use of metarules as syntactic or semantic filters defining the form of interesting single-dimensional association rules was proposed in Klemettinen, Mannila, Ronkainen, et al. [KMR<sup>+</sup>94]. Metarule-guided mining, where the metarule consequent specifies an action (such as Bayesian clustering or plotting) to be applied to the data satisfying the metarule antecedent, was proposed in Shen, Ong, Mitbender, and Zaniolo [SOMZ96]. A relation-based approach to metarule-guided mining of association rules was studied in Fu and Han [FH95]. Methods for constraint-based association rule mining discussed in this chapter were studied by Ng, Lakshmanan, Han, and Pang [NLHP98], Lakshmanan, Ng, Han, and Pang [LNHP99], and Pei, Han, and Lakshmanan [PHL01]. An efficient method for mining constrained correlated sets was given in Grahne, Lakshmanan, and Wang [GLW00]. A dual mining approach was proposed by Bucila, Gehrke, Kifer, and White [BGKW03]. Other ideas involving the use of templates or predicate constraints in mining have been discussed in [AK93], [DT93], [HK91], [LHC97], [ST96], [SVA97].

The association mining language presented in this chapter was based on an extension of the data mining query language, DMQL, proposed in Han, Fu, Wang, et al. [HFW<sup>+</sup>96], by incorporation of the spirit of the SQL-like operator for mining single-dimensional association rules proposed by Meo, Psaila, and Ceri [MPC96]. MSQL, a query language for mining flexible association rules, was proposed by Imielinski and Virmani [IV99]. *OLE DB for Data Mining (DM)*, a data mining query language that includes association mining modules was proposed by Microsoft Corporation [Cor00].

## Chapter 6

# Classification and Prediction

Databases are rich with hidden information that can be used for intelligent decision making. Classification and prediction are two forms of data analysis that can be used to extract models describing important data classes or to predict future data trends. Such analysis can help provide us with a better understanding of the data at large. Whereas *classification* predicts categorical (discrete, unordered) labels, *prediction* models continuous-valued functions. For example, we can build a classification model to categorize bank loan applications as either safe or risky, or a prediction model to predict the expenditures in dollars of potential customers on computer equipment given their income and occupation. Many classification and prediction methods have been proposed by researchers in machine learning, pattern recognition, and statistics. Most algorithms are memory resident, typically assuming a small data size. Recent data mining research has built on such work, developing scalable classification and prediction techniques capable of handling large disk-resident data.

In this chapter, you will learn basic techniques for data classification such as how to build decision tree classifiers, Bayesian classifiers, Bayesian belief networks, and rule-based classifiers. Backpropagation (a neural network technique) is also discussed, in addition to a more recent approach to classification known as support vector machines. Classification based on association rule mining is explored. Other approaches to classification, such as *k*-nearest neighbor classifiers, case-based reasoning, genetic algorithms, rough sets, and fuzzy logic techniques are introduced. Methods for prediction, including linear regression, nonlinear regression, and other regression-based models, are briefly discussed. Where applicable, you will learn about extensions to these techniques for their application to classification and prediction in *large* databases. Classification and prediction have numerous applications including fraud detection, target marketing, performance prediction, manufacturing, and medical diagnosis.

### 6.1 What Is Classification? What Is Prediction?

A bank loans officer needs analysis of her data in order to learn which loan applicants are “safe” and which are “risky” for the bank. A marketing manager at *AllElectronics* needs data analysis to help guess whether or not a customer with a given profile will buy a new computer. A medical researcher wants to analyze breast cancer data in order to predict which one of three specific treatments a patient should receive. In each of these examples, the data analysis task is **classification**, where a model or **classifier** is constructed to predict *categorical labels*, such as “safe” or “risky” for the loan application data, “yes” or “no” for the marketing data, or “treatment A”, “treatment B”, or “treatment C” for the medical data. These categories can be represented by discrete values, where the ordering among values has no meaning. For example, the values 1, 2, and 3 may be used to represent treatments A, B, and C, above, where there is no ordering implied among this group of treatment regimes.

Suppose that the marketing manager above would like to predict how much a given customer will spend during a sale at *AllElectronics*. This data analysis task is an example of **numeric prediction**, where the model constructed predicts a *continuous-valued function*, or *ordered value*, as opposed to a categorical label. This model is a **predictor**. **Regression analysis** is a statistical methodology that is most often used for numeric prediction,

hence the two terms are often used synonymously. We do not treat the two terms as synonyms, however, since several other methods can be used for numeric prediction, as we shall see later in this chapter. Classification and numeric prediction are the two major types of **prediction problems**. For simplicity, when there is no ambiguity, we will use the shortened term of *prediction* to refer to *numeric prediction*.

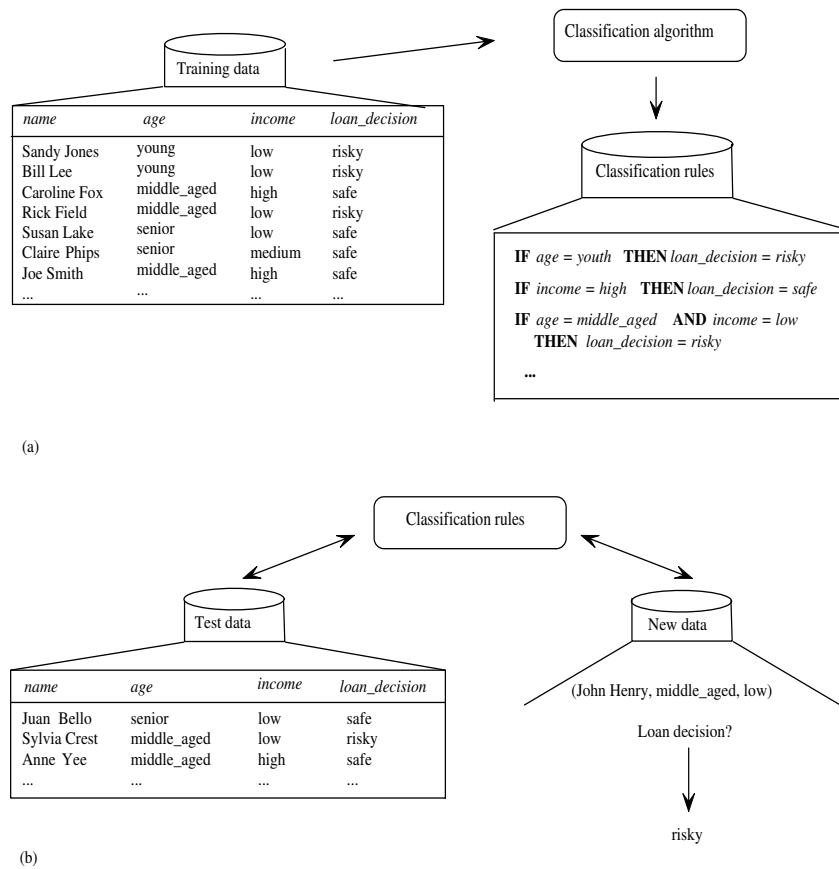


Figure 6.1: The data classification process: (a) *Learning*: Training data are analyzed by a classification algorithm. Here, the class label attribute is *loan\_decision*, and the learned model or classifier is represented in the form of classification rules. (b) *Classification*: Test data are used to estimate the accuracy of the classification rules. If the accuracy is considered acceptable, the rules can be applied to the classification of new data tuples.

*“How does classification work?”* **Data classification** is a two-step process, as shown for the loan application data of Figure 6.1. (The data are simplified for illustrative purposes. In reality, we may expect many more attributes to be considered.) In the first step, a classifier is built describing a predetermined set of data classes or concepts. This is the **learning** step (or training phase), where a classification algorithm builds the classifier by analyzing or “learning from” a **training set** made up of database tuples and their associated class labels. A tuple,  $\mathbf{X}$ , is represented by an  $n$ -dimensional **attribute vector**,  $\mathbf{X} = (x_1, x_2, \dots, x_n)$ , depicting  $n$  measurements made on the tuple from  $n$  database attributes, respectively,  $A_1, A_2, \dots, A_n$ .<sup>1</sup> Each tuple,  $\mathbf{X}$ , is assumed to belong to a predefined class as determined by another database attribute called the **class label attribute**. The class label attribute is discrete-valued and unordered. It is *categorical* in that each value serves as a category or class. The individual tuples making up the training set are referred to as **training tuples** and are selected from the database

<sup>1</sup>Each attribute represents a “feature” of  $\mathbf{X}$ . Hence, the pattern recognition literature uses the term *feature vector* rather than *attribute vector*. Since our discussion is from a database perspective, we propose the term “attribute vector”. In our notation, any variable representing a vector is shown in bold italic font; measurements depicting the vector are shown in italic font, e.g.,  $\mathbf{X} = (x_1, x_2, x_3)$ .

under analysis. In the context of classification, data tuples can be referred to as *samples*, *examples*, *instances*, *data points*, or *objects*.<sup>2</sup>

Since the class label of each training tuple *is provided*, this step is also known as **supervised learning** (i.e., the learning of the classifier is “supervised” in that it is told to which class each training tuple belongs). It contrasts with **unsupervised learning** (or **clustering**), in which the class label of each training tuple is not known, and the number or set of classes to be learned may not be known in advance. For example, if we did not have the *loan\_decision* data available for the training set, we could use clustering to try to determine “groups of like tuples”, which may correspond to risk groups within the loan application data. Clustering is the topic of Chapter 7.

This first step of the classification process can also be viewed as the learning of a mapping or function,  $y = f(\mathbf{X})$ , that can predict the associated class label  $y$  of a given tuple  $\mathbf{X}$ . In this view, we wish to learn a mapping or function that separates the data classes. Typically, this mapping is represented in the form of classification rules, decision trees, or mathematical formulae. In our example, the mapping is represented as classification rules that identify loan applications as being either safe or risky (Figure 6.1(a)). The rules can be used to categorize future data tuples, as well as provide deeper insight into the database contents. They also provide a compressed representation of the data.

“What about classification accuracy?” In the second step (Figure 6.1(b)), the model is used for classification. First, the predictive accuracy of the classifier is estimated. If we were to use the training set to measure the accuracy of the classifier, this estimate would likely be optimistic since the classifier tends to **overfit** the data (that is, during learning it may incorporate some particular anomalies of the training data that are not present in the general data set overall). Therefore, a **test set** is used, made up of **test tuples** and their associated class labels. These tuples are randomly selected from the general data set. They are independent of the training tuples, meaning that they are not used to construct the classifier.

The **accuracy** of a classifier on a given test set is the percentage of test set tuples that are correctly classified by the classifier. The associated class label of each test tuple is compared with the learned classifier’s class prediction for that tuple. Section 6.13 describes several methods for estimating classifier accuracy. If the accuracy of the classifier is considered acceptable, the classifier can be used to classify future data tuples for which the class label is not known. (Such data are also referred to in the machine learning literature as “*unknown*” or “*previously unseen*” data.) For example, the classification rules learned in Figure 6.1(a) from the analysis of data from previous loan applications can be used to approve or reject new or future loan applicants.

“How is (numeric) prediction different from classification?” Data prediction is a two-step process, similar to that of data classification as described in Figure 6.1. However, for prediction, we lose the terminology of “class label attribute” since the attribute for which values are being predicted is continuous-valued (ordered), rather than categorical (discrete-valued and unordered). The attribute can be referred to, simply, as the **predicted attribute**<sup>3</sup>. Suppose that, in our example, we instead wanted to predict the amount (in dollars) that would be “safe” for the bank to loan an applicant. The data mining task becomes prediction, rather than classification. We would replace the categorical attribute, *loan\_decision*, with the continuous-valued *loan\_amount* as the predicted attribute, and build a predictor for our task.

Note that prediction can also be viewed as a mapping or function,  $y = f(\mathbf{X})$ , where  $\mathbf{X}$  is the input (e.g., a tuple describing a loan applicant) and the output  $y$  is a continuous or ordered value (such as the predicted amount that the bank can safely loan the applicant). That is, we wish to learn a mapping or function that models the relationship between  $\mathbf{X}$  and  $y$ .

Prediction and classification also differ in the methods that are used to build their respective models. As with classification, the training set used to build a predictor should not be used to assess its accuracy. An independent test set should be used instead. The accuracy of a predictor is estimated by computing an error based on the difference between the predicted value and the actual known value of  $y$  for each of the test tuples,  $\mathbf{X}$ . There are various predictor error measures (Section 6.12.2). General methods for error estimation are discussed in Section 6.13.

<sup>2</sup>In the machine learning literature, training tuples are commonly referred to as *training samples*. Throughout this text, we prefer to use the term *tuples* instead of *samples* since we discuss the theme of classification from a database-oriented perspective.

<sup>3</sup>We could also use this term for classification, although, for that task, the term “class label attribute” is more descriptive.

## 6.2 Issues Regarding Classification and Prediction

This section describes issues regarding preprocessing the data for classification and prediction. Criteria for the comparison and evaluation of classification methods are also described.

### 6.2.1 Preparing the Data for Classification and Prediction

The following preprocessing steps may be applied to the data in order to help improve the accuracy, efficiency, and scalability of the classification or prediction process.

- **Data cleaning:** This refers to the preprocessing of data in order to remove or reduce *noise* (by applying smoothing techniques, for example), and the treatment of *missing values* (e.g., by replacing a missing value with the most commonly occurring value for that attribute, or with the most probable value based on statistics). Although most classification algorithms have some mechanisms for handling noisy or missing data, this step can help reduce confusion during learning.
- **Relevance analysis:** Many of the attributes in the data may be *redundant*. **Correlation analysis** can be used to identify whether any two given attributes are statistically related. For example, a strong correlation between attributes  $A_1$  and  $A_2$  would suggest that one of the two could be removed from further analysis. A database may also contain *irrelevant* attributes. **Attribute subset selection**<sup>4</sup> can be used in these cases to find a reduced set of attributes such that the resulting probability distribution of the data classes is as close as possible to the original distribution obtained using all attributes. Hence, relevance analysis, in the form of correlation analysis and attribute subset selection, can be used to detect attributes that do not contribute to the classification or prediction task. Including such attributes may otherwise slow down, and possibly mislead, the learning step.

Ideally, the time spent on relevance analysis, when added to the time spent on learning from the resulting “reduced” attribute (or feature) subset, should be less than the time that would have been spent on learning from the original set of attributes. Hence, such analysis can help improve classification efficiency and scalability.

- **Data transformation and reduction:** The data may be transformed by normalization, particularly when neural networks or methods involving distance measurements are used in the learning step. **Normalization** involves scaling all values for a given attribute so that they fall within a small specified range, such as  $-1.0$  to  $1.0$ , or  $0.0$  to  $1.0$ . In methods that use distance measurements, for example, this would prevent attributes with initially large ranges (like, say, *income*) from outweighing attributes with initially smaller ranges (such as binary attributes).

The data can also be transformed by *generalizing* it to higher-level concepts. Concept hierarchies may be used for this purpose. This is particularly useful for continuous-valued attributes. For example, numeric values for the attribute *income* can be generalized to discrete ranges such as *low*, *medium*, and *high*. Similarly, categorical attributes, like *street*, can be generalized to higher-level concepts, like *city*. Since generalization compresses the original training data, fewer input/output operations may be involved during learning.

Data can also be reduced by applying many other methods, ranging from wavelet transformation and principle components analysis, to discretization techniques such as binning, histogram analysis, and clustering.

Data cleaning, relevance analysis (in the form of correlation analysis and attribute subset selection), and data transformation are described in greater detail in Chapter 2 of this book.

### 6.2.2 Comparing Classification and Prediction Methods

Classification and prediction methods can be compared and evaluated according to the following criteria:

---

<sup>4</sup>In machine learning, this is known as *feature subset selection*.

- **Accuracy:** The **accuracy of a classifier** refers to the ability of a given classifier to correctly predict the class label of new or previously unseen data (i.e., tuples without class label information). Similarly, the **accuracy of a predictor** refers to how well a given predictor can guess the value of the predicted attribute for new or previously unseen data. Accuracy measures are given in Section 6.12. Accuracy can be estimated using one or more test sets that are independent of the training set. Estimation techniques, such as cross-validation and bootstrapping, are described in Section 6.13. Strategies for improving the accuracy of a model are given in Section 6.14. Since the accuracy computed is only an estimate of how well the classifier or predictor will do on new data tuples, confidence limits can be computed to help gauge this estimate. This is discussed in Section 6.15.
- **Speed:** This refers to the computational costs involved in generating and using the given classifier or predictor.
- **Robustness:** This is the ability of the classifier or predictor to make correct predictions given noisy data or data with missing values.
- **Scalability:** This refers to the ability to construct the classifier or predictor efficiently given large amounts of data.
- **Interpretability:** This refers to the level of understanding and insight that is provided by the classifier or predictor. Interpretability is subjective and therefore more difficult to assess. We discuss some work in this area, such as the extraction of classification rules from a “black box” neural network classifier called backpropagation (Section 6.6.4).

These issues are discussed throughout the chapter with respect to the various classification and prediction methods presented. Recent data mining research has contributed to the development of scalable algorithms for classification and prediction. Additional contributions include the exploration of mined “associations” between attributes and their use for effective classification. Model selection is discussed in Section 6.15.

## 6.3 Classification by Decision Tree Induction

**Decision tree induction** is the learning of decision trees from class-labeled training tuples. A **decision tree** is a flow-chart-like tree structure, where each **internal node** (non-leaf node) denotes a test on an attribute, each **branch** represents an outcome of the test, and each **leaf node** (or *terminal node*) holds a class label. The topmost node in a tree is the **root** node. A typical decision tree is shown in Figure 6.2. It represents the concept *buys\_computer*, that is, it predicts whether or not a customer at *AllElectronics* is likely to purchase a computer. Internal nodes are denoted by rectangles, and leaf nodes are denoted by ovals. Some decision tree algorithms produce only *binary* trees (where each internal node branches to exactly two other nodes), while others can produce non-binary trees.

*“How are decision trees used for classification?”* Given a tuple,  $\mathbf{X}$ , for which the associated class label is unknown, the attribute values of the tuple are tested against the decision tree. A path is traced from the root to a leaf node, which holds the class prediction for that tuple. Decision trees can easily be converted to classification rules.

*“Why are decision tree classifiers so popular?”* The construction of decision tree classifiers does not require any domain knowledge or parameter setting and therefore is appropriate for exploratory knowledge discovery. Decision trees can handle high dimensional data. Their representation of acquired knowledge in tree form is intuitive and generally easy to assimilate by humans. The learning and classification steps of decision tree induction are simple and fast. In general, decision tree classifiers have good accuracy. However, successful use may depend on the data at hand. Decision tree induction algorithms have been used for classification in many application areas, such as medicine, manufacturing and production, financial analysis, astronomy, and molecular biology. Decision trees are the basis of several commercial rule induction systems.

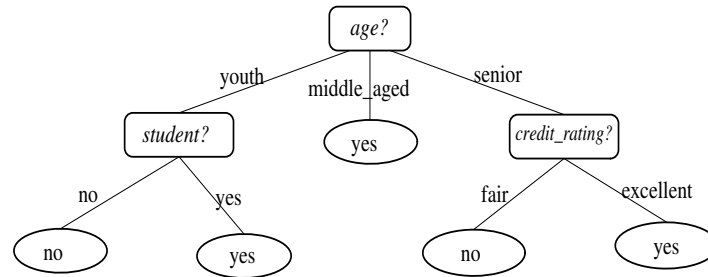


Figure 6.2: A decision tree for the concept *buys\_computer*, indicating whether or not a customer at *AllElectronics* is likely to purchase a computer. Each internal (nonleaf) node represents a test on an attribute. Each leaf node represents a class (either *buys\_computer* = *yes* or *buys\_computer* = *no*).

In Section 6.3.1, we describe a basic algorithm for learning decision trees. During tree construction, *attribute selection measures* are used to select the attribute that best partitions the tuples into distinct classes. Popular measures of attribute selection are given in Section 6.3.2. When decision trees are built, many of the branches may reflect noise or outliers in the training data. *Tree pruning* attempts to identify and remove such branches, with the goal of improving classification accuracy on unseen data. Tree pruning is described in Section 6.3.3. Scalability issues for the induction of decision trees from large databases are discussed in Section 6.3.4.

### 6.3.1 Decision Tree Induction

During the late 1970s and early 1980s, J. Ross Quinlan, a researcher in machine learning, developed a decision tree algorithm known as ID3 (Iterative Dichotomiser). This work expanded on earlier work on *concept learning systems*, described by E. B. Hunt, J. Marin, and P. T. Stone. Quinlan later presented C4.5 (a successor of ID3), which became a benchmark to which newer supervised learning algorithms are often compared. In 1984, a group of statisticians (L. Breiman, J. Friedman, R. Olshen, and C. Stone) published the book *Classification and Regression Trees* (CART), which described the generation of binary decision trees. ID3 and CART were invented independently of one another at around the same time, yet follow a similar approach for learning decision trees from training tuples. These two cornerstone algorithms spawned a flurry of work on decision tree induction.

ID3, C4.5, and CART adopt a greedy (i.e., non-backtracking) approach where decision trees are constructed in a top-down recursive divide-and-conquer manner. The vast majority of algorithms for decision tree induction also follow such a top-down approach, which starts with a training set of tuples and their associated class labels. The training set is recursively partitioned into smaller subsets as the tree is being built. A basic decision tree algorithm is summarized in Figure 6.3. At first glance, the algorithm may appear long, but fear not! It is quite straightforward. The strategy is as follows.

- The algorithm is called with three parameters: *D*, *attribute\_list*, and *Attribute\_selection\_method*. We refer to *D* as a data partition. Initially, it is the complete set of training tuples and their associated class labels. The parameter *attribute\_list* is a list of attributes describing the tuples. *Attribute\_selection\_method* specifies a heuristic procedure for selecting the attribute that “best” discriminates the given tuples according to class. This procedure employs an attribute selection measure, such as information gain or the gini index. Whether or not the tree is strictly binary is generally driven by the attribute selection measure. Some attribute selection measures, such as the gini index, enforce the resulting tree to be binary. Others, like information gain, do not, therein allowing multiway splits (i.e., two or more branches to be grown from a node).
- The tree starts as a single node, *N*, representing the training tuples in *D* (step 1).<sup>5</sup>

<sup>5</sup>The partition of class-labeled training tuples at node *N* is the set of tuples that follow a path from the root of the tree to node *N* when being processed by the tree. This set is sometimes referred to in the literature as the *family* of tuples at node *N*. We have



**Algorithm: Generate\_decision\_tree.** Generate a decision tree from the training tuples of data partition  $D$ .

**Input:**

- Data partition,  $D$ , which is a set of training tuples and their associated class labels;
- *attribute\_list*, the set of candidate attributes;
- *Attribute\_selection\_method*, a procedure to determine the splitting criterion that “best” partitions the data tuples into individual classes. This criterion consists of a *splitting\_attribute* and, possibly, either a *split point* or *splitting subset*.

**Output:** A decision tree.

**Method:**

```

(1) create a node  $N$ ;
(2) if tuples in  $D$  are all of the same class,  $C$  then
(3)     return  $N$  as a leaf node labeled with the class  $C$ ;
(4) if attribute_list is empty then
(5)     return  $N$  as a leaf node labeled with the majority class in  $D$ ; // majority voting
(6) apply Attribute_selection_method( $D$ , attribute_list) to find “best” splitting_criterion;
(7) label node  $N$  with splitting_criterion;
(8) if splitting_attribute is discrete-valued and
    multiway splits allowed then // not restricted to binary trees
(9)     attribute_list  $\leftarrow$  attribute_list  $-$  splitting_attribute; // remove splitting_attribute
(10) for each outcome  $j$  of splitting_criterion
    // partition the tuples and grow subtrees for each partition
(11)     let  $D_j$  be the set of data tuples in  $D$  satisfying outcome  $j$ ; // a partition
(12)     if  $D_j$  is empty then
(13)         attach a leaf labeled with the majority class in  $D$  to node  $N$ ;
(14)     else attach the node returned by Generate_decision_tree( $D_j$ , attribute_list) to node  $N$ ;
    endfor
(15) return  $N$ ;

```

Figure 6.3: Basic algorithm for inducing a decision tree from training tuples.

- If the tuples in  $D$  are all of the same class, then node  $N$  becomes a leaf and is labeled with that class (steps 2 and 3). Note that steps 4 and 5 are terminating conditions. All of the terminating conditions are explained at the end of the algorithm.
- Otherwise, the algorithm calls *Attribute\_selection\_method* to determine the **splitting criterion**. The splitting criterion tells us which attribute to test at node  $N$  by determining the “best” way to separate or partition the tuples in  $D$  into individual classes (step 6). The splitting criterion also tells us which branches to grow from node  $N$  with respect to the outcomes of the chosen test. More specifically, the splitting criterion indicates the **splitting attribute** and may also indicate either a **split point** or a **splitting subset**. The splitting criterion is determined so that, ideally, the resulting partitions at each branch are as “pure” as possible. A partition is **pure** if all of the tuples in it belong to the same class. In other words, if we were to split up the tuples in  $D$  according to the mutually exclusive outcomes of the splitting criterion, we hope for the resulting partitions to be as pure as possible.
- The node  $N$  is labeled with the splitting criterion, which serves as a test at the node (step 7). A branch is grown from node  $N$  for each of the outcomes of the splitting criterion. The tuples in  $D$  are partitioned accordingly (steps 10-11). There are three possible scenarios, as illustrated in Figure 6.4. Let  $A$  be the splitting attribute.  $A$  has  $v$  distinct values,  $\{a_1, a_2, \dots, a_v\}$ , based on the training data.

1. *A is discrete-valued*: In this case, the outcomes of the test at node  $N$  correspond directly to the known

---

referred to this set as the “tuples represented at node  $N$ ”, “the tuples that reach node  $N$ ”, or simply “the tuples at node  $N$ ”. Rather than storing the actual tuples at a node, most implementations store pointers to these tuples.

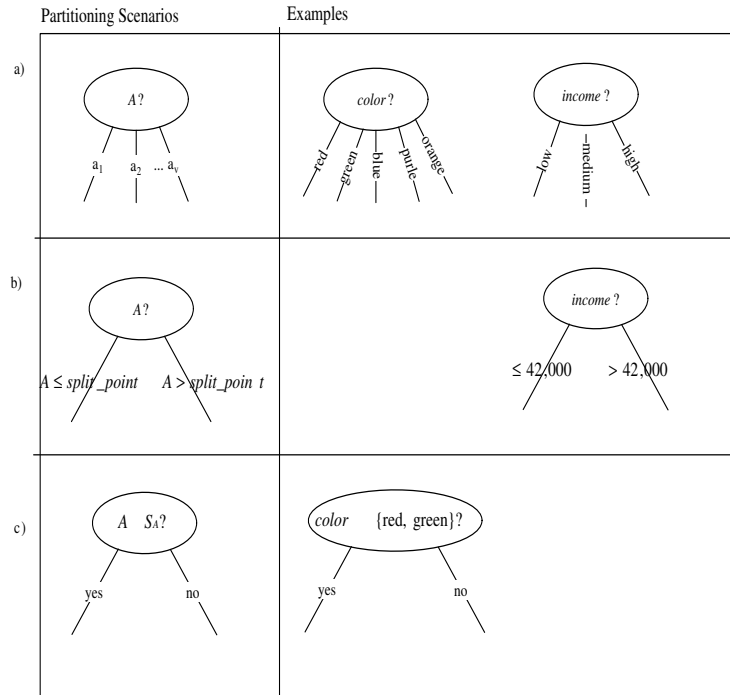


Figure 6.4: Three possibilities for partitioning tuples based on the splitting criterion, shown with examples. Let  $A$  be the splitting attribute. (a) If  $A$  is discrete-valued then one branch is grown for each known value of  $A$ . (b) If  $A$  is continuous-valued then two branches are grown, corresponding to  $A \leq \text{split\_point}$  and  $A > \text{split\_point}$ . (c) If  $A$  is discrete-valued and a binary tree must be produced then the test is of the form  $A \in S_A$ , where  $S_A$  is the splitting subset for  $A$ . [TO EDITOR The symbol  $\in$  somehow is not appearing in  $A \in S_A$ . Please fix! Also, two lines from part (a) show up on screen but not in printout. Please check. Thanks!]

values of  $A$ . A branch is created for each known value,  $a_j$ , of  $A$  and labeled with that value (Figure 6.4a)). Partition  $D_j$  is the subset of class-labeled tuples in  $D$  having value  $a_j$  of  $A$ . Since all of the tuples in a given partition have the same value for  $A$ , then  $A$  need not be considered in any future partitioning of the tuples. Therefore, it is removed from *attribute\_list* (steps 8-9).

2. *A is continuous-valued*: In this case, the test at node  $N$  has two possible outcomes, corresponding to the conditions  $A \leq \text{split\_point}$  and  $A > \text{split\_point}$ , respectively, where *split\_point* is the split point returned by *Attribute\_selection\_method* as part of the splitting criterion. (In practice, the split point,  $a$ , is often taken as the midpoint of two known adjacent values of  $A$  and therefore may not actually be a pre-existing value of  $A$  from the training data.) Two branches are grown from  $N$  and labeled according to the above outcomes (Figure 6.4b)). The tuples are partitioned such that  $D_1$  holds the subset of class-labeled tuples in  $D$  for which  $A \leq \text{split\_point}$ , while  $D_2$  holds the rest.
  3. *A is discrete-valued* and a *binary tree* must be produced (as dictated by the attribute selection measure or algorithm being used): The test at node  $N$  is of the form “ $A \in S_A$ ?”.  $S_A$  is the splitting subset for  $A$ , returned by *Attribute\_selection\_method* as part of the splitting criterion. It is a subset of the known values of  $A$ . If a given tuple has value  $a_j$  of  $A$  and if  $a_j \in S_A$ , then the test at node  $N$  is satisfied. Two branches are grown from  $N$  (Figure 6.4c)). By convention, the left branch out of  $N$  is labeled *yes* so that  $D_1$  corresponds to the subset of class-labeled tuples in  $D$  that satisfy the test. The right branch out of  $N$  is labeled *no* so that  $D_2$  corresponds to the subset of class-labeled tuples from  $D$  that do not satisfy the test.
- The algorithm uses the same process recursively to form a decision tree for the tuples at each resulting partition,  $D_j$ , of  $D$  (step 14).

- The recursive partitioning stops only when any one of the following terminating conditions is true:
  1. All of the tuples in partition  $D$  (represented at node  $N$ ) belong to the same class (steps 2 and 3), or
  2. There are no remaining attributes on which the tuples may be further partitioned (step 4). In this case, **majority voting** is employed (step 5). This involves converting node  $N$  into a leaf and labelling it with the most common class in  $D$ . Alternatively, the class distribution of the node tuples may be stored.
  3. There are no tuples for a given branch, i.e., a partition  $D_j$  is empty (step 12). In this case, a leaf is created with the majority class in  $D$  (step 13).
- The resulting decision tree is returned (step 15).

The computational complexity of the algorithm given training set  $D$  is  $O(n \times |D| \times \log(|D|))$ , where  $n$  is the number of attributes describing the tuples in  $D$  and  $|D|$  is the number of training tuples in  $D$ . This means that the computational cost of growing a tree grows at most  $n \times |D| \times \log(|D|)$  with  $|D|$  tuples. The proof is left as an exercise.

**Incremental** versions of decision tree induction have also been proposed. When given new training data, these restructure the decision tree acquired from learning on previous training data, rather than relearning a new tree from scratch.

Differences in decision tree algorithms include how the attributes are selected in creating the tree (Section 6.3.2) and the mechanisms used for pruning (Section 6.3.3). The basic algorithm described above requires one pass over the training tuples in  $D$  for each level of the tree. This can lead to long training times and lack of available memory when dealing with large databases. Improvements regarding the scalability of decision tree induction are discussed in Section 6.3.4. A discussion of strategies for extracting rules from decision trees is given in Section 6.5.2 regarding rule-based classification.

### 6.3.2 Attribute Selection Measures

An **attribute selection measure** is a heuristic for selecting the splitting criterion that “best” separates a given data partition,  $D$ , of class-labeled training tuples into individual classes. If we were to split  $D$  into smaller partitions according to the outcomes of the splitting criterion, ideally each partition would be pure (where all of the tuples that fall into a given partition would belong to the same class). Conceptually, the “best” splitting criterion is the one that most closely results in such a scenario. Attribute selection measures are also known as **splitting rules** since they determine how the tuples at a given node are to be split. The attribute selection measure provides a ranking for each attribute describing the given training tuples. The attribute having the best score for the measure<sup>6</sup> is chosen as the *splitting attribute* for the given tuples. If the splitting attribute is continuous-valued or if we are restricted to binary trees then, respectively, either a *split point* or a *splitting subset* must also be determined as part of the splitting criterion. The tree node created for partition  $D$  is labeled with the splitting criterion, branches are grown for each outcome of the criterion, and the tuples are partitioned accordingly. This section describes three popular attribute selection measures—*information gain*, *gain ratio*, and *gini index*.

The notation used herein is as follows. Let  $D$ , the data partition, be a training set of class-labeled tuples. Suppose the class label attribute has  $m$  distinct values defining  $m$  distinct classes,  $C_i$  (for  $i = 1, \dots, m$ ). Let  $C_{i,D}$  be the set of tuples of class  $C_i$  in  $D$ . Let  $|D|$  and  $|C_{i,D}|$  denote the number of tuples in  $D$  and  $C_{i,D}$ , respectively.

#### Information gain

ID3 uses **information gain** as its attribute selection measure. This measure is based on pioneering work by Claude Shannon on information theory, which studied the value or “information content” of messages. Let node  $N$  represent or hold the tuples of partition  $D$ . The attribute with the highest information gain is chosen as

---

<sup>6</sup>Depending on the measure, either the highest or lowest score is chosen as the best, i.e., some measures strive to maximize while others strive to minimize.

the splitting attribute for node  $N$ . This attribute minimizes the information needed to classify the tuples in the resulting partitions and reflects the least randomness or “impurity” in these partitions. Such an approach minimizes the expected number of tests needed to classify a given tuple and guarantees that a simple (but not necessarily the simplest) tree is found.

The expected information needed to classify a tuple in  $D$  is given by

$$Info(D) = - \sum_{i=1}^m p_i \log_2(p_i), \quad (6.1)$$

where  $p_i$  is the probability that an arbitrary tuple in  $D$  belongs to class  $C_i$  and is estimated by  $|C_{i,D}|/|D|$ . A log function to the base 2 is used since the information is encoded in bits.  $Info(D)$  is just the average amount of information needed to identify the class label of a tuple in  $D$ . Note that, at this point, the information we have is based solely on the proportions of tuples of each class.  $Info(D)$  is also known as the **entropy** of  $D$ .

Now, suppose we were to partition the tuples in  $D$  on some attribute  $A$  having  $v$  distinct values,  $\{a_1, a_2, \dots, a_v\}$ , as observed from the training data. If  $A$  is discrete-valued, these values correspond directly to the  $v$  outcomes of a test on  $A$ . Attribute  $A$  can be used to split  $D$  into  $v$  partitions or subsets,  $\{D_1, D_2, \dots, D_v\}$ , where  $D_j$  contains those tuples in  $D$  that have outcome  $a_j$  of  $A$ . These partitions would correspond to the branches grown from node  $N$ . Ideally, we would like this partitioning to produce an exact classification of the tuples. That is, we would like for each partition to be pure. However, it is quite likely that the partitions will be impure, e.g., where a partition may contain a collection of tuples from different classes rather than from a single class. How much more information would we still need (after the partitioning) in order to arrive at an exact classification? This amount is measured by

$$Info_A(D) = \sum_{j=1}^v \frac{|D_j|}{|D|} \times Info(D_j). \quad (6.2)$$

The term  $\frac{|D_j|}{|D|}$  acts as the weight of the  $j$ th partition.  $Info_A(D)$  is the expected information required to classify a tuple from  $D$  based on the partitioning by  $A$ . The smaller the expected information (still) required, the greater the purity of the partitions.

Information gain is defined as the difference between the original information requirement (i.e., based on just the proportion of classes) and the new requirement (i.e., obtained after partitioning on  $A$ ). That is,

$$Gain(A) = Info(D) - Info_A(D). \quad (6.3)$$

In other words,  $Gain(A)$  tells us how much would be gained by branching on  $A$ . It is the expected reduction in the information requirement caused by knowing the value of  $A$ . The attribute  $A$  with the highest information gain ( $Gain(A)$ ), is chosen as the splitting attribute at node  $N$ . This is equivalent to saying that we want to partition on the attribute  $A$  that would do the “best classification”, so that the amount of information still required to finish classifying the tuples is minimal (i.e., minimum  $Info_A(D)$ ).

**Example 6.1 Induction of a decision tree using information gain.** Table 6.1 presents a training set,  $D$ , of class-labeled tuples randomly selected from the *Allelectronics* customer database. (The data are adapted from [Qui86]. In this example, each attribute is discrete-valued. Continuous-valued attributes have been generalized.) The class label attribute, *buys\_computer*, has two distinct values (namely,  $\{yes, no\}$ ); therefore, there are two distinct classes (that is,  $m = 2$ ). Let class  $C_1$  correspond to *yes* and class  $C_2$  correspond to *no*. There are 9 tuples of class *yes* and 5 tuples of class *no*. A (root) node  $N$  is created for the tuples in  $D$ . To find the splitting criterion for these tuples, we must compute the information gain of each attribute. We first use Equation (6.1) to compute the expected information needed to classify a tuple in  $D$ :

$$Info(D) = - \frac{9}{14} \log_2\left(\frac{9}{14}\right) - \frac{5}{14} \log_2\left(\frac{5}{14}\right) = 0.940 \text{ bits.}$$

Next, we need to compute the expected information requirement for each attribute. Let's start with the attribute *age*. We need to look at the distribution of *yes* and *no* tuples for each category of *age*. For the *age*

<i>RID</i>	<i>age</i>	<i>income</i>	<i>student</i>	<i>credit_rating</i>	<i>Class: buys_computer</i>
1	youth	high	no	fair	no
2	youth	high	no	excellent	no
3	middle_aged	high	no	fair	yes
4	senior	medium	no	fair	yes
5	senior	low	yes	fair	yes
6	senior	low	yes	excellent	no
7	middle_aged	low	yes	excellent	yes
8	youth	medium	no	fair	no
9	youth	low	yes	fair	yes
10	senior	medium	yes	fair	yes
11	youth	medium	yes	excellent	yes
12	middle_aged	medium	no	excellent	yes
13	middle_aged	high	yes	fair	yes
14	senior	medium	no	excellent	no

Table 6.1: Class-labeled training tuples from the *AllElectronics* customer database.

category “youth”, there are 2 *yes* tuples and 3 *no* tuples. For the category “middle\_aged”, there are 4 *yes* tuples and 0 *no* tuples. For the category “senior”, there are 3 *yes* tuples and 2 *no* tuples. Using Equation (6.2), the expected information needed to classify a tuple in  $D$  if the tuples are partitioned according to *age* is

$$\begin{aligned}
 Info_{age}(D) &= \frac{5}{14} \times \left( -\frac{2}{5} \log_2 \frac{2}{5} - \frac{3}{5} \log_2 \frac{3}{5} \right) \\
 &\quad + \frac{4}{14} \times \left( -\frac{4}{4} \log_2 \frac{4}{4} - \frac{0}{4} \log_2 \frac{0}{4} \right) \\
 &\quad + \frac{5}{14} \times \left( -\frac{3}{5} \log_2 \frac{3}{5} - \frac{2}{5} \log_2 \frac{2}{5} \right) \\
 &= 0.694 \text{ bits.}
 \end{aligned}$$

Hence, the gain in information from such a partitioning would be

$$Gain(age) = Info(D) - Info_{age}(D) = 0.940 - 0.694 = 0.246 \text{ bits.}$$

Similarly, we can compute  $Gain(income) = 0.029$  bits,  $Gain(student) = 0.151$  bits, and  $Gain(credit\_rating) = 0.048$  bits. Since *age* has the highest information gain among the attributes, it is selected as the splitting attribute. Node  $N$  is labeled with *age*, and branches are grown for each of the attribute’s values. The tuples are then partitioned accordingly, as shown in Figure 6.5. Notice that the tuples falling into the partition for *age* = *middle\_aged* all belong to the same class. Since they all belong to class “yes”, a leaf should therefore be created at the end of this branch and labeled with “yes”. The final decision tree returned by the algorithm is shown in Figure 6.2. ■

“But how can we compute the information gain of an attribute that is continuous-valued, unlike above?” Suppose, instead, that we have an attribute  $A$  that is continuous-valued, rather than discrete-valued. (For example, suppose that instead of the discretized version of *age* above, we instead have the raw values for this attribute.) For such a scenario, we must determine the “best” **split point** for  $A$ , where the split point is a threshold on  $A$ . We first sort the values of  $A$  in increasing order. Typically, the midpoint between each pair of adjacent values is considered as a possible split point. Therefore, given  $v$  values of  $A$ , then  $v - 1$  possible splits are evaluated. For example, the midpoint between the values  $a_i$  and  $a_{i+1}$  of  $A$  is

$$\frac{a_i + a_{i+1}}{2}. \quad (6.4)$$

If the values of  $A$  are sorted in advance, then determining the best split for  $A$  requires only one pass through the values. For each possible split point for  $A$ , we evaluate  $Info_A(D)$ , where the number of partitions is two, i.e.,

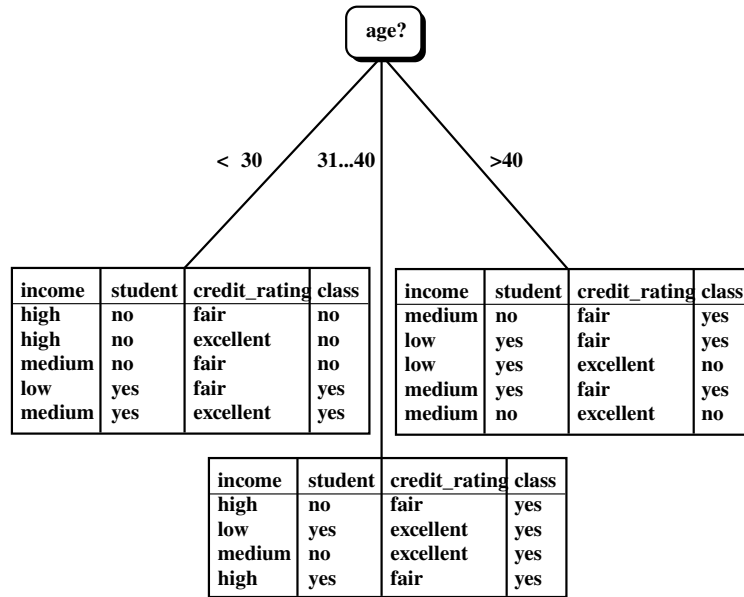


Figure 6.5: The attribute *age* has the highest information gain and therefore becomes the splitting attribute at the root node of the decision tree. Branches are grown for each outcome of *age*. The tuples are shown partitioned accordingly. [TO EDITOR Please replace “< 30” with *youth*, “31...40” with *middle\_aged*, and “> 40” with *senior*. Please italicize *age*, *income*, *student*, *credit\_rating*, and *class*. If possible please organize the three branches (tables) so that they all line up at the same level. Note, some rows from middle branch may not be showing up on printout. Please verify with actual figure file. ]

$v = 2$  (or  $j = 1, 2$ ) in Equation (6.2). The point with the minimum expected information requirement for  $A$  is selected as the *split\_point* for  $A$ .  $D_1$  is the set of tuples in  $D$  satisfying  $A \leq \text{split\_point}$  and  $D_2$  is the set of tuples in  $D$  satisfying  $A > \text{split\_point}$ .

### Gain ratio

The information gain measure is biased towards tests with many outcomes. That is, it prefers to select attributes having a large number of values. For example, consider an attribute that acts as a unique identifier, such as *product\_ID*. A split on *product\_ID* would result in a large number of partitions (as many as there are values), each one containing just one tuple. Since each partition is pure, the information required to classify data set  $D$  based on this partitioning would be  $\text{Info}_{\text{product\_ID}}(D) = 0$ . Therefore, the information gained by partitioning on this attribute is maximal. Clearly, such a partitioning is useless for classification.

C4.5, a successor of ID3, uses an extension to information gain known as *gain ratio*, which attempts to overcome this bias. It applies a kind of normalization to information gain using a “split information” value defined analogously with  $\text{Info}(D)$  as

$$\text{SplitInfo}_A(D) = - \sum_{j=1}^v \frac{|D_j|}{|D|} \times \log_2 \left( \frac{|D_j|}{|D|} \right). \quad (6.5)$$

This value represents the potential information generated by splitting the training data set,  $D$ , into  $v$  partitions, corresponding to the  $v$  outcomes of a test on attribute  $A$ . Note that, for each outcome, it considers the number of tuples having that outcome with respect to the total number of tuples in  $D$ . It differs from information gain,

which measures the information with respect to classification that is acquired based on the same partitioning. The gain ratio is defined as

$$\text{GainRatio}(A) = \frac{\text{Gain}(A)}{\text{SplitInfo}(A)}. \quad (6.6)$$

The attribute with the maximum gain ratio is selected as the splitting attribute. Note, however, that as the split information approaches 0, the ratio becomes unstable. A constraint is added to avoid this, whereby the information gain of the test selected must be large—at least as great as the average gain over all tests examined.

**Example 6.2 Computation of gain ratio for the attribute *income*.** A test on *income* splits the data of Table 6.1 into three partitions, namely *low*, *medium*, and *high*, containing 4, 6, and 4 tuples, respectively. To compute the gain ratio of *income*, we first use Equation (6.5) to obtain

$$\begin{aligned} \text{SplitInfo}_A(D) &= -\frac{4}{14} \times \log_2\left(\frac{4}{14}\right) - \frac{6}{14} \times \log_2\left(\frac{6}{14}\right) - \frac{4}{14} \times \log_2\left(\frac{4}{14}\right) \\ &= 0.926. \end{aligned}$$

From Example 6.1, we have  $\text{Gain}(\text{income}) = 0.029$ . Therefore,  $\text{GainRatio}(\text{income}) = 0.029/0.926 = 0.013$ . ■

### Gini index

The gini index is used in CART. Using the notation described above, the gini index measures the impurity of  $D$ , a data partition or set of training tuples, as

$$\text{Gini}(D) = 1 - \sum_{i=1}^m p_i^2, \quad (6.7)$$

where  $p_i$  is the probability that a tuple in  $D$  belongs to class  $C_i$  and is estimated by  $|C_{i,D}|/|D|$ . The sum is computed over  $m$  classes.

The gini index considers a binary split for each attribute. Let's first consider the case where  $A$  is a discrete-valued attribute having  $v$  distinct values,  $\{a_1, a_2, \dots, a_v\}$ , occurring in  $D$ . To determine the best binary split on  $A$ , we examine all of the possible subsets that can be formed using known values of  $A$ . Each subset,  $S_A$ , can be considered as a binary test for attribute  $A$  of the form " $A \in S_A$ ". Given a tuple, this test is satisfied if the value of  $A$  for the tuple is among the values listed in  $S_A$ . If  $A$  has  $v$  possible values then there are  $2^v$  possible subsets. For example, if *income* has three possible values, namely  $\{\text{low}, \text{medium}, \text{high}\}$ , then the possible subsets are  $\{\text{low}, \text{medium}, \text{high}\}$ ,  $\{\text{low}, \text{medium}\}$ ,  $\{\text{low}, \text{high}\}$ ,  $\{\text{medium}, \text{high}\}$ ,  $\{\text{low}\}$ ,  $\{\text{medium}\}$ ,  $\{\text{high}\}$ , and  $\{\}$ . We exclude the power set,  $\{\text{low}, \text{medium}, \text{high}\}$ , and the empty set from consideration since, conceptually, they do not represent a split. Therefore, there are  $2^v - 2$  possible ways to form two partitions of the data,  $D$ , based on a binary split on  $A$ .

When considering a binary split, we compute a weighted sum of the impurity of each resulting partition. For example, if a binary split on  $A$  partitions  $D$  into  $D_1$  and  $D_2$ , the gini index of  $D$  given that partitioning is

$$\text{Gini}_A(D) = \frac{|D_1|}{|D|} \text{Gini}(D_1) + \frac{|D_2|}{|D|} \text{Gini}(D_2). \quad (6.8)$$

For each attribute, each of the possible binary splits is considered. For a discrete-valued attribute, the subset that gives the minimum gini index for that attribute is selected as its splitting subset.

For continuous-valued attributes, each possible split point must be considered. The strategy is similar to that described above for information gain, where the midpoint between each pair of (sorted) adjacent values is taken

as a possible split point. The point giving the minimum gini index for a given (continuous-valued) attribute is taken as the split point of that attribute. Recall that for a possible split point of  $A$ ,  $D_1$  is the set of tuples in  $D$  satisfying  $A \leq \text{split\_point}$  and  $D_2$  is the set of tuples in  $D$  satisfying  $A > \text{split\_point}$ .

The reduction in impurity that would be incurred by a binary split on a discrete- or continuous-valued attribute  $A$  is

$$\Delta Gini(A) = Gini(D) - Gini_A(D). \quad (6.9)$$

The attribute that maximizes the reduction in impurity (or, equivalently, has the minimum gini index) is selected as the splitting attribute. This attribute and either its splitting subset (for a discrete-valued splitting attribute) or split point (for a continuous-valued splitting attribute) together form the splitting criterion.

**Example 6.3 Induction of a decision tree using gini index.** Let  $D$  be the training data of Table 6.1 where there are 9 tuples belonging to the class *buys\_computer = yes* and the remaining 5 tuples belong to the class *buys\_computer = no*. A (root) node  $N$  is created for the tuples in  $D$ . We first use Equation (6.7) for gini index to compute the impurity of  $D$ :

$$Gini(D) = 1 - \left(\frac{9}{14}\right)^2 - \left(\frac{5}{14}\right)^2 = 0.459.$$

To find the splitting criterion for the tuples in  $D$ , we need to compute the gini index for each attribute. Let's start with the attribute *income* and consider each of the possible splitting subsets. Consider the subset  $\{low, medium\}$ . This would result in 10 tuples in partition  $D_1$  satisfying the condition "*income*  $\in \{low, medium\}$ ". The remaining 4 tuples of  $D$  would be assigned to partition  $D_2$ . The gini index value computed based on this partitioning is

$$\begin{aligned} Gini_{income \in \{low, medium\}}(D) &= \frac{10}{14}Gini(D_1) + \frac{4}{14}Gini(D_2) \\ &= \frac{10}{14}\left(1 - \left(\frac{6}{10}\right)^2 - \left(\frac{4}{10}\right)^2\right) + \frac{4}{14}\left(1 - \left(\frac{1}{4}\right)^2 - \left(\frac{3}{4}\right)^2\right) \\ &= 0.450 \\ &= Gini_{income \in \{high\}}(D) \end{aligned}$$

Similarly, the gini index values for splits on the remaining subsets are: 0.315 (for the subsets  $\{low, high\}$  and  $\{medium\}$ ) and 0.300 (for the subsets  $\{medium, high\}$  and  $\{low\}$ ). Therefore, the best binary split for attribute *income* is on  $\{medium, high\}$  (or  $\{low\}$ ) because it minimizes the gini index. Evaluating the attribute, we obtain  $\{youth, senior\}$  (or  $\{middle\_aged\}$ ) as the best split for *age* with a gini index of 0.375; the attributes  $\{student\}$  and  $\{credit\_rating\}$  are both binary, with gini index values of 0.367 and 0.429, respectively.

The attribute *income* and splitting subset  $\{medium, high\}$  therefore give the minimum gini index overall, with a reduction in impurity of  $0.459 - 0.300 = 0.159$ . The binary split "*income*  $\in \{medium, high\}$ " results in the maximum reduction in impurity of the tuples in  $D$  and is returned as the splitting criterion. Node  $N$  is labeled with the criterion, two branches are grown from it, and the tuples are partitioned accordingly. Hence, the gini index has selected *income* instead of *age* at the root node, unlike the (non-binary) tree created by information gain (Example 6.1). ■

This section on attribute selection measures was not intended to be exhaustive. We have shown three measures that are commonly used for building decision trees. These measures are not without their biases. Information gain, as we saw, is biased towards multivalued attributes. Although the gain ratio adjusts for this bias, it tends to prefer unbalanced splits in which one partition is much smaller than the others. The gini index is biased towards multivalued attributes and has difficulty when the number of classes is large. It also tends to favor tests that result in equal-sized partitions and purity in both partitions. Although biased, these measures give reasonably good results in practice.

Many other attribute selection measures have been proposed. CHAID, a decision tree algorithm that is popular in marketing, uses an attribute selection measure that is based on the statistical  $\chi^2$  test for independence. Other



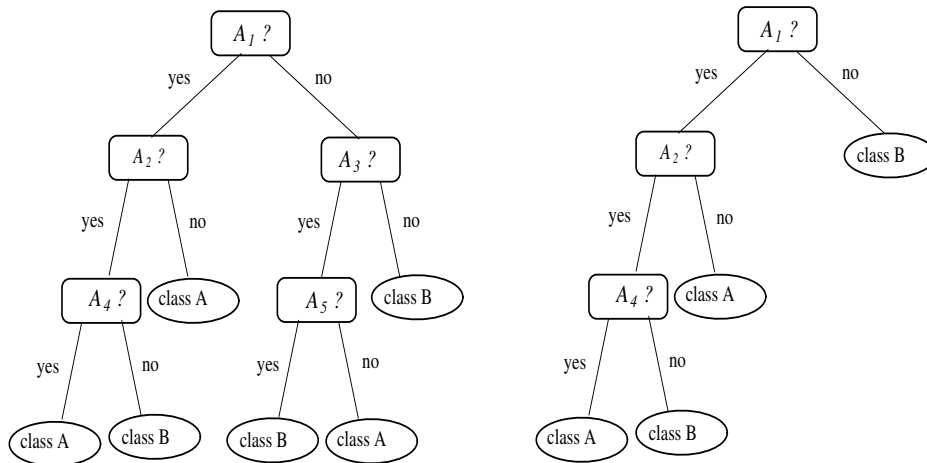


Figure 6.6: An unpruned decision tree, and a pruned version of it.

measures include C-SEP (which performs better than information gain and gini index in certain cases) and G-statistic (an information theoretic measure that is a close approximation to  $\chi^2$  distribution).

Attribute selection measures based on the **Minimum Description Length (MDL)** principle have the least bias toward multivalued attributes. MDL-based measures use encoding techniques to define the “best” decision tree as the one that requires the fewest number of bits to both (1) encode the tree and (2) encode the exceptions to the tree (i.e., cases that are not correctly classified by the tree). Its main idea is that the simplest of solutions is preferred.

Other attribute selection measures consider **multivariate splits**, i.e., where the partitioning of tuples is based on a *combination* of attributes, rather than on a single attribute. The CART system, for example, can find multivariate splits based on a linear combination of attributes. Multivariate splits are a form of **attribute** (or feature) **construction**, where new attributes are created based on the existing ones. (Attribute construction is also discussed in Chapter 2, as a form of data transformation.) These other measures mentioned here are beyond the scope of this book. Additional references are given in the Bibliographic Notes at the end of this chapter.

“Which attribute selection measure is the best?” All measures have some bias. It has been shown that the time complexity of decision tree induction generally increases exponentially with tree height. Hence, measures that tend to produce shallower trees (e.g., with multiway rather than binary splits, and that favor more balanced splits) may be preferred. However, some studies have found that shallow trees tend to have a large number of leaves and higher error rates. In spite of several comparative studies, no one attribute selection measure has been found to be significantly superior to others. Most measures give quite good results.

### 6.3.3 Tree Pruning

When a decision tree is built, many of the branches will reflect anomalies in the training data due to noise or outliers. Tree pruning methods address this problem of *overfitting* the data. Such methods typically use statistical measures to remove the least reliable branches. An unpruned tree and a pruned version of it are shown in Figure 6.6. Pruned trees tend to be smaller and less complex and, thus, easier to comprehend. They are usually faster and better at correctly classifying independent test data (i.e., of previously unseen tuples) than unpruned trees.

“How does tree pruning work?” There are two common approaches to tree pruning: *prepruning* and *postpruning*.

In the **prepruning** approach, a tree is “pruned” by halting its construction early (e.g., by deciding not to further split or partition the subset of training tuples at a given node). Upon halting, the node becomes a leaf. The leaf may hold the most frequent class among the subset tuples or the probability distribution of those tuples.

When constructing a tree, measures such as statistical significance, information gain, gini index, and so on, can be used to assess the goodness of a split. If partitioning the tuples at a node would result in a split that falls below a prespecified threshold, then further partitioning of the given subset is halted. There are difficulties, however, in choosing an appropriate threshold. High thresholds could result in oversimplified trees, while low thresholds could result in very little simplification.

The second and more common approach is **postpruning**, which removes subtrees from a “fully grown” tree. A subtree at a given node is pruned by removing its branches and replacing it with a leaf. The leaf is labeled with the most frequent class among the subtree being replaced. For example, notice the subtree at node “ $A_3$ ?” in the unpruned tree of Figure 6.6. Suppose that the most common class within this subtree is “*class B*”. In the pruned version of the tree, the subtree in question is pruned by replacing it with the leaf “*class B*”.

The **cost complexity** pruning algorithm used in CART is an example of the postpruning approach. This approach considers the cost complexity of a tree to be a function of the number of leaves in the tree and the error rate of the tree (where the **error rate** is the percentage of tuples misclassified by the tree). It starts from the bottom of the tree. For each internal node,  $N$ , it computes the cost complexity of the subtree at  $N$ , and the cost complexity of the subtree at  $N$  if it were to be pruned (i.e., replaced by a leaf node). The two values are compared. If pruning the subtree at node  $N$  would result in a smaller cost complexity, then the subtree is pruned. Otherwise, it is kept. A **pruning set** of class-labeled tuples is used to estimate cost complexity. This set is independent of the training set used to build the unpruned tree and of any test set used for accuracy estimation. The algorithm generates a set of progressively pruned trees. In general, the smallest decision tree that minimizes the cost complexity is preferred.

C4.5 uses a method called **pessimistic pruning**, which is similar to the cost complexity method in that it also uses error rate estimates to make decisions regarding subtree pruning. Pessimistic pruning, however, does not require the use of a prune set. Instead, it uses the training set to estimate error rates. Recall that an estimate of accuracy or error based on the training set is overly optimistic and, therefore, strongly biased. The pessimistic pruning method therefore adjusts the error rates obtained from the training set by adding a penalty, so as to counter the bias incurred.

Rather than pruning trees based on estimated error rates, we can prune trees based on the number of bits required to encode them. The “best” pruned tree is the one that minimizes the number of encoding bits. This method adopts the Minimum Description Length (MDL) principle, which was briefly introduced in Section 6.3.2. The basic idea is that the simplest solution is preferred. Unlike cost complexity pruning, it does not require an independent set of tuples.

Alternatively, prepruning and postpruning may be interleaved for a combined approach. Postpruning requires more computation than prepruning, yet generally leads to a more reliable tree. No single pruning method has been found to be superior over all others. While some pruning methods do depend on the availability of additional data for pruning, this is usually not a concern when dealing with large databases.

Although pruned trees tend to be more compact than their unpruned counterparts, they may still be rather large and complex. Decision trees can suffer from *repetition* and *replication* (Figure 6.7), making them overwhelming to interpret. **Repetition** occurs when an attribute is repeatedly tested along a given branch of the tree (such as “*age < 60?*”, followed by “*age < 45?*”, and so on). In **replication**, duplicate subtrees exist within the tree. These situations can impede the accuracy and comprehensibility of a decision tree. The use of multivariate splits (splits based on a combination of attributes) can prevent these problems. Another approach is to use a different form of knowledge representation, such as rules, instead of decision trees. This is described in Section 6.5.2, which shows how a *rule-based classifier* can be constructed by extracting IF-THEN rules from a decision tree.

### 6.3.4 Scalability and Decision Tree Induction

“What if  $D$ , the disk-resident training set of class-labeled tuples, does not fit in memory? In other words, how scalable is decision tree induction?” The efficiency of existing decision tree algorithms, such as ID3, C4.5, and CART, has been well established for relatively small data sets. Efficiency becomes an issue of concern when these algorithms are applied to the mining of very large real-world databases. The pioneering decision tree algorithms

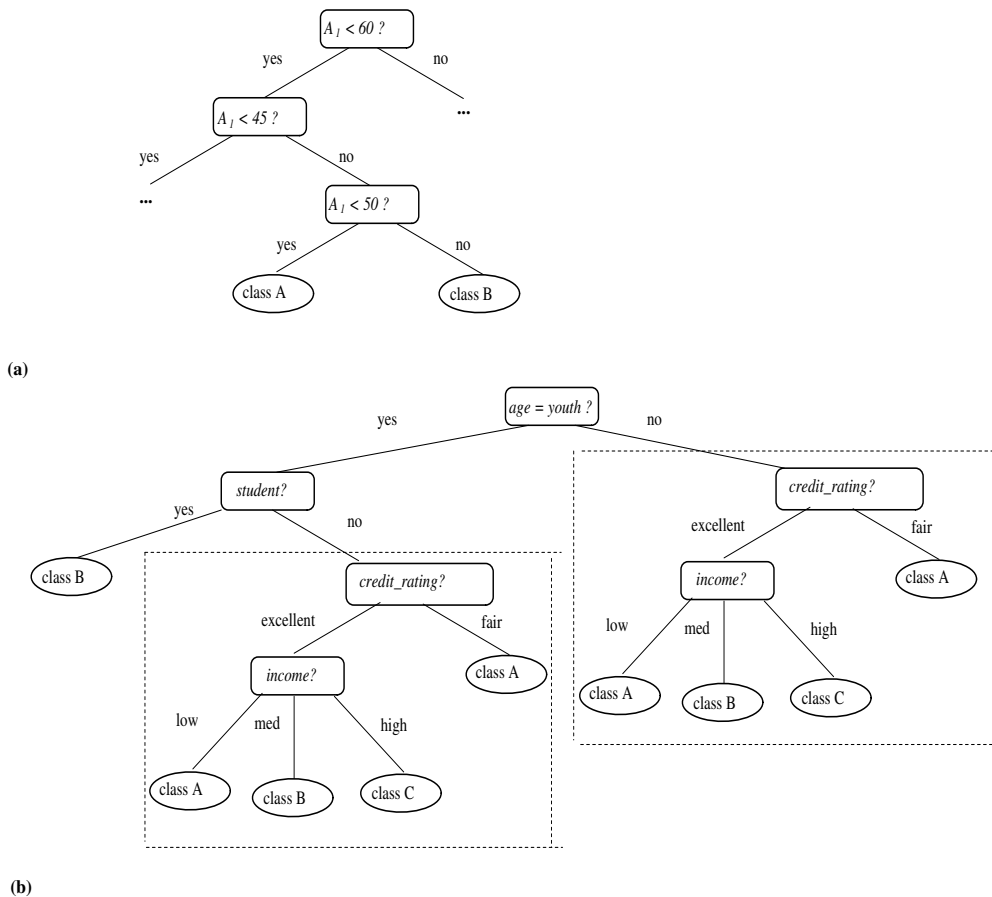


Figure 6.7: An example of subtree (a) **repetition** (where an attribute is repeatedly tested along a given branch of the tree, e.g., *age*) and (b) **replication** (where duplicate subtrees exist within a tree, such as the subtree headed by the node “*credit\_rating?*”).

that we have discussed so far have the restriction that the training tuples should reside *in memory*. In data mining applications, very large training sets of millions of tuples are common. Most often, the training data will not fit in memory! Decision tree construction therefore becomes inefficient due to swapping of the training tuples in and out of main and cache memories. More scalable approaches, capable of handling training data that are too large to fit in memory, are required. Earlier strategies to “save space” included discretizing continuous-valued attributes and sampling data at each node. These, however, still assume that the training set can fit in memory.

More recent decision tree algorithms that address the scalability issue have been proposed. Algorithms for the induction of decision trees from very large training sets include SLIQ and SPRINT, both of which can handle categorical and continuous-valued attributes. Both algorithms propose presorting techniques on disk-resident data sets that are too large to fit in memory. Both define the use of new data structures to facilitate the tree construction. SLIQ employs disk-resident *attribute lists* and a single memory-resident *class list*. The attribute lists and class list generated by SLIQ for the tuple data of Table 6.2 are shown in Figure 6.8. Each attribute has an associated attribute list, indexed by *RID* (a record identifier). Each tuple is represented by a linkage of one entry from each attribute list to an entry in the class list (holding the class label of the given tuple), which in turn is linked to its corresponding leaf node in the decision tree. The class list remains in memory since it is often accessed and modified in the building and pruning phases. The size of the class list grows proportionally with the number of tuples in the training set. When a class list cannot fit into memory, the performance of SLIQ decreases.

SPRINT uses a different *attribute list* data structure that holds the class and *RID* information, as shown in

<i>RID</i>	<i>credit_rating</i>	<i>age</i>	<i>buys_computer</i>
1	excellent	38	yes
2	excellent	26	yes
3	fair	35	no
4	excellent	49	no
...	...	...	...

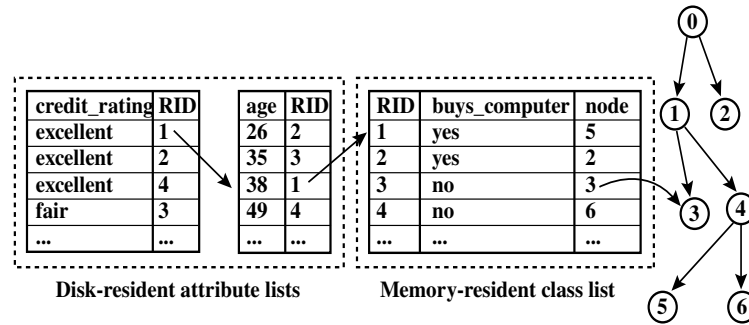
Table 6.2: Tuple data for the class *buys\_computer*.Figure 6.8: Attribute list and class list data structures used in SLIQ for the tuple data of Table 6.2. [TO EDITOR Please italicize *credit\_rating*, *age*, *RID*, *buys\_computer* and *node*.]

Figure 6.9. When a node is split, the attribute lists are partitioned and distributed among the resulting child nodes accordingly. When a list is partitioned, the order of the records in the list is maintained. Hence, partitioning lists does not require resorting. SPRINT was designed to be easily parallelized, further contributing to its scalability.

While both SLIQ and SPRINT handle disk-resident data sets that are too large to fit into memory, the scalability of SLIQ is limited by the use of its memory-resident data structure. SPRINT removes all memory restrictions, yet requires the use of a hash tree proportional in size to the training set. This may become expensive as the training set size grows.

To further enhance the scalability of decision-tree induction, a method called RainForest was proposed. It adapts to the amount of main memory available and applies to any decision tree induction algorithm. The method maintains an **AVC-set** (where AVC stands for “Attribute-Value, Classlabel”) for each attribute, at each tree node, describing the training tuples at the node. The AVC-set of an attribute *A* at node *N* gives the class label counts for each value of *A* for the tuples at *N*. Figure 6.10 shows AVC-sets for the tuple data of Table 6.1. The set of all AVC-sets at a node *N* is the **AVC-group** of *N*. The size of an AVC-set for attribute *A* at node *N* depends only on the number of distinct values of *A* and the number of classes in the set of tuples at *N*. Typically, this size should fit in memory, even for real-world data. RainForest has techniques, however, for handling the case where the AVC-group does not fit in memory. RainForest can use any attribute selection measure and was shown to be more efficient than earlier approaches employing aggregate data structures, such as SLIQ and SPRINT.

BOAT (Bootstrapped Optimistic Algorithm for Tree Construction) is a decision tree algorithm that takes a completely different approach to scalability—it is not based on the use of any special data structures. Instead, it uses a statistical technique known as “bootstrapping” (Section 6.13.3) to create several smaller samples (or subsets) of the given training data, each of which fits in memory. Each subset is used to construct a tree, resulting in several trees. The trees are examined and used to construct a new tree,  $T'$ , that turns out to be “very close” to the tree that would have been generated if all of the original training data had fit in memory. BOAT can use any attribute selection measure that selects binary splits and that is based on the notion of purity of partitions, such as the gini index. BOAT uses a lower-bound on the attribute selection measure in order to detect if this “very good” tree,  $T'$ , is different than the “real” tree,  $T$ , that would have been generated using the entire data. It refines  $T'$  in order to arrive at  $T$ .

<i>credit_rating</i>	<i>buys_computer</i>	<i>RID</i>
excellent	yes	1
excellent	yes	2
excellent	no	4
fair	no	3
...	...	...

<i>age</i>	<i>buys_computer</i>	<i>RID</i>
26	yes	2
35	no	3
38	yes	1
49	no	4
...	...	...

Figure 6.9: Attribute list data structure used in SPRINT for the tuple data of Table 6.2. [TO EDITOR Please italicize *credit\_rating*, *age*, *RID* and *buys\_computer*.]

<i>age</i>	<i>buys_computer</i>	
	yes	no
youth	2	3
middle_age	4	0
senior	3	2

<i>income</i>	<i>buys_computer</i>	
	yes	no
low	3	1
medium	4	2
high	2	2

<i>student</i>	<i>buys_computer</i>	
	yes	no
yes	6	1
no	3	4

<i>credit_rating</i>	<i>buys_computer</i>	
	yes	no
fair	6	2
excellent	3	3

Figure 6.10: The use of data structures to hold aggregate information regarding the training data (such as these AVC-sets describing the data of Table 6.1) are one approach to improving the scalability of decision tree induction. [TO EDITOR Please italicize *age*, *income*, *student*, *credit\_rating* and *buys\_computer*.]

BOAT usually requires only two scans of  $D$ . This is quite an improvement, even in comparison to traditional decision tree algorithms (such as the basic algorithm in Figure 6.3), which require one scan per level of the tree! BOAT was found to be two to three times faster than RainForest, while constructing exactly the same tree. An additional advantage of BOAT is that it can be used for incremental updates. That is, BOAT can take new insertions and deletions for the training data and update the decision tree to reflect these changes, without having to reconstruct the tree from scratch.

## 6.4 Bayesian Classification

“What are *Bayesian classifiers*?” Bayesian classifiers are statistical classifiers. They can predict class membership probabilities, such as the probability that a given tuple belongs to a particular class.

Bayesian classification is based on Bayes’ theorem, described below. Studies comparing classification algorithms have found a simple Bayesian classifier known as the *naive Bayesian classifier* to be comparable in performance with decision tree and selected neural network classifiers. Bayesian classifiers have also exhibited high accuracy and speed when applied to large databases.

Naive Bayesian classifiers assume that the effect of an attribute value on a given class is independent of the values of the other attributes. This assumption is called *class conditional independence*. It is made to simplify the computations involved and, in this sense, is considered “naive.” *Bayesian belief networks* are graphical models, which unlike naive Bayesian classifiers, allow the representation of dependencies among subsets of attributes. Bayesian belief networks can also be used for classification.

Section 6.4.1 reviews basic probability notation and Bayes’ theorem. In Section 6.4.2 you will learn how to do

naive Bayesian classification. Bayesian belief networks are described in Section 6.4.3.

### 6.4.1 Bayes' Theorem

Bayes' theorem is named after Thomas Bayes, a nonconformist English clergyman who did early work in probability and decision theory during the 18th century. Let  $\mathbf{X}$  be a data tuple. In Bayesian terms,  $\mathbf{X}$  is considered “evidence”. As usual, it is described by measurements made on a set of  $n$  attributes. Let  $H$  be some hypothesis, such as that the data tuple  $\mathbf{X}$  belongs to a specified class  $C$ . For classification problems, we want to determine  $P(H|\mathbf{X})$ , the probability that the hypothesis  $H$  holds given the “evidence” or observed data tuple  $\mathbf{X}$ . In other words, we are looking for the probability that tuple  $\mathbf{X}$  belongs to class  $C$ , given that we know the attribute description of  $\mathbf{X}$ .

$P(H|\mathbf{X})$  is the **posterior probability**, or a *posteriori probability*, of  $H$  conditioned on  $\mathbf{X}$ . For example, suppose our world of data tuples is confined to customers described by the attributes *age* and *income*, respectively, and that  $\mathbf{X}$  is a 35 year old customer with an income of \$40K. Suppose that  $H$  is the hypothesis that our customer will buy a computer. Then  $P(H|\mathbf{X})$  reflects the probability that customer  $\mathbf{X}$  will buy a computer given that we know the customer's age and income.

In contrast,  $P(H)$  is the **prior probability**, or a *priori probability*, of  $H$ . For our example, this is the probability that any given customer will buy a computer, regardless of their age, income, or any other information, for that matter. The posterior probability,  $P(H|\mathbf{X})$ , is based on more information (e.g., customer information) than the prior probability,  $P(H)$ , which is independent of  $\mathbf{X}$ .

Similarly,  $P(\mathbf{X}|H)$  is the posterior probability of  $\mathbf{X}$  conditioned on  $H$ . That is, it is the probability that a customer,  $\mathbf{X}$ , is 35 years old and earns \$40K, given that we know the customer will buy a computer.

$P(\mathbf{X})$  is the prior probability of  $\mathbf{X}$ . Using our example, it is the probability that a person from our set of customers is 35 years old and earns \$40K.

“How are these probabilities estimated?”  $P(H)$ , and  $P(\mathbf{X}|H)$ , and  $P(\mathbf{X})$  may be estimated from the given data, as we shall see below. **Bayes' theorem** is useful in that it provides a way of calculating the posterior probability,  $P(H|\mathbf{X})$ , from  $P(H)$ ,  $P(\mathbf{X}|H)$ , and  $P(\mathbf{X})$ . Bayes' theorem is

$$P(H|\mathbf{X}) = \frac{P(\mathbf{X}|H)P(H)}{P(\mathbf{X})}. \quad (6.10)$$

Now that we've got that out of the way, in the next section, we will look at how Bayes' theorem is used in the naive Bayesian classifier.

### 6.4.2 Naive Bayesian Classification

The **naive Bayesian** classifier, or **simple Bayesian** classifier, works as follows:

1. Let  $D$  be a training set of tuples and their associated class labels. As usual, each tuple is represented by an  $n$ -dimensional attribute vector,  $\mathbf{X} = (x_1, x_2, \dots, x_n)$ , depicting  $n$  measurements made on the tuple from  $n$  attributes, respectively,  $A_1, A_2, \dots, A_n$ .
2. Suppose that there are  $m$  classes,  $C_1, C_2, \dots, C_m$ . Given a tuple,  $\mathbf{X}$ , the classifier will predict that  $\mathbf{X}$  belongs to the class having the highest posterior probability, conditioned on  $\mathbf{X}$ . That is, the naive Bayesian classifier predicts that tuple  $\mathbf{X}$  belongs to the class  $C_i$  if and only if

$$P(C_i|\mathbf{X}) > P(C_j|\mathbf{X}) \quad \text{for } 1 \leq j \leq m, j \neq i.$$

Thus we maximize  $P(C_i|\mathbf{X})$ . The class  $C_i$  for which  $P(C_i|\mathbf{X})$  is maximized is called the *maximum posteriori hypothesis*. By Bayes' theorem (Equation (6.10)),

$$P(C_i|\mathbf{X}) = \frac{P(\mathbf{X}|C_i)P(C_i)}{P(\mathbf{X})}. \quad (6.11)$$

3. As  $P(\mathbf{X})$  is constant for all classes, only  $P(\mathbf{X}|C_i)P(C_i)$  need be maximized. If the class prior probabilities are not known, then it is commonly assumed that the classes are equally likely, that is,  $P(C_1) = P(C_2) = \dots = P(C_m)$ , and we would therefore maximize  $P(\mathbf{X}|C_i)$ . Otherwise, we maximize  $P(\mathbf{X}|C_i)P(C_i)$ . Note that the class prior probabilities may be estimated by  $P(C_i) = |C_{i,D}|/|D|$ , where  $|C_{i,D}|$  is the number of training tuples of class  $C_i$  in  $D$ .
4. Given data sets with many attributes, it would be extremely computationally expensive to compute  $P(\mathbf{X}|C_i)$ . In order to reduce computation in evaluating  $P(\mathbf{X}|C_i)$ , the naive assumption of **class conditional independence** is made. This presumes that the values of the attributes are conditionally independent of one another, given the class label of the tuple, i.e., that there are no dependence relationships among the attributes. Thus,

$$\begin{aligned} P(\mathbf{X}|C_i) &= \prod_{k=1}^n P(x_k|C_i) \\ &= P(x_1|C_i) \times P(x_2|C_i) \times \dots \times P(x_n|C_i). \end{aligned} \quad (6.12)$$

We can easily estimate the probabilities  $P(x_1|C_i), P(x_2|C_i), \dots, P(x_n|C_i)$  from the training tuples. Recall that here  $x_k$  refers to the value of attribute  $A_k$  for tuple  $\mathbf{X}$ . For each attribute, we look at whether the attribute is categorical or continuous-valued. For instance, to compute  $P(\mathbf{X}|C_i)$ , we consider the following:

- (a) If  $A_k$  is categorical, then  $P(x_k|C_i)$  is the number of tuples of class  $C_i$  in  $D$  having the value  $x_k$  for  $A_k$ , divided by  $|C_{i,D}|$ , the number of tuples of class  $C_i$  in  $D$ .
- (b) If  $A_k$  is continuous-valued then we need to do a bit more work, but the calculation is pretty straightforward. A continuous-valued attribute is typically assumed to have a Gaussian distribution with a mean  $\mu$  and standard deviation  $\sigma$ , defined by

$$g(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \quad (6.13)$$

so that

$$P(x_k|C_i) = g(x_k, \mu_{C_i}, \sigma_{C_i}). \quad (6.14)$$

These equations may appear daunting, but hold on! We need to compute  $\mu_{C_i}$  and  $\sigma_{C_i}$ , which are the mean (i.e., average) and standard deviation, respectively, of the values of attribute  $A_k$  for training tuples of class  $C_i$ . We then plug these two quantities into Equation (6.13), together with  $x_k$ , in order to estimate  $P(x_k|C_i)$ . For example, let  $\mathbf{X} = (35, \$40K)$ , where  $A_1$  and  $A_2$  are the attributes *age* and *income*, respectively. Let the class label attribute be *buys\_computer*. The associated class label for  $\mathbf{X}$  is “yes”, i.e., *buys\_computer* = yes. Let’s suppose that *age* has not been discretized and therefore exists as a continuous-valued attribute. Suppose that from the training set, we find that customers in  $D$  who buy a computer are  $38 \pm 12$  years of age. In other words, for attribute *age* and this class, we have  $\mu = 38$  years and  $\sigma = 12$ . We can plug these quantities, along with  $x_1 = 35$  for our tuple  $\mathbf{X}$  into Equation (6.13) in order to estimate  $P(\text{age} = 35 | \text{buys\_computer} = \text{yes})$ . For a quick review of mean and standard deviation calculations, please see Section 2.2.

5. In order to predict the class label of  $\mathbf{X}$ ,  $P(\mathbf{X}|C_i)P(C_i)$  is evaluated for each class  $C_i$ . The classifier predicts that the class label of tuple  $\mathbf{X}$  is the class  $C_i$  if and only if

$$P(\mathbf{X}|C_i)P(C_i) > P(\mathbf{X}|C_j)P(C_j) \quad \text{for } 1 \leq j \leq m, j \neq i. \quad (6.15)$$

In other words, the predicted class label is the class  $C_i$  for which  $P(\mathbf{X}|C_i)P(C_i)$  is the maximum.

“How effective are Bayesian classifiers?” Various empirical studies of this classifier in comparison to decision tree and neural network classifiers have found it to be comparable in some domains. In theory, Bayesian classifiers have the minimum error rate in comparison to all other classifiers. However, in practice this is not always the case owing to inaccuracies in the assumptions made for its use, such as class conditional independence, and the lack of available probability data.

Bayesian classifiers are also useful in that they provide a theoretical justification for other classifiers that do not explicitly use Bayes' theorem. For example, under certain assumptions, it can be shown that many neural network and curve-fitting algorithms output the *maximum posteriori* hypothesis, as does the naive Bayesian classifier.

**Example 6.4 Predicting a class label using naive Bayesian classification.** We wish to predict the class label of a tuple using naive Bayesian classification, given the same training data as in Example 6.1 for decision tree induction. The training data are in Table 6.1. The data tuples are described by the attributes *age*, *income*, *student*, and *credit\_rating*. The class label attribute, *buys\_computer*, has two distinct values (namely, {*yes*, *no*}). Let  $C_1$  correspond to the class *buys\_computer* = *yes* and  $C_2$  correspond to *buys\_computer* = *no*. The tuple we wish to classify is

$$\mathbf{X} = (\text{age} = \text{youth}, \text{income} = \text{medium}, \text{student} = \text{yes}, \text{credit\_rating} = \text{fair})$$

We need to maximize  $P(\mathbf{X}|C_i)P(C_i)$ , for  $i = 1, 2$ .  $P(C_i)$ , the prior probability of each class, can be computed based on the training tuples:

$$\begin{aligned} P(\text{buys\_computer} = \text{yes}) &= 9/14 = 0.643 \\ P(\text{buys\_computer} = \text{no}) &= 5/14 = 0.357 \end{aligned}$$

To compute  $P(\mathbf{X}|C_i)$ , for  $i = 1, 2$ , we compute the following conditional probabilities:

$$\begin{aligned} P(\text{age} = \text{youth} \mid \text{buys\_computer} = \text{yes}) &= 2/9 = 0.222 \\ P(\text{age} = \text{youth} \mid \text{buys\_computer} = \text{no}) &= 3/5 = 0.600 \\ P(\text{income} = \text{medium} \mid \text{buys\_computer} = \text{yes}) &= 4/9 = 0.444 \\ P(\text{income} = \text{medium} \mid \text{buys\_computer} = \text{no}) &= 2/5 = 0.400 \\ P(\text{student} = \text{yes} \mid \text{buys\_computer} = \text{yes}) &= 6/9 = 0.667 \\ P(\text{student} = \text{yes} \mid \text{buys\_computer} = \text{no}) &= 1/5 = 0.200 \\ P(\text{credit\_rating} = \text{fair} \mid \text{buys\_computer} = \text{yes}) &= 6/9 = 0.667 \\ P(\text{credit\_rating} = \text{fair} \mid \text{buys\_computer} = \text{no}) &= 2/5 = 0.400 \end{aligned}$$

Using the above probabilities, we obtain

$$\begin{aligned} P(\mathbf{X}|\text{buys\_computer} = \text{yes}) &= P(\text{age} = \text{youth} \mid \text{buys\_computer} = \text{yes}) \times \\ &\quad P(\text{income} = \text{medium} \mid \text{buys\_computer} = \text{yes}) \times \\ &\quad P(\text{student} = \text{yes} \mid \text{buys\_computer} = \text{yes}) \times \\ &\quad P(\text{credit\_rating} = \text{fair} \mid \text{buys\_computer} = \text{yes}) \\ &= 0.222 \times 0.444 \times 0.667 \times 0.667 = 0.044. \end{aligned}$$

Similarly,

$$P(\mathbf{X}|\text{buys\_computer} = \text{no}) = 0.600 \times 0.400 \times 0.200 \times 0.400 = 0.019.$$

To find the class,  $C_i$ , that maximizes  $P(\mathbf{X}|C_i)P(C_i)$ , we compute

$$\begin{aligned} P(\mathbf{X}|\text{buys\_computer} = \text{yes})P(\text{buys\_computer} = \text{yes}) &= 0.044 \times 0.643 = 0.028 \\ P(\mathbf{X}|\text{buys\_computer} = \text{no})P(\text{buys\_computer} = \text{no}) &= 0.019 \times 0.357 = 0.007 \end{aligned}$$

Therefore, the naive Bayesian classifier predicts *buys\_computer* = *yes* for tuple  $\mathbf{X}$ . ■

“What if I encounter probability values of zero?” Recall that in Equation (6.12), we estimate  $P(\mathbf{X}|C_i)$  as the product of the probabilities  $P(x_1|C_i), P(x_2|C_i), \dots, P(x_n|C_i)$ , based on the assumption of class conditional independence. These probabilities can be estimated from the training tuples (step 4). We need to compute  $P(\mathbf{X}|C_i)$  for each class ( $i = 1, 2, \dots, m$ ) in order to find the class  $C_i$  for which  $P(\mathbf{X}|C_i)P(C_i)$  is the maximum (step 5). Let's consider this calculation. For each attribute-value pair (i.e.,  $A_k = x_k$ , for  $k = 1, 2, \dots, n$ ) in tuple  $\mathbf{X}$ , we need to count the number of tuples having that attribute-value pair, per class (i.e., per  $C_i$ , for  $i = 1, \dots, m$ ). In Example 6.4, we have two classes ( $m = 2$ ), namely *buys\_computer* = *yes* and *buys\_computer* = *no*. Therefore, for the attribute-value pair *student* = *yes* of  $\mathbf{X}$ , say, we need two counts—the number of customers who are students and for which *buys\_computer* = *yes* (which contributes to  $P(\mathbf{X}|\text{buys\_computer} = \text{yes})$ ), and the number of customers who are students and for which *buys\_computer* = *no* (which contributes to  $P(\mathbf{X}|\text{buys\_computer} = \text{no})$ ). But what if, say, there are no training tuples representing students for the class *buys\_computer* = *no*, resulting in



$P(\text{student} = \text{yes} | \text{buys\_computer} = \text{no}) = 0$ ? In other words, what happens if we should end up with a probability value of zero for some  $P(x_k | C_i)$ ? Plugging this zero value into Equation (6.12) would return a zero probability for  $P(\mathbf{X} | C_i)$ , even though, without the zero probability, we may have ended up with a high probability, suggesting that  $\mathbf{X}$  belonged to class  $C_i$ ! A zero probability cancels the effects of all of the other (posteriori) probabilities (on  $C_i$ ) involved in the product.

There is a simple trick to avoid this problem. We can assume that our training database,  $D$ , is so large so that adding one to each count that we need would only make a negligible difference in the estimated probability value, yet would conveniently avoid the case of probability values of zero. This technique for probability estimation is known as the **Laplacian correction** or **Laplace estimator**, named after Pierre Laplace, a French mathematician who lived from 1749-1827. If we have, say,  $q$  counts to which we each add one, then we must remember to add  $q$  to the corresponding denominator used in the probability calculation. We illustrate this technique in the following example.

**Example 6.5 Using the Laplacian correction to avoid computing probability values of zero.** Suppose that for the class  $\text{buys\_computer} = \text{yes}$  in some training database,  $D$ , containing 1,000 tuples, we have 0 tuples with  $\text{income} = \text{low}$ , 990 tuples with  $\text{income} = \text{medium}$ , and 10 tuples with  $\text{income} = \text{high}$ . The probabilities of these events, without the Laplacian correction, are 0, 0.990 (from 990/1000), and 0.010 (from 10/1,000), respectively. Using the Laplacian correction for the three quantities, we pretend that we have 1 more tuple for each income-value pair. In this way, we instead obtain the following probabilities (rounded up to three decimal places):

$$\frac{1}{1,003} = 0.001, \frac{991}{1,003} = 0.988, \text{ and } \frac{11}{1,003} = 0.011,$$

respectively. The “corrected” probability estimates are close to their “uncorrected” counterparts, yet the zero probability value is avoided. ■

### 6.4.3 Bayesian Belief Networks

The naive Bayesian classifier makes the assumption of class conditional independence, that is, given the class label of a tuple, the values of the attributes are assumed to be conditionally independent of one another. This simplifies computation. When the assumption holds true, then the naive Bayesian classifier is the most accurate in comparison with all other classifiers. In practice, however, dependencies can exist between variables. **Bayesian belief networks** specify joint conditional probability distributions. They allow class conditional independencies to be defined between subsets of variables. They provide a graphical model of causal relationships, on which learning can be performed. Trained Bayesian belief networks can be used for classification. Bayesian belief networks are also known as **belief networks**, **Bayesian networks**, and **probabilistic networks**. For brevity, we will refer to them as belief networks.

A belief network is defined by two components—a *directed acyclic graph* and a set of *conditional probability tables* (Figure 6.11). Each node in the directed acyclic graph represents a random variable. The variables may be discrete or continuous-valued. They may correspond to actual attributes given in the data or to “hidden variables” believed to form a relationship (e.g., in the case of medical data, a hidden variable may indicate a syndrome, representing a number of symptoms that, together, characterize a specific disease). Each arc represents a probabilistic dependence. If an arc is drawn from a node  $Y$  to a node  $Z$ , then  $Y$  is a **parent** or **immediate predecessor** of  $Z$ , and  $Z$  is a **descendent** of  $Y$ . *Each variable is conditionally independent of its nondescendents in the graph, given its parents.*

Figure 6.11 is a simple belief network, adapted from [RBKK95] for six Boolean variables. The arcs in Figure 6.11(a) allow a representation of causal knowledge. For example, having lung cancer is influenced by a person’s family history of lung cancer, as well as whether or not the person is a smoker. Note that the variable *PositiveXRay* is independent of whether the patient has a family history of lung cancer or is a smoker, given that we know the patient has lung cancer. In other words, once we know the outcome of the variable *LungCancer*, then the variables *FamilyHistory* and *Smoker* do not provide any additional information regarding *PositiveXRay*. The arcs also show

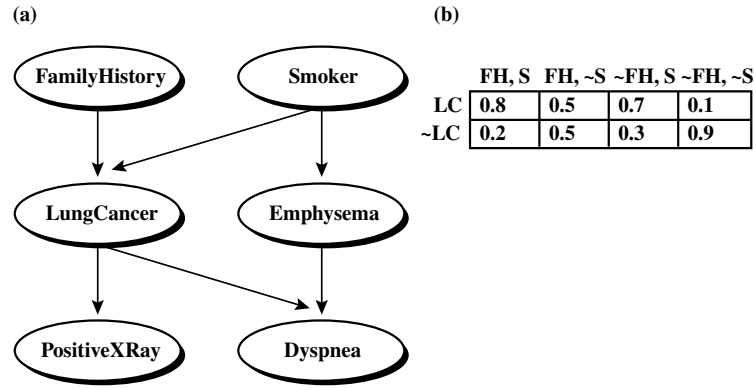


Figure 6.11: A simple Bayesian belief network: (a) A proposed causal model, represented by a directed acyclic graph. (b) The conditional probability table for the values of the variable *LungCancer* (*LC*) showing each possible combination of the values of its parent nodes, *FamilyHistory* (*FH*) and *Smoker* (*S*). Figure is adapted from [RBKK95]. [TO EDITOR Please italicize the text in each node as they are attribute names, along with their corresponding abbreviations (LC, FH, etc.).]

that the variable *LungCancer* is conditionally independent of *Emphysema*, given its parents, *FamilyHistory* and *Smoker*.

A belief network has one **conditional probability table (CPT)** for each variable. The CPT for a variable *Y* specifies the conditional distribution  $P(Y|Parents(Y))$ , where  $Parents(Y)$  are the parents of *Y*. Figure 6.11(b) shows a CPT for the variable *LungCancer*. The conditional probability for each known value of *LungCancer* is given for each possible combination of values of its parents. For instance, from the upper leftmost and bottom rightmost entries, respectively, we see that

$$P(LungCancer = yes \mid FamilyHistory = yes, Smoker = yes) = 0.8$$

$$P(LungCancer = no \mid FamilyHistory = no, Smoker = no) = 0.9$$

Let  $\mathbf{X} = (x_1, \dots, x_n)$  be a data tuple described by the variables or attributes  $Y_1, \dots, Y_n$ , respectively. Recall that each variable is conditionally independent of its nondescendants in the network graph, given its parents. This allows the network to provide a complete representation of the existing joint probability distribution with the following equation:

$$P(x_1, \dots, x_n) = \prod_{i=1}^n P(x_i | Parents(Y_i)), \quad (6.16)$$

where  $P(x_1, \dots, x_n)$  is the probability of a particular combination of values of  $\mathbf{X}$ , and the values for  $P(x_i | Parents(Y_i))$  correspond to the entries in the CPT for  $Y_i$ .

A node within the network can be selected as an “output” node, representing a class label attribute. There may be more than one output node. Various algorithms for learning can be applied to the network. Rather than returning a single class label, the classification process can return a probability distribution that gives the probability of each class.

#### 6.4.4 Training Bayesian Belief Networks

“How does a Bayesian belief network learn?” In the learning or training of a belief network, a number of scenarios are possible. The network **topology** (or “layout” of nodes and arcs) may be given in advance or inferred from the data. The network variables may be *observable* or *hidden* in all or some of the training tuples. The case of hidden data is also referred to as *missing values* or *incomplete data*.

Several algorithms exist for learning the network topology from the training data given observable variables. The problem is one of discrete optimization. For solutions, please see the bibliographic notes at the end of this chapter. Human experts usually have a good grasp of the direct conditional dependencies that hold in the domain under analysis, which helps in network design. Experts must specify conditional probabilities for the nodes that participate in direct dependencies. These probabilities can then be used to compute the remaining probability values.

If the network topology is known and the variables are observable, then training the network is straightforward. It consists of computing the CPT entries, as is similarly done when computing the probabilities involved in naive Bayesian classification.

When the network topology is given and some of the variables are hidden, there are various methods to choose from for training the belief network. We will describe a promising method of gradient descent. For those without an advanced math background, the description may look rather intimidating with its calculus-packed formulae. However, packaged software exists to solve these equations and the general idea is easy to follow.

Let  $D$  be a training set of data tuples,  $\mathbf{X}_1, \mathbf{X}_2, \dots, \mathbf{X}_{|D|}$ . Training the belief network means that we must learn the values of the CPT entries. Let  $w_{ijk}$  be a CPT entry for the variable  $Y_i = y_{ij}$  having the parents  $U_i = u_{ik}$ , where  $w_{ijk} \equiv P(Y_i = y_{ij} | U_i = u_{ik})$ . For example, if  $w_{ijk}$  is the upper leftmost CPT entry of Figure 6.11(b), then  $Y_i$  is *LungCancer*;  $y_{ij}$  is its value, “yes”;  $U_i$  lists the parent nodes of  $Y_i$ , namely,  $\{\text{FamilyHistory}, \text{Smoker}\}$ ; and  $u_{ik}$  lists the values of the parent nodes, namely,  $\{\text{“yes”}, \text{“yes”}\}$ . The  $w_{ijk}$  are viewed as weights, analogous to the weights in hidden units of neural networks (Section 6.6). The set of weights is collectively referred to as  $\mathbf{W}$ . The weights are initialized to random probability values. A *gradient descent* strategy performs greedy hill-climbing. At each iteration, the weights are updated and will eventually converge to a local optimum solution.

A **gradient descent** strategy is used to search for the  $w_{ijk}$  values that best model the data, based on the assumption that each possible setting of  $w_{ijk}$  is equally likely. Such a strategy is iterative. It searches for a solution along the negative of the gradient (i.e., steepest descent) of a criterion function. We want to find the set of weights,  $\mathbf{W}$ , that maximize this function. To start with, the weights are initialized to random probability values. The gradient descent method performs greedy hill-climbing in that, at each iteration or step along the way, the algorithm moves towards what appears to be the best solution at the moment, without backtracking. The weights are updated at each iteration. Eventually, they converge to a local optimum solution.

For our problem, we maximize  $P_w(D) = \prod_{d=1}^{|D|} P_w(\mathbf{X}_d)$ . This can be done by following the gradient of  $\ln P_w(S)$ , which makes the problem simpler. Given the network topology and initialized  $w_{ijk}$ , the algorithm proceeds as follows:

1. **Compute the gradients:** For each  $i, j, k$ , compute

$$\frac{\partial \ln P_w(D)}{\partial w_{ijk}} = \sum_{d=1}^{|D|} \frac{P(Y_i = y_{ij}, U_i = u_{ik} | \mathbf{X}_d)}{w_{ijk}} \quad (6.17)$$

The probability in the right-hand side of Equation (6.17) is to be calculated for each training tuple,  $\mathbf{X}_d$ , in  $D$ . For brevity, let's refer to this probability simply as  $p$ . When the variables represented by  $Y_i$  and  $U_i$  are hidden for some  $\mathbf{X}_d$ , then the corresponding probability  $p$  can be computed from the observed variables of the tuple using standard algorithms for Bayesian network inference such as those available in the commercial software package HUGIN (<http://www.hugin.dk>).

2. **Take a small step in the direction of the gradient:** The weights are updated by

$$w_{ijk} \leftarrow w_{ijk} + (l) \frac{\partial \ln P_w(D)}{\partial w_{ijk}}, \quad (6.18)$$

where  $l$  is the **learning rate** representing the step size and  $\frac{\partial \ln P_w(D)}{\partial w_{ijk}}$  is computed from Equation (6.17). The learning rate is set to a small constant and helps with convergence.

3. **Renormalize the weights:** Because the weights  $w_{ijk}$  are probability values, they must be between 0.0 and 1.0, and  $\sum_j w_{ijk}$  must equal 1 for all  $i, k$ . These criteria are achieved by renormalizing the weights after they have been updated by Equation (6.18).

Algorithms that follow this form of learning are called Adaptive Probabilistic Networks. Other methods for training belief networks are referenced in the bibliographic notes at the end of this chapter. Belief networks are computationally intensive. Because belief networks provide explicit representations of causal structure, a human expert can provide prior knowledge to the training process in the form of network topology and/or conditional probability values. This can significantly improve the learning rate.

## 6.5 Rule-Based Classification

In this section, we look at rule-based classifiers, where the learned model is represented as a set of IF-THEN rules. We first examine how such rules are used for classification. We then study ways in which they can be extracted, either from a decision tree or directly from the training data.

### 6.5.1 Using IF-THEN Rules For Classification

Rules are a good way of representing information or bits of knowledge. A **rule-based classifier** uses a set of IF-THEN rules for classification. An **IF-THEN** rule is an expression of the form

IF *condition* THEN *conclusion*.

An example is rule *R1*,

R1: IF *age* = *youth* AND *student* = *yes* THEN *buys\_computer* = *yes*.

The “IF”-part (or left-hand side) of a rule is known as the **rule antecedent** or **precondition**. The “THEN”-part (or right-hand side) is the **rule consequent**. In the rule antecedent, the condition consists of one or more *attribute tests* (such as *age* = *youth*, and *student* = *yes*) that are logically ANDed. The rule’s consequent contains a class prediction (in this case, we are predicting whether a customer will buy a computer). *R1* can also be written as

R1: (*age* = *youth*)  $\wedge$  (*student* = *yes*)  $\Rightarrow$  (*buys\_computer* = *yes*).

If the condition (that is, all of the attribute tests) in a rule antecedent holds true for a given tuple, we say that the rule antecedent is **satisfied** (or simply, that the rule is satisfied) and that the rule **covers** the tuple.

A rule *R* can be assessed by its coverage and accuracy. Given a tuple, *X*, from a class-labeled data set, *D*, let  $n_{covers}$  be the number of tuples covered by *R*;  $n_{correct}$  be the number of tuples correctly classified by *R*; and  $|D|$  be the number of tuples in *D*. We can define the **coverage** and **accuracy** of *R* as

$$coverage(R) = \frac{n_{covers}}{|D|} \quad (6.19)$$

$$accuracy(R) = \frac{n_{correct}}{n_{covers}}. \quad (6.20)$$

That is, a rule’s coverage is the percentage of tuples that are covered by the rule (i.e., whose attribute values hold true for the rule’s antecedent). For a rule’s accuracy, we look at the tuples that it covers and see what percentage of them the rule can correctly classify.

**Example 6.6 Rule accuracy and coverage.** Let’s go back to our data of Table 6.1. These are class-labeled tuples from the *Allelectronics* customer database. Our task is to predict whether or not a customer will buy a computer. Consider rule *R1* above, which covers 2 of the 14 tuples. It can correctly classify both tuples. Therefore,  $coverage(R1) = 2/14 = 14.28\%$  and  $accuracy(R1) = 2/2 = 100\%$ . ■

Let's see how we can use rule-based classification to predict the class label of a given tuple,  $X$ . If a rule is satisfied by  $X$ , the rule is said to be **triggered**. For example, suppose we have:

$$X = (\text{age} = \text{youth}, \text{income} = \text{medium}, \text{student} = \text{yes}, \text{credit\_rating} = \text{fair}).$$

We would like to classify  $X$  according to *buys\_computer*.  $X$  satisfies  $R1$ , which triggers the rule.

If  $R1$  is the only rule satisfied, then the rule **fires** by returning the class prediction for  $X$ . Note that triggering does not always mean firing because they may be more than one rule that is satisfied! If more than one rule is triggered, we have a potential problem. What if they each specify a different class? Or what if no rule is satisfied by  $X$ ?

We tackle the first question. If more than one rule is triggered, we need a **conflict resolution strategy** to figure out which rule gets to fire and assign its class prediction to  $X$ . There are many possible strategies. We look at two, namely *size ordering* and *rule ordering*.

The **size ordering** scheme assigns the highest priority to the triggering rule that has the “toughest” requirements, where toughness is measured by the rule antecedent *size*. That is, the triggering rule with the most attribute tests is fired.

The **rule ordering** scheme prioritizes the rules beforehand. The ordering may be *class-based* or *rule-based*. With **class-based ordering**, the classes are sorted in order of decreasing “importance”, such as by decreasing *order of prevalence*. That is, all of the rules for the most prevalent (or most frequent) class come first, the rules for the next prevalent class come next, and so on. Alternatively, they may be sorted based the misclassification cost per class. Within each class, the rules are not ordered—they don't have to be since they all predict the same class (and so there can be no class conflict!) With **rule-based ordering**, the rules are organized into one long priority list, according to some measure of rule quality such as accuracy, coverage, or size (number of attribute tests in the rule antecedent), or based on advice from domain experts. When rule-ordering is used, the rule set is known as a **decision list**. With rule-ordering, the triggering rule that appears earliest in the list has highest priority, and so it gets to fire its class prediction. Any other rule that satisfies  $X$  is ignored. Most rule-based classification systems use a class-based rule ordering strategy.

Note that in the first strategy, overall the rules are *unordered*. They can be applied in any order when classifying a tuple. That is, a disjunction (logical OR) is implied between each of the rules. Each rule represents a stand-alone nugget or piece of knowledge. This is in contrast to the rule-ordering (decision list) scheme for which rules must be applied in the prescribed order so as to avoid conflicts. Each rule in a decision list implies the negation of the rules that come before it in the list. Hence, rules in a decision list are more difficult to interpret.

Now that we have seen how we can handle conflicts, let's go back to the scenario where there is no rule satisfied by  $X$ . How, then, can we determine the class label of  $X$ ? In this case, a fallback or **default rule** can be set up to specify a default class, based on a training set. This may be the class in majority, or the majority class of the tuples that were not covered by any rule. The default rule is evaluated at the end, if and only if no other rule covers  $X$ . The condition in the default rule is empty. In this way, the rule fires when no other rule is satisfied.

In the following sections, we examine how to build a rule-based classifier.

### 6.5.2 Rule Extraction from a Decision Tree

In Section 6.3, we learned how to build a decision tree classifier from a set of training data. Decision tree classifiers are a popular method of classification—it is easy to understand how decision trees work and they are known for their accuracy. Decision trees can become large and difficult to interpret. In this subsection, we look at how to build a rule-based classifier by extracting IF-THEN rules from a decision tree. In comparison with a decision tree, the IF-THEN rules may be easier for humans to understand, particularly if the decision tree is very large.

To extract rules from a decision tree, one rule is created for each path from the root to a leaf node. Each splitting criterion along a given path is logically ANDed to form the rule antecedent (“IF” part). The leaf node holds the class prediction, forming the rule consequent (“THEN” part).

**Example 6.7 Extracting classification rules from a decision tree.** The decision tree of Figure 6.2 can be converted to classification IF-THEN rules by tracing the path from the root node to each leaf node in the tree. The rules extracted from Figure 6.2 are

R1: IF <i>age</i> = <i>youth</i> AND <i>student</i> = <i>no</i>	THEN <i>buys_computer</i> = <i>no</i>
R2: IF <i>age</i> = <i>youth</i> AND <i>student</i> = <i>yes</i>	THEN <i>buys_computer</i> = <i>yes</i>
R3: IF <i>age</i> = <i>middle_aged</i>	THEN <i>buys_computer</i> = <i>yes</i>
R4: IF <i>age</i> = <i>senior</i> AND <i>credit_rating</i> = <i>excellent</i>	THEN <i>buys_computer</i> = <i>yes</i>
R5: IF <i>age</i> = <i>senior</i> AND <i>credit_rating</i> = <i>fair</i>	THEN <i>buys_computer</i> = <i>no</i>

■

A disjunction (logical OR) is implied between each of the extracted rules. Since the rules are extracted directly from the tree, they are **mutually exclusive** and **exhaustive**. By *mutually exclusive*, this means that we cannot have rule conflicts here because no two rules will be triggered for the same tuple. (We have one rule per leaf, and any tuple can map to only one leaf). By *exhaustive*, there is one rule for each possible attribute-value combination, so that this set of rules does not require a default rule. Therefore, the order of the rules does not matter—they are *unordered*.

[FROM MK: **Changes made in this and next paragraph:**] Since we end up with one rule per leaf, the set of extracted rules is not much simpler than the corresponding decision tree! The extracted rules may be even more difficult to interpret than the original trees in some cases. As an example, Figure 6.7 showed decision trees that suffer from subtree repetition and replication. The resulting set of rules extracted can be large and difficult to follow, since some of the attribute tests may be irrelevant or redundant. So, the plot thickens. Although it is easy to extract rules from a decision tree, we may need to do some more work by pruning the resulting rule set.

“How can we prune the rule set?” For a given rule antecedent, any condition that does not improve the estimated accuracy of the rule can be pruned (i.e., removed), thereby generalizing the rule. C4.5 extracts rules from an unpruned tree, and then prunes the rules using a pessimistic approach similar to its tree pruning method. The training tuples and their associated class labels are used to estimate rule accuracy. However, since this would result in an optimistic estimate, alternatively, the estimate is adjusted to compensate for the bias, resulting in a pessimistic estimate. In addition, any rule that does not contribute to the overall accuracy of the entire rule set can also be pruned.

Other problems arise during rule pruning, however, as the rules *will no longer be* mutually exclusive and exhaustive. For conflict resolution, C4.5 adopts a **class-based ordering scheme**. It groups all rules for a single class together, and then determines a ranking of these class rule sets. Within a rule set, the rules are not ordered. C4.5 orders the class rule sets so as to minimize the number of *false positive errors* (i.e., where a rule predicts a class, *C*, but the actual class is not *C*). The class rule set with the least number of false positives is examined first. Once pruning is complete, a final check is done to remove any duplicates. When choosing a default class, C4.5 does not choose the majority class, since this class will likely have many rules for its tuples. Instead, it selects the class that contains the most training tuples that were not covered by any rule.

### 6.5.3 Rule Extraction from the Training Data

IF-THEN rules can be extracted directly from the training data (that is, without having to generate a decision tree first) using a **sequential covering algorithm**. The name comes from the notion that the rules are learned *sequentially* (one at a time), where each rule for a given class will ideally *cover* many of the tuples of that class (and hopefully none of the tuples of other classes). Sequential covering algorithms are the most widely-used approach to mining disjunctive sets of classification rules.

There are many sequential covering algorithms. Popular variations include AQ, CN2, and the more recent, RIPPER. The general strategy is as follows. Rules are learned one at a time. Each time a rule is learned, the tuples covered by the rule are removed, and the process repeats on the remaining tuples. This sequential learning of rules

is in contrast to decision tree induction. Since the path to each leaf in a decision tree corresponds to a rule, we can consider decision tree induction as learning a set of rules *simultaneously*.

**Algorithm: Sequential covering.** Learn a set of IF-THEN rules for classification.

**Input:**

- $D$ , a data set class-labeled tuples;
- $Att\text{-}vals$ , the set of all attributes and their possible values.

**Output:** A set of IF-THEN rules.

**Method:**

```

(1)  $Rule\_set = \{\}$ ; // initial set of rules learned is empty
(2) for each class  $c$  do
(3)   repeat
(4)      $Rule = \text{Learn\_One\_Rule}(D, Att - vals, c)$ ;
(5)     remove tuples covered by  $Rule$  from  $D$ ;
(6)   until terminating condition;
(7)    $Rule\_set = Rule\_set + Rule$  // add new rule to rule set
(8) endfor
(9) return  $Rule\_Set$ ;
```

Figure 6.12: Basic sequential covering algorithm.

A basic sequential covering algorithm is shown in Figure 6.12. Here, rules are learned for one class at a time. Ideally, when learning a rule for a class,  $C_i$ , we would like the rule to cover all (or many) of the training tuples of class  $C$  and none (or few) of the tuples from other classes. In this way, the rules learned should be of high accuracy. The rules need not necessarily be of high coverage. This is because we can have more than one rule for a class, so that different rules may cover different tuples within the same class. The process continues until the terminating condition is met, such as when there are no more training tuples, or the quality of a rule returned is below a user-specified threshold. The *Learn\_One\_Rule* procedure finds the “best” rule for the current class, given the current set of training tuples.

“How are rules learned?” Typically, rules are grown in a *general-to-specific* manner (Figure 6.13). We can think of this as a beam search, where we start off with an empty rule and then gradually keep appending attribute tests to it. We append by adding the attribute test as a logical conjunct to the existing condition of the rule antecedent. Suppose our training set,  $D$ , consists of loan application data. Attributes regarding each applicant include their age, income, education level, residence, credit rating, and the term of the loan. The classifying attribute is *loan\_decision*, which indicates whether a loan is accepted (considered safe) or rejected (considered risky). To learn a rule for the class “accept”, we start off with the most general rule possible, that is, the condition of the rule antecedent is empty. The rule is:

IF    THEN *loan\_decision* = *accept*.

We then consider each possible attribute-test that may be added to the rule. These can be derived from the parameter *Att-vals*, which contains a list of attributes with their associated values. For example, for an attribute-value pair (*att*, *val*), we can consider attribute tests such as  $att = val$ ,  $att \leq val$ ,  $att > val$ , and so on. Typically, the training data will contain many attributes, each of which may have several possible values. Finding an optimal rule set becomes computationally explosive. Instead, *Learn\_One\_Rule* adopts a greedy depth-first strategy. Each time it is faced with adding a new attribute test (conjunct) to the current rule, it picks the one that most improves the rule quality, based on the training samples. We will say more about rule quality measures in a minute. For

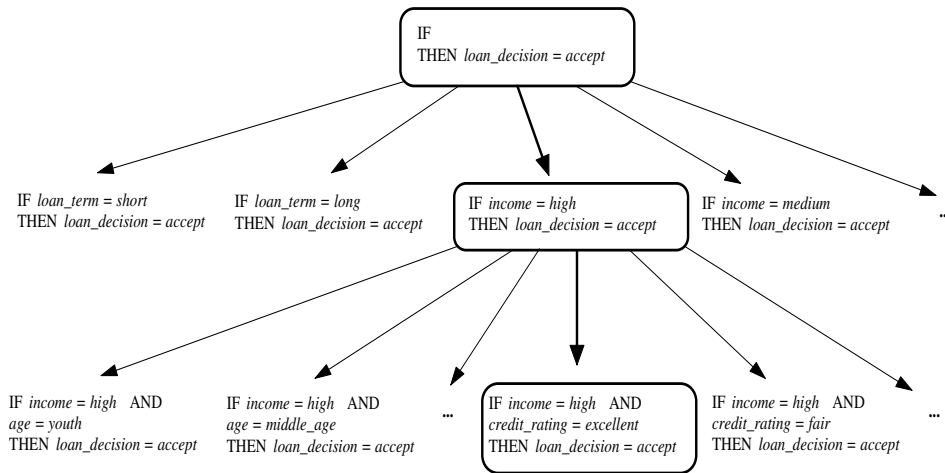


Figure 6.13: A general-to-specific search through rule space.

the moment, let's say we use rule accuracy as our quality measure. Getting back to our example with Figure 6.13, suppose *Learn\_One\_Rule* finds that the attribute test *income = high* best improves the accuracy of our current (empty) rule. We append it to the condition, so that the current rule becomes

IF *income = high* THEN *loan\_decision = accept*.

Each time we add an attribute test to a rule, the resulting rule should cover more of the “accept” tuples. During the next iteration, we again consider the possible attribute tests and end up selecting *credit\_rating = excellent*. Our current rule grows to become

IF *income = high* AND *credit\_rating = excellent* THEN *loan\_decision = accept*.

The process repeats, where at each step, we continue to greedily grow rules until the resulting rule meets an acceptable quality level.

Greedy search does not allow for backtracking. At each step, we *heuristically* add what appears to be the best choice at the moment. What if we unknowingly made a poor choice along the way? To lessen the chance of this happening, instead of selecting the best attribute test to append to the current rule, we can select the best  $k$  attribute tests. In this way, we perform a beam search of width  $k$  wherein we maintain the  $k$  best candidates overall at each step, rather than a single best candidate.

## Rule Quality Measures

*Learn\_One\_Rule* needs a measure of rule quality. Every time it considers an attribute test, it must check to see if appending such a test to the current rule's condition will result in an improved rule. Accuracy may seem like an obvious choice at first, but consider the following example.

**Example 6.8 Choosing between two rules based on accuracy.** Consider the two rules as illustrated in Figure 6.14. Both are for the class *loan\_decision = accept*. We use “*a*” to represent the tuples of class “*accept*” and “*r*” for the tuples of class “*reject*”. Rule *R1* correctly classifies 38 of the 40 tuples it covers. Rule *R2* covers only 2 tuples, which it correctly classifies. Their respective accuracies are 95% and 100%. Thus, *R2* has greater accuracy than *R1*, however, it is not the better rule because of its small coverage. ■



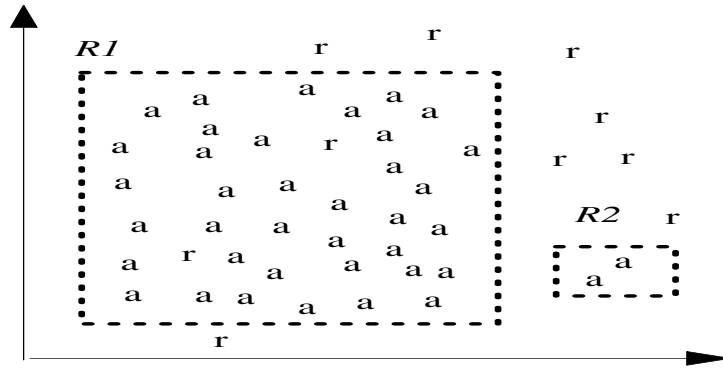


Figure 6.14: Rules rules for the class *loan\_decision = accept*, showing *accept* (*a*) and *reject* (*r*) tuples.

From the above example, we see that accuracy on its own is not a reliable estimate of rule quality. Coverage on its own is not useful either—for a given class we could have a rule that covers many tuples, most of which belong to other classes! Thus, we seek other measures for evaluating rule quality, which may integrate aspects of accuracy and coverage. Here we will look at a few, namely *entropy*, another based on *information gain*, and a *statistical test* that considers coverage. For our discussion, suppose we are learning rules for the class *c*. Our current rule is *R*: IF *condition* THEN *class = c*. We want to see if logically ANDing a given attribute test to *condition* would result in a better rule. We call the new condition, *condition'*, where *R'*: IF *condition'* THEN *class = c* is our potential new rule. In other words, we want to see if *R'* is any better than *R*.

We have already seen entropy in our discussion of the information gain measure used for attribute selection in decision tree induction (Section 6.3.2, Equation 6.1). It is also known as the *expected information* needed to classify a tuple in data set, *D*. Here, *D* is the set of tuples covered by *condition'* and  $p_i$  is the probability of class  $C_i$  in *D*. The lower the entropy, the better *condition'* is. Entropy prefers conditions that cover a large number of tuples of a single class and few tuples of other classes.

Another measure is based on information gain and was proposed in FOIL, a sequential covering algorithm that learns first-order logic rules. Learning first-order rules is more complex because such rules contain variables, whereas the rules we are concerned with in this section are propositional (that is, variable-free).<sup>7</sup> In machine learning, the tuples of the class for which we are learning rules are called *positive* tuples, while the remaining tuples are *negative*. Let *pos* (*neg*) be the number of positive (negative) tuples covered by *R*. Let *pos'* (*neg'*) be the number of positive (negative) tuples covered by *R'*. FOIL assesses the information gained by extending *condition* as

$$FOIL\_Gain = pos' \times \left( \log_2 \frac{pos'}{pos' + neg'} - \log_2 \frac{pos}{pos + neg} \right). \quad (6.21)$$

It favors rules that have high accuracy and cover many positive tuples.

We can also use a statistical test of significance to determine if the apparent effect of a rule is not attributed to chance but instead indicates a genuine correlation between attribute values and classes. The test compares the observed distribution among classes of tuples covered by a rule with the expected distribution that would result if the rule made predictions at random. We want to assess whether any observed differences between these two distributions may be attributed to chance. We can use the **likelihood ratio statistic**,

$$Likelihood\_Ratio = 2 \sum_{i=1}^m f_i \log \left( \frac{f_i}{e_i} \right), \quad (6.22)$$

where *m* is the number of classes. For tuples satisfying the rule,  $f_i$  is the observed frequency of each class *i* among the tuples.  $e_i$  is what we would expect the frequency of each class *i* to be if the rule made random predictions. The

<sup>7</sup>Incidentally, FOIL was also proposed by Quinlan, the father of ID3.

statistic has a  $\chi^2$  distribution with  $m - 1$  degrees of freedom. The higher the likelihood ratio is, the more likely that there is a *significant* difference in the number of correct predictions made by our rule in comparison with a “random guesser”. That is, the performance of our rule is not due to chance. The ratio helps identify rules with insignificant coverage.

CN2 uses entropy together with the likelihood ratio test, while FOIL’s information gain is used by RIPPER.

### Rule Pruning

*Learn\_One\_Rule* does not employ a test set when evaluating rules. Assessments of rule quality as described above are made with tuples from the original training data. Such assessment is optimistic since the rules will likely overfit the data. That is, the rules may perform well on the training data, but less well on subsequent data. To compensate for this, we can prune the rules. A rule is pruned by removing a conjunct (attribute test). We choose to prune a rule,  $R$ , if the pruned version of  $R$  has greater quality, as assessed on an independent set of tuples. As in decision tree pruning, we refer to this set as a *pruning set*. Various pruning strategies can be used, such as the pessimistic pruning approach described in the previous section. FOIL uses a simple yet effective method. Given a rule,  $R$ ,

$$FOIL\_Prune(R) = \frac{pos - neg}{pos + neg}, \quad (6.23)$$

where  $pos$  and  $neg$  are the number of positive and negative tuples covered by  $R$ , respectively. This value will increase with the accuracy of  $R$  on a pruning set. Therefore, if the *FOIL\_Pruned* value is higher for the pruned version of  $R$ , then we prune  $R$ . By convention, RIPPER starts with the most recently added conjunct when considering pruning. Conjuncts are pruned one at a time as long as this results in an improvement.

## 6.6 Classification by Backpropagation

“What is backpropagation?” Backpropagation is a neural network learning algorithm. The field of neural networks was originally kindled by psychologists and neurobiologists who sought to develop and test computational analogues of neurons. Roughly speaking, a **neural network** is a set of connected input/output units where each connection has a weight associated with it. During the learning phase, the network learns by adjusting the weights so as to be able to predict the correct class label of the input tuples. Neural network learning is also referred to as **connectionist learning** due to the connections between units.

Neural networks involve long training times and are therefore more suitable for applications where this is feasible. They require a number of parameters that are typically best determined empirically, such as the network topology or “structure.” Neural networks have been criticized for their poor interpretability. For example, it is difficult for humans to interpret the symbolic meaning behind the learned weights and of “hidden units” in the network. These features initially made neural networks less desirable for data mining.

Advantages of neural networks, however, include their high tolerance to noisy data as well as their ability to classify patterns on which they have not been trained. They can be used when you may have little knowledge of the relationships between attributes and classes. They are well-suited for continuous-valued inputs *and* outputs, unlike most decision tree algorithms. They have been successful on a wide array of real-world data, ranging from handwritten character recognition, pathology and laboratory medicine, to training a computer to pronounce English text. Neural network algorithms are inherently parallel; parallelization techniques can be used to speed up the computation process. In addition, several techniques have recently been developed for the extraction of rules from trained neural networks. These factors contribute towards the usefulness of neural networks for classification and prediction in data mining.

There are many different kinds of neural networks and neural network algorithms. The most popular neural network algorithm is *backpropagation*, which gained reputation in the 1980s. In Section 6.6.1 you will learn about multilayer feed-forward networks, the type of neural network on which the backpropagation algorithm performs.

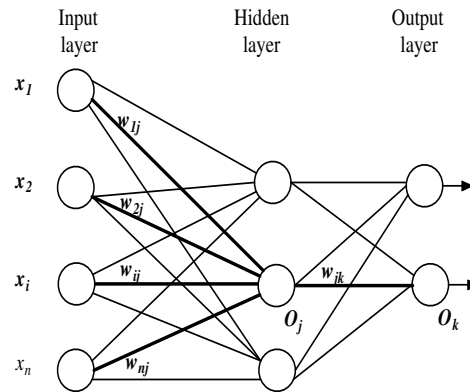


Figure 6.15: A multilayer feed-forward neural network:

Section 6.6.2 discusses defining a network topology. The backpropagation algorithm is described in Section 6.6.3. Rule extraction from trained neural networks is discussed in Section 6.6.4.

### 6.6.1 A Multilayer Feed-Forward Neural Network

The backpropagation algorithm performs learning on a *multilayer feed-forward* neural network. It iteratively learns a set of weights for prediction of the class label of tuples. A **multilayer feed-forward** neural network consists of an *input layer*, one or more *hidden layers*, and an *output layer*. An example of a multilayer feed-forward network is shown in Figure 6.15.

Each layer is made up of units. The inputs to the network correspond to the attributes measured for each training tuple. The inputs are fed simultaneously into the units making up the **input layer**. These inputs pass through the input layer, and are then weighted and fed simultaneously to a second layer of “neuronlike” units, known as a **hidden layer**. The outputs of the hidden layer units can be input to another hidden layer, and so on. The number of hidden layers is arbitrary, although in practice, usually only one is used. The weighted outputs of the last hidden layer are input to units making up the **output layer**, which emits the network’s prediction for given tuples.

The units in the input layer are called **input units**. The units in the hidden layers and output layer are sometimes referred to as **neuromodes**, due to their symbolic biological basis, or as **output units**. The multilayer neural network shown in Figure 6.15 has two layers of output units. Therefore, we say that it is a **two-layer** neural network. (The input layer is not counted since it serves only to pass the input values to the next layer.) Similarly, a network containing two hidden layers is called a *three-layer* neural network, and so on. The network is **feed-forward** in that none of the weights cycles back to an input unit or to an output unit of a previous layer. It is **fully connected** in that each unit provides input to each unit in the next forward layer.

Each output unit takes, as input, a weighted sum of the outputs from units in the previous layer (Figure 6.17). It applies a nonlinear (activation) function to the weighted input. Multilayer feed-forward neural networks are able to model the class prediction as a nonlinear combination of the inputs. From a statistical point of view, they perform nonlinear regression. *Multilayer feed-forward networks, given enough hidden units and enough training samples, can closely approximate any function.*

### 6.6.2 Defining a Network Topology

“How can I design the topology of the neural network?” Before training can begin, the user must decide on the network topology by specifying the number of units in the input layer, the number of hidden layers (if more than

one), the number of units in each hidden layer, and the number of units in the output layer.

Normalizing the input values for each attribute measured in the training tuples will help speed up the learning phase. Typically, input values are normalized so as to fall between 0.0 and 1.0. Discrete-valued attributes may be encoded such that there is one input unit per domain value. For example, if an attribute  $A$  has three possible or known values, namely  $\{a_0, a_1, a_2\}$ , then we may assign three input units to represent  $A$ . That is, we may have, say,  $I_0, I_1, I_2$  as input units. Each unit is initialized to 0. If  $A = a_0$ , then  $I_0$  is set to 1. If  $A = a_1$ ,  $I_1$  is set to 1, and so on. Neural networks can be used for both classification (to predict the class label of a given tuple) or prediction (to predict a continuous-valued output). For classification, one output unit may be used to represent two classes (where the value 1 represents one class, and the value 0 represents the other). If there are more than two classes, then one output unit per class is used.

There are no clear rules as to the “best” number of hidden layer units. Network design is a trial-and-error process and may affect the accuracy of the resulting trained network. The initial values of the weights may also affect the resulting accuracy. Once a network has been trained and its accuracy is not considered acceptable, it is common to repeat the training process with a different network topology or a different set of initial weights. Cross-validation techniques for accuracy estimation (described in Section 6.13) can be used to help decide when an acceptable network has been found. A number of automated techniques have been proposed that search for a “good” network structure. These typically use a hill-climbing approach that starts with an initial structure that is selectively modified.

### 6.6.3 Backpropagation

“How does backpropagation work?” Backpropagation learns by iteratively processing a data set of training tuples, comparing the network’s prediction for each tuple with the actual known *target* value. The target value may be the known class label of the training tuple (for classification problems) or a continuous value (for prediction). For each training tuple, the weights are modified so as to minimize the mean squared error between the network’s prediction and the actual target value. These modifications are made in the “backwards” direction, that is, from the output layer, through each hidden layer down to the first hidden layer (hence the name *backpropagation*). Although it is not guaranteed, in general the weights will eventually converge, and the learning process stops. The algorithm is summarized in Figure 6.16. The steps involved are expressed in terms of inputs, outputs, and errors, and may seem awkward if this is your first look at neural network learning. However, once you become familiar with the process, you will see that each step is inherently simple. The steps are described below.

**Initialize the weights:** The weights in the network are initialized to small random numbers (e.g., ranging from  $-1.0$  to  $1.0$ , or  $-0.5$  to  $0.5$ ). Each unit has a *bias* associated with it, as explained below. The biases are similarly initialized to small random numbers.

Each training tuple,  $\mathbf{X}$ , is processed by the following steps.

**Propagate the inputs forward:** First, the training tuple is fed to the input layer of the network. The inputs pass through the input units, unchanged. That is, for an input unit,  $j$ , its output,  $O_j$  is equal to its input value,  $I_j$ . Next, the net input and output of each unit in the hidden and output layers are computed. The net input to a unit in the hidden or output layers is computed as a linear combination of its inputs. To help illustrate this, a hidden layer or output layer unit is shown in Figure 6.17. Each such unit has a number of inputs to it that are, in fact, the outputs of the units connected to it in the previous layer. Each connection has a weight. To compute the net input to the unit, each input connected to the unit is multiplied by its corresponding weight, and this is summed. Given a unit  $j$  in a hidden or output layer, the net input,  $I_j$ , to unit  $j$  is

$$I_j = \sum_i w_{ij} O_i + \theta_j, \quad (6.24)$$

where  $w_{ij}$  is the weight of the connection from unit  $i$  in the previous layer to unit  $j$ ;  $O_i$  is the output of unit  $i$  from the previous layer; and  $\theta_j$  is the **bias** of the unit. The bias acts as a threshold in that it serves to vary the activity of the unit.

**Algorithm: Backpropagation.** Neural network learning for classification or prediction, using the backpropagation algorithm.

**Input:**

- $D$ , a data set consisting of the training tuples and their associated target values;
- $l$ , the learning rate;
- *network*, a multilayer feed-forward network.

**Output:** A trained neural network.

**Method:**

```

(1) Initialize all weights and biases in network;
(2) while terminating condition is not satisfied {
(3)   for each training tuple  $\mathbf{X}$  in  $D$  {
(4)     // Propagate the inputs forward:
(5)     for each input layer unit  $j$  {
(6)        $O_j = I_j$ ; // output of an input unit is its actual input value
(7)     }
(8)     for each hidden or output layer unit  $j$  {
(9)        $I_j = \sum_i w_{ij} O_i + \theta_j$ ; // compute the net input of unit  $j$  with respect to the previous layer,  $i$ 
(10)       $O_j = \frac{1}{1+e^{-I_j}}$ ; } // compute the output of each unit  $j$ 
(11)    // Backpropagate the errors:
(12)    for each unit  $j$  in the output layer
(13)       $Err_j = O_j(1 - O_j)(T_j - O_j)$ ; // compute the error
(14)    for each unit  $j$  in the hidden layers, from the last to the first hidden layer
(15)       $Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk}$ ; // compute the error with respect to the next higher layer,  $k$ 
(16)    for each weight  $w_{ij}$  in network {
(17)       $\Delta w_{ij} = (l) Err_j O_i$ ; // weight increment
(18)       $w_{ij} = w_{ij} + \Delta w_{ij}$ ; } // weight update
(19)    for each bias  $\theta_j$  in network {
(20)       $\Delta \theta_j = (l) Err_j$ ; // bias increment
(21)       $\theta_j = \theta_j + \Delta \theta_j$ ; } // bias update
  } }
```

Figure 6.16: Backpropagation algorithm.

Each unit in the hidden and output layers takes its net input and then applies an **activation** function to it, as illustrated in Figure 6.17. The function symbolizes the activation of the neuron represented by the unit. The **logistic**, or **sigmoid**, function is used. Given the net input  $I_j$  to unit  $j$ , then  $O_j$ , the output of unit  $j$ , is computed as

$$O_j = \frac{1}{1 + e^{-I_j}}. \quad (6.25)$$

This function is also referred to as a *squashing function*, since it maps a large input domain onto the smaller range of 0 to 1. The logistic function is nonlinear and differentiable, allowing the backpropagation algorithm to model classification problems that are linearly inseparable.

We compute the output values,  $O_j$ , for each hidden layer, up to and including the output layer, which gives the network's prediction. In practice, it is a good idea to cache (i.e., save) the intermediate output values at each unit as they are required again later, when backpropagating the error. This trick can substantially reduce the amount of computation required.

**Backpropagate the error:** The error is propagated backwards by updating the weights and biases to reflect the error of the network's prediction. For a unit  $j$  in the output layer, the error  $Err_j$  is computed by

$$Err_j = O_j(1 - O_j)(T_j - O_j) \quad (6.26)$$

where  $O_j$  is the actual output of unit  $j$ , and  $T_j$  is the known target value of the given training tuple. Note that  $O_j(1 - O_j)$  is the derivative of the logistic function.

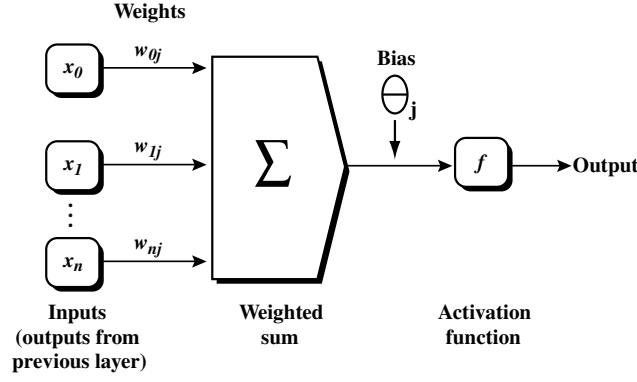


Figure 6.17: A hidden or output layer unit  $j$ : The inputs to unit  $j$  are outputs from the previous layer. These are multiplied by their corresponding weights in order to form a weighted sum, which is added to the bias associated with unit  $j$ . A nonlinear activation function is applied to the net input. (For ease of explanation, the inputs to unit  $j$  are labeled  $y_1, y_2, \dots, y_n$ . If unit  $j$  were in the first hidden layer, then these inputs would correspond to the input tuple  $(x_1, x_2, \dots, x_n)$ .)

To compute the error of a hidden layer unit  $j$ , the weighted sum of the errors of the units connected to unit  $j$  in the next layer are considered. The error of a hidden layer unit  $j$  is

$$Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk}, \quad (6.27)$$

where  $w_{jk}$  is the weight of the connection from unit  $j$  to a unit  $k$  in the next higher layer, and  $Err_k$  is the error of unit  $k$ .

The weights and biases are updated to reflect the propagated errors. Weights are updated by the following equations, where  $\Delta w_{ij}$  is the change in weight  $w_{ij}$ :

$$\Delta w_{ij} = (l)Err_j O_i \quad (6.28)$$

$$w_{ij} = w_{ij} + \Delta w_{ij} \quad (6.29)$$

“What is the ‘ $l$ ’ in Equation (6.28)?” The variable  $l$  is the **learning rate**, a constant typically having a value between 0.0 and 1.0. Backpropagation learns using a method of gradient descent to search for a set of weights that fits the training data so as to minimize the mean squared distance between the network’s class prediction and the known target value of the tuples.<sup>8</sup> The learning rate helps to avoid getting stuck at a local minimum in decision space (i.e., where the weights appear to converge, but are not the optimum solution) and encourages finding the global minimum. If the learning rate is too small, then learning will occur at a very slow pace. If the learning rate is too large, then oscillation between inadequate solutions may occur. A rule of thumb is to set the learning rate to  $1/t$ , where  $t$  is the number of iterations through the training set so far.

Biases are updated by the following equations below, where  $\Delta \theta_j$  is the change in bias  $\theta_j$ :

$$\Delta \theta_j = (l)Err_j \quad (6.30)$$

$$\theta_j = \theta_j + \Delta \theta_j \quad (6.31)$$

<sup>8</sup>A method of gradient descent was also used for training Bayesian belief networks, as described in Section 6.4.4.

Note that here we are updating the weights and biases after the presentation of each tuple. This is referred to as **case updating**. Alternatively, the weight and bias increments could be accumulated in variables, so that the weights and biases are updated after all of the tuples in the training set have been presented. This latter strategy is called **epoch updating**, where one iteration through the training set is an **epoch**. In theory, the mathematical derivation of backpropagation employs epoch updating, yet in practice, case updating is more common since it tends to yield more accurate results.

**Terminating condition:** Training stops when

- all  $\Delta w_{ij}$  in the previous epoch were so small as to be below some specified threshold, or
- the percentage of tuples misclassified in the previous epoch is below some threshold, or
- a prespecified number of epochs has expired.

In practice, several hundreds of thousands of epochs may be required before the weights will converge.

*“How efficient is backpropagation?”* The computational efficiency depends on the time spent training the network. Given  $|D|$  tuples and  $w$  weights, then each epoch requires  $O(|D| \times w)$  time. However, in the worst case scenario, the number of epochs can be exponential in  $n$ , the number of inputs. In practice, the time required for the networks to converge is highly variable. A number of techniques exist that help speed up the training time. For example, a technique known as *simulated annealing* can be used, which also assures convergence to a global optimum.

**Example 6.9 Sample calculations for learning by the backpropagation algorithm.** Figure 6.18 shows a multilayer feed-forward neural network. Let the learning rate be 0.9. The initial weight and bias values of the network are given in Table 6.3, along with the first training tuple,  $\mathbf{X}=(1,0,1)$ , whose class label is 1.

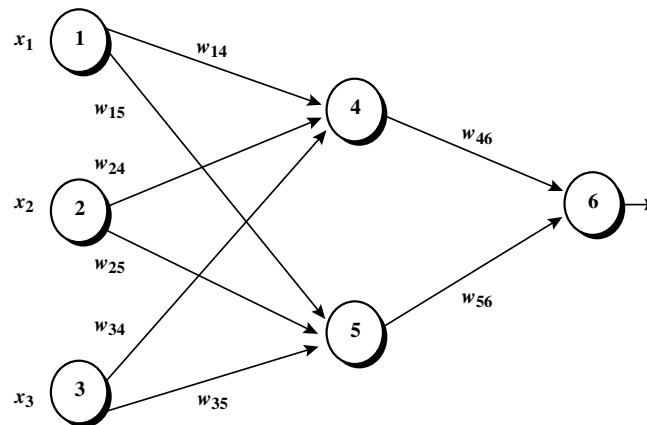


Figure 6.18: An example of a multilayer feed-forward neural network.

Table 6.3: Initial input, weight, and bias values.

$x_1$	$x_2$	$x_3$	$w_{14}$	$w_{15}$	$w_{24}$	$w_{25}$	$w_{34}$	$w_{35}$	$w_{46}$	$w_{56}$	$\theta_4$	$\theta_5$	$\theta_6$
1	0	1	0.2	-0.3	0.4	0.1	-0.5	0.2	-0.3	-0.2	-0.4	0.2	0.1

This example shows the calculations for backpropagation, given the first training tuple,  $\mathbf{X}$ . The tuple is fed into the network, and the net input and output of each unit are computed. These values are shown in Table 6.4. The error of each unit is computed and propagated backwards. The error values are shown in Table 6.5. The weight and bias updates are shown in Table 6.6. ■

Table 6.4: The net input and output calculations.

Unit $j$	Net input, $I_j$	Output, $O_j$
4	$0.2 + 0 - 0.5 - 0.4 = -0.7$	$1/(1 + e^{0.7}) = 0.332$
5	$-0.3 + 0 + 0.2 + 0.2 = 0.1$	$1/(1 + e^{-0.1}) = 0.525$
6	$(-0.3)(0.332) - (0.2)(0.525) + 0.1 = -0.105$	$1/(1 + e^{0.105}) = 0.474$

Table 6.5: Calculation of the error at each node.

Unit $j$	$Err_j$
6	$(0.474)(1 - 0.474)(1 - 0.474) = 0.1311$
5	$(0.525)(1 - 0.525)(0.1311)(-0.2) = -0.0065$
4	$(0.332)(1 - 0.332)(0.1311)(-0.3) = -0.0087$

Several variations and alternatives to the backpropagation algorithm have been proposed for classification in neural networks. These may involve the dynamic adjustment of the network topology and of the learning rate or other parameters, or the use of different error functions.

#### 6.6.4 Inside the Black Box: Backpropagation and Interpretability

*“Neural networks are like a black box. How can I ‘understand’ what the backpropagation network has learned?”* A major disadvantage of neural networks lies in their knowledge representation. Acquired knowledge in the form of a network of units connected by weighted links is difficult for humans to interpret. This factor has motivated research in extracting the knowledge embedded in trained neural networks and in representing that knowledge symbolically. Methods include extracting rules from networks and sensitivity analysis.

Various algorithms for the extraction of rules have been proposed. The methods typically impose restrictions regarding procedures used in training the given neural network, the network topology, and the discretization of input values.

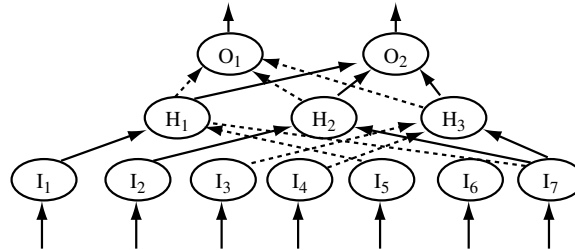
Fully connected networks are difficult to articulate. Hence, often the first step towards extracting rules from neural networks is **network pruning**. This consists of simplifying the network structure by removing weighted links that have the least effect on the trained network. For example, a weighted link may be deleted if such removal does not result in a decrease in the classification accuracy of the network.

Table 6.6: Calculations for weight and bias updating.

Weight or bias	New value
$w_{46}$	$-0.3 + (0.9)(0.1311)(0.332) = -0.261$
$w_{56}$	$-0.2 + (0.9)(0.1311)(0.525) = -0.138$
$w_{14}$	$0.2 + (0.9)(-0.0087)(1) = 0.192$
$w_{15}$	$-0.3 + (0.9)(-0.0065)(1) = -0.306$
$w_{24}$	$0.4 + (0.9)(-0.0087)(0) = 0.4$
$w_{25}$	$0.1 + (0.9)(-0.0065)(0) = 0.1$
$w_{34}$	$-0.5 + (0.9)(-0.0087)(1) = -0.508$
$w_{35}$	$0.2 + (0.9)(-0.0065)(1) = 0.194$
$\theta_6$	$0.1 + (0.9)(0.1311) = 0.218$
$\theta_5$	$0.2 + (0.9)(-0.0065) = 0.194$
$\theta_4$	$-0.4 + (0.9)(-0.0087) = -0.408$



Once the trained network has been pruned, some approaches will then perform link, unit, or activation value clustering. In one method, for example, clustering is used to find the set of common activation values for each hidden unit in a given trained two-layer neural network (Figure 6.19). The combinations of these activation values for each hidden unit are analyzed. Rules are derived relating combinations of activation values with corresponding output unit values. Similarly, the sets of input values and activation values are studied to derive rules describing the relationship between the input and hidden unit layers. Finally, the two sets of rules may be combined to form IF-THEN rules. Other algorithms may derive rules of other forms, including  $M$ -of- $N$  rules (where  $M$  out of a given  $N$  conditions in the rule antecedent must be true in order for the rule consequent to be applied), decision trees with  $M$ -of- $N$  tests, fuzzy rules, and finite automata.



Identify sets of common activation values for each hidden node, $H_i$ : for $H_1$ : (-1,0,1) for $H_2$ : (0,1) for $H_3$ : (-1,0,0.24,1)
Derive rules relating common activation values with output nodes, $O_j$ : IF ( $H_2 = 0$ and $H_3 = -1$ ) OR ( $H_1 = -1$ and $H_2 = 1$ and $H_3 = -1$ ) OR ( $H_1 = -1$ and $H_2 = 0$ and $H_3 = 0.24$ ) THEN $O_1 = 1, O_2 = 0$ ELSE $O_1 = 0, O_2 = 1$
Derive rules relating input nodes, $I_i$ , to output nodes, $O_j$ : IF ( $I_2 = 0$ AND $I_7 = 0$ ) THEN $H_2 = 0$ IF ( $I_4 = 1$ AND $I_6 = 1$ ) THEN $H_3 = -1$ IF ( $I_5 = 0$ ) THEN $H_3 = -1$
Obtain rules relating inputs and output classes: IF ( $I_2 = 0$ AND $I_7 = 0$ AND $I_4 = 1$ AND $I_6 = 1$ ) THEN class = 1 IF ( $I_2 = 0$ AND $I_7 = 0$ and $I_5 = 0$ ) THEN class = 1

Figure 6.19: Rules can be extracted from training neural networks. [TO EDITOR For consistency (for logic terms), please replace all 5 incidences of lower case “and” (i.e., in second box) by upper case “AND”. In addition, all  $O_1, H_1, I_1$ , etc. should be italicized.] Adapted from [LSL95].

**Sensitivity analysis** is used to assess the impact that a given input variable has on a network output. The input to the variable is varied while the remaining input variables are fixed at some value. Meanwhile, changes in the network output are monitored. The knowledge gained from this form of analysis can be represented in rules such as “*IF X decreases 5% THEN Y increases 8%*”.

## 6.7 Support Vector Machines

In this section, we study **Support Vector Machines**, a promising new method for the classification of both linear and nonlinear data. In a nutshell, a support vector machine (or **SVM**) is an algorithm that works as follows. It uses a nonlinear mapping to transform the original training data into a higher dimension. Within this new dimension, it searches for the linear optimal separating hyperplane (that is, a “decision boundary” separating the tuples of one class from another). With an appropriate nonlinear mapping to a sufficiently high dimension, data from two classes can always be separated by a hyperplane. The SVM finds this hyperplane using *support vectors* (“essential” training tuples) and *margins* (defined by the support vectors). We will delve more into these new concepts further below.

*“I’ve heard that SVMs have attracted a great deal of attention lately. Why?”* The first paper on support vector machines was presented in 1992 by Vladimir Vapnik and colleagues Bernhard Boser and Isabelle Guyon, although the groundwork for them has been around since the 1960s (including early work by Vapnik and Alexei Chervonenkis on statistical learning theory). Although the training time of even the fastest SVMs can be extremely slow, they are highly accurate owing to their ability to model complex nonlinear decision boundaries. They are much less prone to overfitting than other methods. The support vectors found also provide a compact description of the learned model. SVMs can be used for prediction as well as classification. They have been applied to a number of areas, including handwritten digit recognition, object recognition, speaker identification, as well as benchmark time series prediction tests.

### 6.7.1 The Case When the Data are Linearly Separable

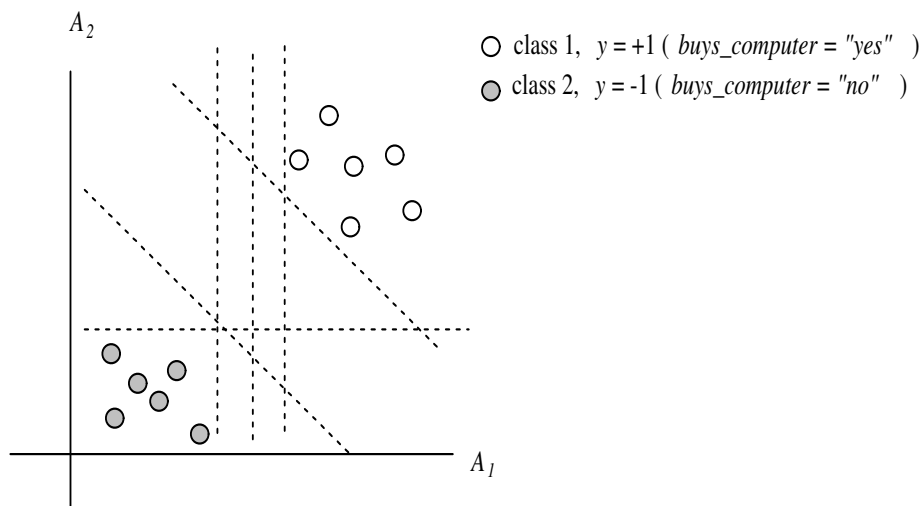


Figure 6.20: The 2-D training data are linearly separable. There are an infinite number of (possible) separating hyperplanes or “decision boundaries”. Which one is best?

To explain the mystery of SVMs, let’s first have a look at the simplest case—a two-class problem where the classes are linearly separable. Let the data set  $D$  be given as  $(\mathbf{X}_1, y_1), (\mathbf{X}_2, y_2), \dots, (\mathbf{X}_{|D|}, y_{|D|})$ , where  $\mathbf{X}_i$  is the set of training tuples with associated class labels,  $y_i$ . Each  $y_i$  can take one of two values, either  $+1$  or  $-1$  (i.e.,  $y_i \in \{+1, -1\}$ ), corresponding to the classes *buys\_computer* = *yes* and *buys\_computer* = *no*, respectively. To aid in visualization, let’s consider an example based on two input attributes,  $A_1$  and  $A_2$ , as shown in Figure 6.20. From the graph, we see that the 2-D data are **linearly separable** (or “linear”, for short) since a straight line can be drawn to separate all of the tuples of class  $+1$  from all of the tuples of class  $-1$ . There are an infinite number of separating lines that could be drawn. We want to find the “best” one, that is, one that (we hope) will

have the minimum classification error on previously unseen tuples. How can we find this best line? Note that if our data were 3-D (i.e., with three attributes), we would want to find the best separating *plane*. Generalizing to  $n$  dimensions, we want to find the best *hyperplane*. We will use the term “hyperplane” to refer to the decision boundary that we are seeking, regardless of the number of input attributes. So, in other words, how can we find the best hyperplane?

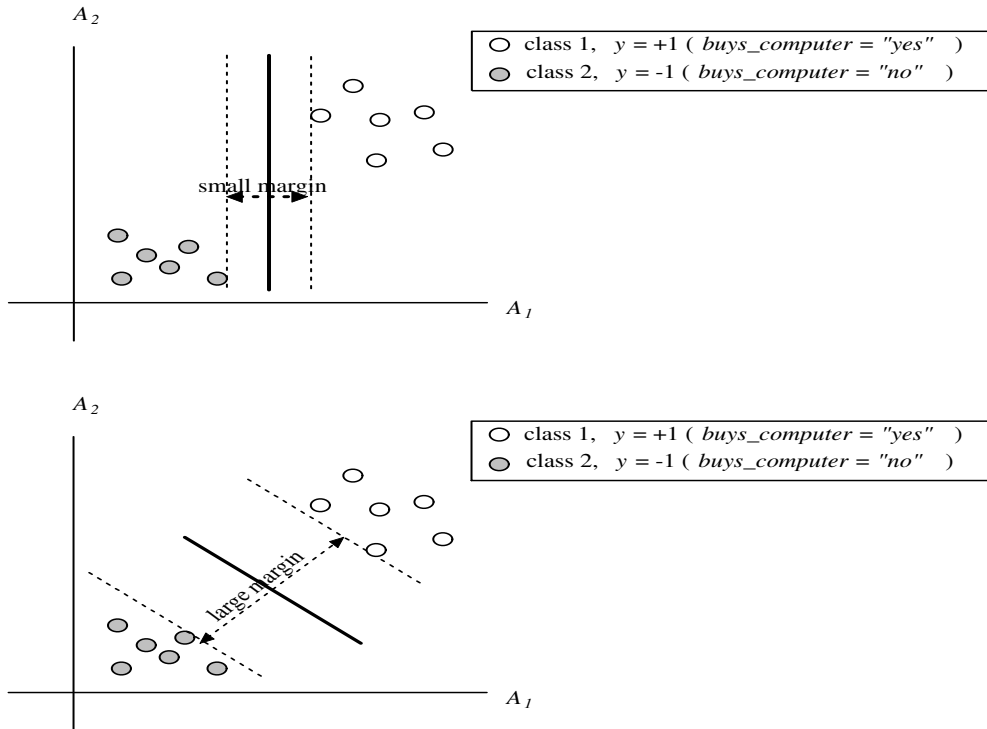


Figure 6.21: Here we see just two possible separating hyperplanes and their associated margins. Which one is better? The one with the larger margin should have greater generalization accuracy.

An SVM approaches this problem by searching for the **maximum marginal hyperplane**. Consider Figure 6.21, which shows two possible separating hyperplanes and their associated margins. Before we get into the definition of margins, let’s take an intuitive look at this figure. Both hyperplanes can correctly classify all of the given data tuples. Intuitively, however, we expect the hyperplane with the larger margin to be more accurate at classifying future data tuples than the hyperplane with the smaller margin. This is why (during the learning or training phase), the SVM searches for the hyperplane with the largest margin, that is, the *maximum marginal hyperplane* (MMH). The associated margin gives the largest separation between classes. Getting to an informal definition of **margin**, we can say that the shortest distance from a hyperplane to one side of its margin is equal to the shortest distance from the hyperplane to the other side of its margin, where the “sides” of the margin are parallel to the hyperplane. When dealing with the MMH, this distance is, in fact, the shortest distance from the MMH to the closest training tuple of either class.

A separating hyperplane can be written as

$$\mathbf{W} \cdot \mathbf{X} + b = 0, \quad (6.32)$$

where  $\mathbf{W}$  is a weight vector, namely,  $\mathbf{W} = \{w_1, w_2, \dots, w_n\}$ ;  $n$  is the number of attributes; and  $b$  is a scalar, often referred to as a bias. To aid in visualization, let’s consider two input attributes,  $A_1$  and  $A_2$ , as in Figure 6.21(b).

Training tuples are 2-D, e.g.,  $\mathbf{X} = (x_1, x_2)$ , where  $x_1$  and  $x_2$  are the values of attributes  $A_1$  and  $A_2$ , respectively, for  $\mathbf{X}$ . If we think of  $b$  as an additional weight,  $w_0$ , we can rewrite the above separating hyperplane as

$$w_0 + w_1x_1 + w_2x_2 = 0. \quad (6.33)$$

Thus, any point that lies above the separating hyperplane satisfies

$$w_0 + w_1x_1 + w_2x_2 > 0. \quad (6.34)$$

Similarly, any point that lies below the separating hyperplane satisfies

$$w_0 + w_1x_1 + w_2x_2 < 0. \quad (6.35)$$

The weights can be adjusted so that the hyperplanes defining the “sides” of the margin can be written as

$$H_1 : w_0 + w_1x_1 + w_2x_2 \geq 1 \text{ for } y_i = +1, \text{ and} \quad (6.36)$$

$$H_2 : w_0 + w_1x_1 + w_2x_2 \leq -1 \text{ for } y_i = -1. \quad (6.37)$$

That is, any tuple that falls on or above  $H_1$  belongs to class  $+1$ , and any tuple that falls on or below  $H_2$  belongs to class  $-1$ . Combining the two inequalities of Equations (6.36) and (6.37), we get

$$y_i(w_0 + w_1x_1 + w_2x_2) \geq 1, \forall i. \quad (6.38)$$

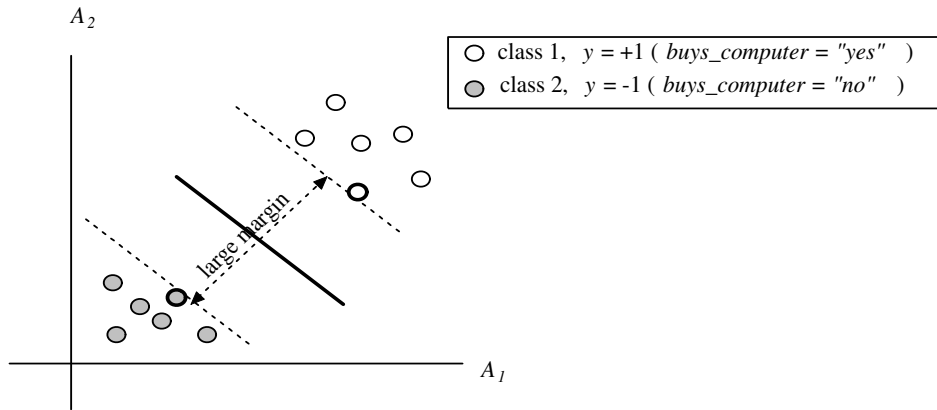


Figure 6.22: Support vectors. The SVM finds the maximum separating hyperplane, that is, the one with maximum distance between the nearest training tuples. The support vectors are shown with a thicker border.

Any training tuples that fall on hyperplanes  $H_1$  or  $H_2$  (i.e., the “sides” defining the margin) satisfy Equation (6.38) and are called **support vectors**. That is, they are equally close to the (separating) MMH. In Figure 6.22, the support vectors are shown encircled with a thicker border. Essentially, the support vectors are the most difficult tuples to classify and give the most information regarding classification.

From the above, we can obtain a formulae for the size of the maximal margin. The distance from the separating hyperplane to any point on  $H_1$  is  $\frac{1}{\|\mathbf{W}\|}$ , where  $\|\mathbf{W}\|$  is the Euclidean norm of  $\mathbf{W}$ , that is  $\sqrt{\mathbf{W} \cdot \mathbf{W}}$ . By definition, this is equal to the distance from any point on  $H_2$  to the separating hyperplane. Therefore, the maximal margin is  $\frac{2}{\|\mathbf{W}\|}$ .

“So, how does an SVM find the MMH and the support vectors?” Using some “fancy math tricks”, we can rewrite Equation (6.38) so that it becomes what is known as a constrained (convex) quadratic optimization problem. Such “fancy math tricks” are beyond the scope of this book. Advanced readers may be interested to note that the “tricks” involve rewriting Equation (6.38) using a Lagrangian formulation, and then solving for the solution using Karush-Kuhn-Tucker (KKT) conditions. Details can be found in references at the end of this chapter. If the data are small (say, less than 2,000 training tuples), any optimization software package for solving constrained convex quadratic problems can then be used to find the support vectors and MMH. For larger data, special and more efficient algorithms for training SVMs can be used instead, the details of which exceed the scope of this book. Once we’ve found the support vectors and MMH (note that the support vectors define the MMH!), we have a trained support vector machine. The MMH is a linear class boundary and so the corresponding SVM can be used to classify linearly separable data. We refer to such a trained SVM as a *linear SVM*.

“Once I’ve got a trained support vector machine, how do I use it to classify test (i.e., new) tuples?” Based on the Lagrangian formulation mentioned above, the maximum marginal hyperplane can be rewritten as the decision boundary

$$d(\mathbf{X}^T) = \sum_{i=1}^l y_i \alpha_i \mathbf{X}_i \mathbf{X}^T + b_0, \quad (6.39)$$

where  $y_i$  is the class label of support vector  $\mathbf{X}_i$ ,  $\mathbf{X}^T$  is a test tuple,  $\alpha_i$  and  $b_0$  are numeric parameters that were determined automatically by the optimization or SVM algorithm above, and  $l$  is the number of support vectors.

Interested readers may note that the  $\alpha_i$  are Lagrangian multipliers. For linearly separable data, the support vectors are a subset of the actual training tuples (although there will be a slight twist regarding this when dealing with nonlinearly separable data, as we shall see below!).

Given a test tuple,  $\mathbf{X}^T$ , we plug it into Equation (6.39), and then check to see the sign of the result. This tells us on which side of the hyperplane the test tuple falls. If the sign is positive, then  $\mathbf{X}^T$  falls on or above the MMH and so the SVM predicts that  $\mathbf{X}^T$  belongs to class +1 (representing *buys\_computer = yes*, in our case). If the sign is negative, then  $\mathbf{X}^T$  falls on or below the MMH and the class prediction is -1 (representing *buys\_computer = no*).

Notice that the Lagrangian formulation of our problem (Equation (6.39)) contains a dot product between support vector  $\mathbf{X}_i$  and test tuple  $\mathbf{X}^T$ . This will prove very useful for finding the MMH and support vectors for the case when the given data are nonlinearly separable, as described further below.

Before we move on to the nonlinear case, there are two more important things to note. The complexity of the learned classifier is characterized by the number of support vectors rather than the dimensionality of the data. Hence, SVMs tend to be less prone to overfitting than some other methods. The support vectors are the essential or critical training tuples—they lie closest to the decision boundary (MMH). If all other training tuples were removed and training were repeated, the same separating hyperplane would be found. Furthermore, the number of support vectors found can be used to compute an (upper) bound on the expected error rate of the SVM classifier, which is independent of the data dimensionality. An SVM with a small number of support vectors can have good generalization, even when the dimensionality of the data is high.

### 6.7.2 The Case When the Data Are Linearly Inseparable

In Section 6.7.1 above we learned about linear SVMs for classifying linearly separable data. But what if the data are not linearly separable, as in Figure 6.23? In such cases, there is no straight line that can be found that would

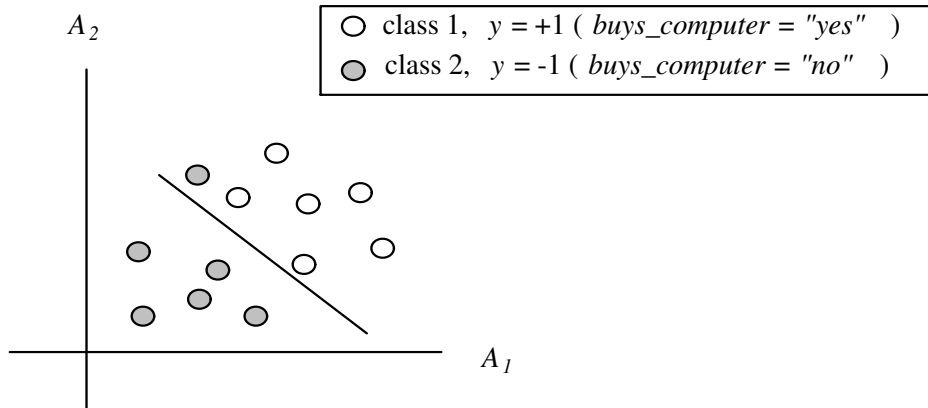


Figure 6.23: A simple 2-D case showing linearly inseparable data. Unlike the linear separable data of Figure 6.20, here it is not possible to draw a straight line to separate the classes. Instead, the decision boundary is nonlinear.

separate the classes. The linear SVMs we studied above would not be able to find a feasible solution here. Now what?

The good news is that the approach described above for linear SVMs can be extended to create *nonlinear SVMs* for the classification of *linearly inseparable data* (also called *nonlinearly separable data*, or *nonlinear data*, for short). Such SVMs are capable of finding nonlinear decision boundaries (that is, nonlinear hypersurfaces) in input space.

“So,” you may ask, “how can we extend the linear approach?” We obtain a nonlinear SVM by extending the approach for linear SVMs as follows. There are two main steps. In the first step, we transform the original input data into a higher dimensional space using a nonlinear mapping. Several common nonlinear mappings can be used in this step, as we will describe further below. Once the data have been transformed into the new higher space, the second step searches for a linear separating hyperplane in the new space. We again end up with a quadratic optimization problem that can be solved using the linear SVM formulation. The maximal marginal hyperplane found in the new space corresponds to a nonlinear separating hypersurface in the original space.

**Example 6.10 Nonlinear transformation of original input data into a higher dimensional space.**

Consider the following example. A 3D input vector  $\mathbf{X} = (x_1, x_2, x_3)$  is mapped into a 6D space,  $Z$ , using the mappings  $\phi_1(\mathbf{X}) = x_1, \phi_2(\mathbf{X}) = x_2, \phi_3(\mathbf{X}) = x_3, \phi_4(\mathbf{X}) = (x_1)^2, \phi_5(\mathbf{X}) = x_1x_2$ , and  $\phi_6(\mathbf{X}) = x_1x_3$ . A decision hyperplane in the new space is  $d(\mathbf{Z}) = \mathbf{W}\mathbf{Z} + b$ , where  $\mathbf{W}$  and  $\mathbf{Z}$  are vectors. This is linear. We solve for  $\mathbf{W}$  and  $b$  and then substitute back so that the linear decision hyperplane in the new ( $\mathbf{Z}$ ) space corresponds to a nonlinear second order polynomial in the original 3-D input space,

$$\begin{aligned} d(\mathbf{Z}) &= w_1x_1 + w_2x_2 + w_3x_3 + w_4(x_1)^2 + w_5x_1x_2 + w_6x_1x_3 + b \\ &= w_1z_1 + w_2z_2 + w_3z_3 + w_4z_4 + w_5z_5 + w_6z_6 + b \end{aligned}$$

But there are some problems. First, how do we choose the nonlinear mapping to a higher dimensional space? Second, the computation involved will be costly. Refer back to Equation (6.39) for the classification of a test tuple,  $\mathbf{X}^T$ . Given the test tuple, we have to compute its dot product with each and every one of the support vectors.<sup>9</sup> In training, we have to compute a similar dot product several times in order to find the maximal marginal hypersurface. This is especially expensive. Hence, the dot product computation required is very heavy and costly. We need another trick!

Luckily, there is another math trick that we can use. It so happens that in solving the quadratic optimization problem of the linear SVM (i.e., when searching for a linear SVM in the new higher dimensional space), the training

<sup>9</sup>The dot product of two vectors,  $\mathbf{X}^T = (x_1^T, x_2^T, \dots, x_n^T)$  and  $\mathbf{X}_i = (x_{i,1}, x_{i,2}, \dots, x_{i,n})$  is  $x_1^T x_{i,1} + x_2^T x_{i,2} + \dots + x_n^T x_{i,n}$ . Note that this involves one multiplication and one addition for each of the  $n$  dimensions.

tuples appear only in the form of dot products,  $\phi(\mathbf{X}_i) \cdot \phi(\mathbf{X}_j)$ , where  $\phi(\mathbf{X})$  is simply the nonlinear mapping function applied to transform the training tuples. Instead of computing the dot product on the transformed data tuples, it turns out that it is mathematically equivalent to instead apply a *kernel function*,  $K(\mathbf{X}_i, \mathbf{X}_j)$ , to the original input data. That is,

$$K(\mathbf{X}_i, \mathbf{X}_j) = \phi(\mathbf{X}_i) \cdot \phi(\mathbf{X}_j), \quad (6.40)$$

In other words, everywhere that  $\phi(\mathbf{X}_i) \cdot \phi(\mathbf{X}_j)$  appears in the training algorithm, we can replace it with  $K(\mathbf{X}_i, \mathbf{X}_j)$ . In this way, all calculations are made in the original input space, which is of potentially much lower dimensionality! We can safely avoid the mapping—it turns out that we don’t even have to know what the mapping is! We will talk more later about what kinds of functions can be used as kernel functions for this problem.

After applying this trick, we can then proceed to find a maximal separating hyperplane. The procedure is similar to that described in Section 6.7.1, although it involves placing a user-specified upper bound,  $C$ , on the Lagrange multipliers,  $\alpha_i$ , above. This upper bound is best determined experimentally.

“What are some of the kernel functions that could be used?” Properties of the kinds of kernel functions that could be used to replace the dot product scenario described above have been studied. Three admissible kernel functions include:

$$\text{Polynomial kernel of degree } h: K(\mathbf{X}_i, \mathbf{X}_j) = (\mathbf{X}_i \cdot \mathbf{X}_j + 1)^h \quad (6.41)$$

$$\text{Gaussian radial basis function kernel: } K(\mathbf{X}_i, \mathbf{X}_j) = e^{-\|\mathbf{X}_i - \mathbf{X}_j\|^2 / 2\sigma^2} \quad (6.42)$$

$$\text{Sigmoid kernel: } K(\mathbf{X}_i, \mathbf{X}_j) = \tanh(\kappa \mathbf{X}_i \cdot \mathbf{X}_j - \delta) \quad (6.43)$$

Each of these results in a different nonlinear classifier in (the original) input space. Neural network aficionados will be interested to note that the resulting decision hyperplanes found for nonlinear SVMs are the same type as those found by other well-known neural network classifiers. For instance, a SVM with a Gaussian radial basis function (RBF) gives the same decision hyperplane as a type of neural network known as a radial basis function (RBF) network. A SVM with a sigmoid kernel is equivalent to a simple 2-layer neural network known as a multilayer perceptron (with no hidden layers). There are no “golden rules” for determining which admissible kernel will result in the most accurate SVM. In practice, the kernel chosen does not generally make a large difference in resulting accuracy. SVM training always find a global solution, unlike neural networks such as backpropagation, where many local minima usually exist (Section 6.6.3).

So far, we have described linear and nonlinear SVMs for binary (i.e., two-class) classification. SVM classifiers can be combined for the multiclass case. A simple and effective approach, given  $m$  classes, trains  $m$  classifiers, one for each class (where classifier  $j$  learns to return a positive value for class  $j$  and a negative value for the rest). A test tuple is assigned the class corresponding to the largest positive distance.

Aside from classification, SVMs can also be designed for linear and nonlinear regression. Here, instead of learning to predict discrete class labels (like the  $y_i \in \{+1, -1\}$  above), SVMs for regression attempt to learn the input-output relationship between input training tuples,  $\mathbf{X}_i$ , and their corresponding continuous-valued outputs,  $y_i \in \mathcal{R}$ . An approach similar to SVMs for classification is followed. Additional user-specified parameters are required.

A major research goal regarding SVMs is to improve the speed in training and testing so that SVMs may become a more feasible option for very large data sets (e.g., of millions of support vectors). Other issues include determining the best kernel for a given data set, and finding more efficient methods for the multiclass case.

## 6.8 Associative Classification: Classification by Association Rule Analysis

Frequent patterns and their corresponding association or correlation rules characterize interesting relationships between attribute conditions and class labels, and thus have been recently used for effective classification. Association rules show strong associations between attribute-value pairs (or *items*) that occur frequently in a given data set. Association rules are commonly used to analyze the purchasing patterns of customers in a store. Such analysis is useful in many decision making processes, such as product placement, catalog design, and cross-marketing. The discovery of association rules is based on *frequent itemset mining*. Many methods for frequent itemset mining and the generation of association rules were described in Chapter 5. In this section, we look at *associative classification*, where association rules are generated and analyzed for use in classification. The general idea is that we can search for strong associations between frequent patterns (conjunctions of attribute-value pairs) and class labels. Since association rules explore highly confident associations among multiple attributes, this approach may overcome some constraints introduced by decision-tree induction, which considers only one attribute at a time. In many studies, associative classification has been found to be more accurate than some traditional classification methods, such as C4.5. In particular, we study three main methods: CBA, CMAR, and CPAR.

Before we begin, let's take a look at association rule mining, in general. Association rules are mined in a two-step process consisting of *frequent itemset mining*, followed by *rule generation*. The first step searches for patterns of attribute-value pairs that occur repeatedly in a data set, where each attribute-value pair is considered an *item*. The resulting attribute-value pairs form *frequent itemsets*. The second step analyzes the frequent itemsets in order to generate association rule. All association rules must satisfy certain criteria regarding their "accuracy" (or *confidence*) and the proportion of the data set that they actually represent (referred to as *support*). For example, the following is an association rule mined from a data set,  $D$ , shown with its confidence and support.

$$age = youth \wedge credit = OK \Rightarrow buys\_computer = yes \quad [support = 20\%, confidence = 93\%] \quad (6.44)$$

where " $\wedge$ " represents a logical "AND". We will say more about confidence and support in a minute.

More formally, let  $D$  be a data set of tuples. Each tuple in  $D$  is described by  $n$  attributes,  $A_1, A_2, \dots, A_n$ , and a class label attribute,  $A_{class}$ . All continuous attributes are discretized and treated as categorical attributes. An **item**,  $p$ , is an attribute-value pair of the form  $(A_i, v)$ , where  $A_i$  is an attribute taking a value,  $v$ . A data tuple  $\mathbf{X} = (x_1, x_2, \dots, x_n)$  satisfies an item,  $p = (A_i, v)$ , if and only if  $x_i = v$ , where  $x_i$  is the value of the  $i$ th attribute of  $\mathbf{X}$ . Association rules can have any number of items in the rule antecedent (left-hand side) and any number of items in the rule consequent (right-hand side). However, when mining association rules for use in classification, we are only interested in association rules of the form  $p_1 \wedge p_2 \wedge \dots \wedge p_l \rightarrow A_{class} = C$  where the rule antecedent is a conjunction of items,  $p_1, p_2, \dots, p_l$  ( $l \leq n$ ), associated with a class label,  $C$ . For a given rule,  $R$ , the percentage of tuples in  $D$  satisfying the rule antecedent that also have the class label  $C$  is called the **confidence** of  $R$ . For example, a confidence of 93% for Association Rule (6.44) means that 93% of the customers in  $D$  who are young and have an OK credit rating belong to the class  $buys\_computer = yes$ . The percentage of tuples in  $D$  satisfying the rule antecedent and having class label  $C$  is called the **support** of  $R$ . A support of 20% for Association Rule (6.44) means that 20% of the customers in  $D$  are young, have an OK credit rating, and belong to the class  $buys\_computer = yes$ .

Methods of associative classification differ primarily in the approach used for frequent itemset mining and in how the derived rules are analyzed and used for classification. We now look at some of the various methods for associative classification.

One of the earliest and simplest algorithms for associative classification is CBA (Classification-Based Association). CBA uses an iterative approach to frequent itemset mining, similar to that described for Apriori in Section 5.2.1, where multiple passes are made over the data and the derived frequent itemsets are used to generate and test longer itemsets. In general, the number of passes made is equal to the length of the longest rule found. The complete set of rules satisfying minimum confidence and minimum support thresholds are found and then analyzed for inclusion in the classifier. CBA uses a heuristic method to construct the classifier, where the rules are organized according to decreasing precedence based on their confidence and support. If a set of rules have the same antecedent, then the rule with the highest confidence is selected to represent the set. When classifying a



new tuple, the first rule satisfying the tuple is used to classify it. The classifier also contains a default rule, having lowest precedence, which specifies a default class for any new tuple that is not satisfied by any other rule in the classifier. In this way, the set of rules making up the classifier form a *decision list*. In general, CBA was empirically found to be more accurate than C4.5 on a good number of data sets.

CMAR (Classification based on Multiple Association Rules) differs from CBA in its strategy for frequent itemset mining and its construction of the classifier. It also employs several rule pruning strategies with the help of a tree-structure for efficient storage and retrieval of rules. CMAR adopts a variant of the *FP-growth* algorithm to find the complete set of rules satisfying the minimum confidence and minimum support thresholds. *FP-growth* was described in Section 5.2.4. *FP-growth* uses a tree structure, called an *FP-tree*, to register all of the frequent itemset information contained in the given data set,  $D$ . This requires only two scans of  $D$ . The frequent itemsets are then mined from the *FP-tree*. CMAR uses an enhanced *FP-tree* that maintains the distribution of class labels among tuples satisfying each frequent itemset. In this way, it is able to combine rule generation together with frequent itemset mining in a single step.

CMAR also employs a different tree structure to store and retrieve rules efficiently, and to prune rules based on confidence, correlation, and database coverage. Rule pruning strategies are triggered whenever a rule is inserted into the tree. For example, given two rules,  $R1$  and  $R2$ , if the antecedent of  $R1$  is more general than that of  $R2$  and  $conf(R1) \geq conf(R2)$ , then  $R2$  is pruned. The rationale is that highly specialized rules with low confidence can be pruned if a more generalized version with higher confidence exists. CMAR also prunes rules for which the rule antecedent and class are not positively correlated, based on a  $\chi^2$  test of statistical significance.

As a classifier, CMAR operates differently than CBA. Suppose that we are given a tuple  $X$  to classify and that only one rule satisfies or matches  $X$ .<sup>10</sup> This case is trivial—we simply assign the class label of the rule. Suppose, instead, that more than one rule satisfies  $X$ . These rules form a set,  $S$ . Which rule would we use to determine the class label of  $X$ ? CBA would assign the class label of the most confident rule among the rule set,  $S$ . CMAR instead considers multiple rules when making its class prediction. It divides the rules into groups according to class labels. All rules within a group share the same class label and each group has a distinct class label. CMAR uses a weighted  $\chi^2$  measure to find the “strongest” group of rules, based on the statistical correlation of rules within a group. It then assigns  $X$  the class label of the strongest group. In this way it considers multiple rules, rather than a single rule with highest confidence, when predicting the class label of a new tuple. On experiments, CMAR had slightly higher average accuracy in comparison with CBA. Its runtime, scalability, and use of memory was found to be more efficient.

CBA and CMAR adopt methods of frequent itemset mining to generate *candidate* association rules, which include all conjunctions of attribute-value pairs (items) satisfying minimum support. These rules are then examined, and a subset is chosen to represent the classifier. However, such methods generate quite a large number of rules. CPAR takes a different approach to rule generation, based on a rule generation algorithm for classification known as FOIL (First Order Inductive Learner) proposed in Quinlan and Cameron-Jones. FOIL builds rules to distinguish positive tuples (say, having class *buys\_computer* = *yes*) from negative tuples (such as *buys\_computer* = *no*). For multiclass problems, FOIL is applied to each class. That is, for a class,  $C$ , all tuples of class  $C$  are considered positive tuples, while the rest are considered negative tuples. Rules are generated to distinguish  $C$  tuples from all others. Each time a rule is generated, the positive samples it satisfies (or *covers*) are removed until all the positive tuples in the data set are covered. CPAR relaxes this step by allowing the covered tuples to remain under consideration, but reducing their ‘weight’. The process is repeated for each class. The resulting rules are merged to form the classifier rule set.

During classification, CPAR employs a somewhat different multiple rule strategy than CMAR. If more than one rule satisfies a new tuple,  $\mathbf{X}$ , the rules are divided into groups according to class, similar to CMAR. However, CPAR uses the best  $k$  rules of each group to predict the class label of  $\mathbf{X}$ , based on expected accuracy. By considering the best  $k$  rules rather than all of the rules of a group, it avoids the influence of lower ranked rules. The accuracy of CPAR on numerous data sets was shown to be close to that of CMAR. However, since CPAR generates far fewer rules than CMAR, it shows much better efficiency with large sets of training data.

In summary, associative classification offers a new alternative to classification schemes by building rules based

<sup>10</sup>If the antecedent of a rule satisfies or matches  $X$ , then we say that the rule satisfies  $X$ .

on conjunctions of attribute-value pairs that occur frequently in data.

## 6.9 Lazy Learners (or Learning from Your Neighbors)

The classification methods discussed so far in this chapter—decision tree induction, Bayesian classification, rule-based classification, classification by backpropagation, support vector machines, and classification based on association rule mining—are all examples of *eager learners*. **Eager learners**, when given a set of training tuples, will construct a generalization (i.e., classification) model before receiving new (e.g., test) tuples to classify. We can think of the learned model as being ready and eager to classify previously unseen tuples.

Imagine a contrasting “lazy” approach, in which the learner instead waits until the last minute before doing any model construction in order to classify a given test tuple. That is, when given a training tuple, a **lazy learner** simply stores it (or does only a little minor processing) and waits until it is given a test tuple. Only when it sees the test tuple does it perform generalization in order to classify the tuple based on its similarity to the stored training tuples. Unlike eager learning methods, lazy learners do less work when a training tuple is presented and more work when making a classification or prediction. Because lazy learners store the training tuples or “instances”, they are also referred to as **instance-based learners**, even though all learning is essentially based on instances.

When making a classification or prediction, lazy learners can be quite computationally expensive. They require efficient storage techniques and are well-suited to implementation on parallel hardware. They offer little explanation or insight into the structure of the data. Lazy learners, however, naturally support incremental learning. They are able to model complex decision spaces having hyper-polygonal shapes that may not be as easily describable by other learning algorithms (such as hyper-rectangular shapes modeled by decision trees). In this section, we look at two examples of lazy learners: *k-nearest-neighbor classifiers* and *case-based reasoning classifiers*.

### 6.9.1 *k*-Nearest Neighbor Classifiers

The *k*-nearest neighbor method was first described in the early 1950s. The method is labor intensive when given large training sets, and did not gain popularity until the 1960s when increased computing power became available. It has since been widely used in the area of pattern recognition.

Nearest neighbor classifiers are based on learning by analogy, that is, by comparing a given test tuple with training tuples that are similar to it. The training tuples are described by  $n$  attributes. Each tuple represents a point in an  $n$ -dimensional space. In this way, all of the training tuples are stored in an  $n$ -dimensional pattern space. When given an unknown tuple, a ***k*-nearest neighbor classifier** searches the pattern space for the  $k$  training tuples that are closest to the unknown tuple. These  $k$  training tuples are the  $k$  “nearest neighbors” of the unknown tuple.

“Closeness” is defined in terms of a distance metric, such as Euclidean distance. The Euclidean distance between two points or tuples, say,  $\mathbf{X}_1=(x_{11}, x_{12}, \dots, x_{1n})$  and  $\mathbf{X}_2=(x_{21}, x_{22}, \dots, x_{2n})$ , is

$$dist(\mathbf{X}_1, \mathbf{X}_2) = \sqrt{\sum_{i=1}^n (x_{1i} - x_{2i})^2}. \quad (6.45)$$

In other words, for each numeric attribute, we take the difference between the corresponding values of that attribute in tuple  $\mathbf{X}_1$  and in tuple  $\mathbf{X}_2$ , square this difference, and accumulate it to the square of the distance count,  $dist(\mathbf{X}_1, \mathbf{X}_2)$ . Typically, we normalize the values of each attribute, prior to using Equation (6.45). This helps prevent attributes with initially large ranges (such as *income*) from outweighing attributes with initially smaller ranges (such as binary attributes). Min-max normalization, for example, can be used to transform a value  $v$  of a numeric attribute  $A$  to  $v'$  in the range  $[0, 1]$  by computing

$$v' = \frac{v - \min_A}{\max_A - \min_A}, \quad (6.46)$$

where  $\min_A$  and  $\max_A$  are the minimum and maximum values of attribute  $A$ . Chapter 2 describes also other methods for data normalization as a form of data transformation.

For  $k$ -nearest neighbor classification, the unknown tuple is assigned the most common class among its  $k$  nearest neighbors. When  $k = 1$ , the unknown tuple is assigned the class of the training tuple that is closest to it in pattern space. Nearest neighbor classifiers can also be used for prediction, that is, to return a real-valued prediction for a given unknown tuple. In this case, the classifier returns the average value of the real-valued labels associated with the  $k$  nearest neighbors of the unknown tuple.

*“But how can distance be computed for attributes that not numeric, but categorical, such as color?”* The above discussion assumes that the attributes used to describe the tuples are all numeric. For categorical attributes, one simple method is to compare the corresponding value of the attribute in tuple  $\mathbf{X}_1$  with that in tuple  $\mathbf{X}_2$ . If the two are identical (e.g., tuples  $\mathbf{X}_1$  and  $\mathbf{X}_2$  both have the color blue), then the difference between the two is taken as 0. If the two are different (e.g., tuple  $\mathbf{X}_1$  is blue but tuple  $\mathbf{X}_2$  is red), then the difference is considered to be 1. Other methods may incorporate more sophisticated schemes for differential grading, e.g., where a larger difference score is assigned, say, for blue and white than for blue and black.

*“What about missing values?”* In general, if the value of a given attribute  $A$  is missing in tuple  $\mathbf{X}_1$  and/or in tuple  $\mathbf{X}_2$ , we assume the maximum possible difference. Suppose that each of the attributes have been mapped to the range  $[0, 1]$ . For categorical attributes, we take the difference value to be 1 if either one or both of the corresponding values of  $A$  are missing. If  $A$  is numeric and missing from both tuples  $\mathbf{X}_1$  and  $\mathbf{X}_2$ , then the difference is also taken to be 1. If only one value is missing and the other (which we’ll call  $v'$ ) is present and normalized, then we can take the difference to be either  $|1 - v'|$  or  $|0 - v'|$  (i.e.,  $1 - v'$  or  $v'$ ), whichever is greater.

*“How can I determine a good value for  $k$ , the number of neighbors?”* This can be determined experimentally. Starting with  $k = 1$ , we use a test set to estimate the error rate of the classifier. This process can be repeated, each time by incrementing  $k$  to allow for one more neighbor. The  $k$  giving the minimum error rate may be selected. In general, the larger the number of training tuples is, the larger the value of  $k$  will be (so that classification and prediction decisions can be based on a larger portion of the stored tuples). As the number of training tuples approaches infinity and  $k = 1$ , the error rate can be no worse than twice the Bayes error rate (the latter being the theoretical minimum). If  $k$  also approaches infinity, the error rate approaches the Bayes error rate.

Nearest neighbor classifiers use distance-based comparisons that intrinsically assign equal weight to each attribute. They therefore can suffer from poor accuracy when given noisy or irrelevant attributes. The method, however, has been modified to incorporate attribute weighting and the pruning of noisy data tuples. The choice of a distance metric can be critical. The Manhattan (city block) distance (Section 7.2.1) or other distance measurements, may also be used.

Nearest-neighbor classifiers can be extremely slow when classifying test tuples. If  $D$  is a training database of  $|D|$  tuples and  $k = 1$ , then  $O(|D|)$  comparisons are required in order to classify a given test tuple. By presorting and arranging the stored tuples into search trees, the number of comparisons can be reduced to  $O(\log(|D|))$ . Parallel implementation can reduce the running time to a constant, that is  $O(1)$ , which is independent of  $|D|$ . Other techniques to speed up classification time include the use of *partial distance* calculations and *editing* the stored tuples. In the **partial distance** method, we compute the distance based on a subset of the  $n$  attributes. If this distance exceeds a threshold then further computation for the given stored tuple is halted, and the process moves on to the next stored tuple. The **editing** method removes training tuples that prove “useless”. This method is also referred to as **pruning** or **condensing** since it reduces the total number of tuples stored.

### 6.9.2 Case-Based Reasoning

**Case-based reasoning** (CBR) classifiers use a database of problem solutions to solve new problems. Unlike nearest neighbor classifiers, which store training tuples as points in Euclidean space, CBR stores the tuples or “cases” for problem-solving as complex symbolic descriptions. Business applications of CBR include problem resolution for customer service help desks, where cases describe product-related diagnostic problems. CBR has also been applied to areas such as engineering and law, where cases are either technical designs or legal rulings, respectively. Medical education is another area for CBR, where patient case histories and treatments are used to

help diagnose and treat new patients.

When given a new case to classify, a case-based reasoner will first check if an identical training case exists. If one is found, then the accompanying solution to that case is returned. If no identical case is found, then the case-based reasoner will search for training cases having components that are similar to those of the new case. Conceptually, these training cases may be considered as neighbors of the new case. If cases are represented as graphs, this involves searching for subgraphs that are similar to subgraphs within the new case. The case-based reasoner tries to combine the solutions of the neighboring training cases in order to propose a solution for the new case. If incompatibilities arise with the individual solutions, then backtracking to search for other solutions may be necessary. The case-based reasoner may employ background knowledge and problem-solving strategies in order to propose a feasible combined solution.

Challenges in case-based reasoning include finding a good similarity metric (e.g., for matching subgraphs) and suitable methods for combining solutions. Other challenges include the selection of salient features for indexing training cases and the development of efficient indexing techniques. A trade-off between accuracy and efficiency evolves as the number of stored cases becomes very large. As this number increases, the case-based reasoner becomes more intelligent. After a certain point, however, the efficiency of the system will suffer as the time required to search for and process relevant cases increases. As with nearest neighbor classifiers, one solution is to edit the training database. Cases that are redundant or that have not proved useful may be discarded for the sake of improved performance. These decisions, however, are not clear-cut and their automation remains an active area of research.

## 6.10 Other Classification Methods

In this section, we give a brief description of a number of other classification methods. These methods include genetic algorithms, rough set, and fuzzy set approaches. In general, these methods are less commonly used for classification in commercial data mining systems than the methods described earlier in this chapter. However, these methods do show their strength in certain applications, and hence it is worthwhile to include them here.

### 6.10.1 Genetic Algorithms

**Genetic algorithms** attempt to incorporate ideas of natural evolution. In general, genetic learning starts as follows. An initial **population** is created consisting of randomly generated rules. Each rule can be represented by a string of bits. As a simple example, suppose that samples in a given training set are described by two Boolean attributes,  $A_1$  and  $A_2$ , and that there are two classes,  $C_1$  and  $C_2$ . The rule “*IF  $A_1$  AND NOT  $A_2$  THEN  $C_2$* ” can be encoded as the bit string “100”, where the two leftmost bits represent attributes  $A_1$  and  $A_2$ , respectively, and the rightmost bit represents the class. Similarly, the rule “*IF NOT  $A_1$  AND NOT  $A_2$  THEN  $C_1$* ” can be encoded as “001”. If an attribute has  $k$  values, where  $k > 2$ , then  $k$  bits may be used to encode the attribute’s values. Classes can be encoded in a similar fashion.

Based on the notion of survival of the fittest, a new population is formed to consist of the *fittest* rules in the current population, as well as *offspring* of these rules. Typically, the **fitness** of a rule is assessed by its classification accuracy on a set of training samples.

Offspring are created by applying genetic operators such as crossover and mutation. In **crossover**, substrings from pairs of rules are swapped to form new pairs of rules. In **mutation**, randomly selected bits in a rule’s string are inverted.

The process of generating new populations based on prior populations of rules continues until a population,  $P$ , “evolves” where each rule in  $P$  satisfies a prespecified fitness threshold.

Genetic algorithms are easily parallelizable and have been used for classification as well as other optimization problems. In data mining, they may be used to evaluate the fitness of other algorithms.

### 6.10.2 Rough Set Approach

Rough set theory can be used for classification to discover structural relationships within imprecise or noisy data. It applies to discrete-valued attributes. Continuous-valued attributes must therefore be discretized prior to its use.

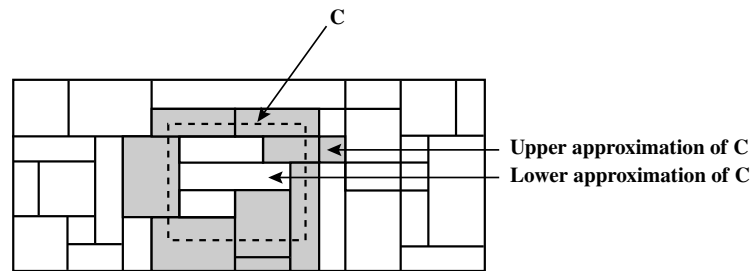


Figure 6.24: A rough set approximation of the set of tuples of the class  $C$  using lower and upper approximation sets of  $C$ . The rectangular regions represent equivalence classes.

Rough set theory is based on the establishment of **equivalence classes** within the given training data. All of the data tuples forming an equivalence class are indiscernible, that is, the samples are identical with respect to the attributes describing the data. Given real-world data, it is common that some classes cannot be distinguished in terms of the available attributes. Rough sets can be used to approximately or “roughly” define such classes. A rough set definition for a given class,  $C$ , is approximated by two sets—a **lower approximation** of  $C$  and an **upper approximation** of  $C$ . The lower approximation of  $C$  consists of all of the data tuples that, based on the knowledge of the attributes, are certain to belong to  $C$  without ambiguity. The upper approximation of  $C$  consists of all of the tuples that, based on the knowledge of the attributes, cannot be described as not belonging to  $C$ . The lower and upper approximations for a class  $C$  are shown in Figure 6.24, where each rectangular region represents an equivalence class. Decision rules can be generated for each class. Typically, a decision table is used to represent the rules.

Rough sets can also be used for attribute subset selection (or feature reduction, where attributes that do not contribute towards the classification of the given training data can be identified and removed) and relevance analysis (where the contribution or significance of each attribute is assessed with respect to the classification task). The problem of finding the minimal subsets (**reducts**) of attributes that can describe all of the concepts in the given data set is NP-hard. However, algorithms to reduce the computation intensity have been proposed. In one method, for example, a **discernibility matrix** is used that stores the differences between attribute values for each pair of data tuples. Rather than searching on the entire training set, the matrix is instead searched to detect redundant attributes.

### 6.10.3 Fuzzy Set Approaches

Rule-based systems for classification have the disadvantage that they involve sharp cutoffs for continuous attributes. For example, consider the following rule for customer credit application approval. The rule essentially says that applications for customers who have had a job for two or more years and who have a high income (i.e., of at least \$50K) are approved:

$$\text{IF } (\text{years\_employed} \geq 2) \text{ AND } (\text{income} \geq 50K) \text{ THEN } \text{credit} = \text{approved}. \quad (6.47)$$

By Rule (6.47), a customer who has had a job for at least two years will receive credit if her income is, say, \$50K, but not if it is \$49K. Such harsh thresholding may seem unfair. Instead, we can discretize *income* into categories such as  $\{\text{low\_income}, \text{medium\_income}, \text{high\_income}\}$ , and then apply **fuzzy logic** to allow “fuzzy” thresholds or boundaries to be defined for each category (Figure 6.25). Rather than having a precise cutoff between categories, fuzzy logic uses truth values between 0.0 and 1.0 to represent the degree of membership that a certain value has in

a given category. Each category then represents a **fuzzy set**. Hence, with fuzzy logic, we can capture the notion that an income of \$49K is, more or less, high, although not as high as an income of \$50K. Fuzzy logic systems typically provide graphical tools to assist users in converting attribute values to fuzzy truth values.

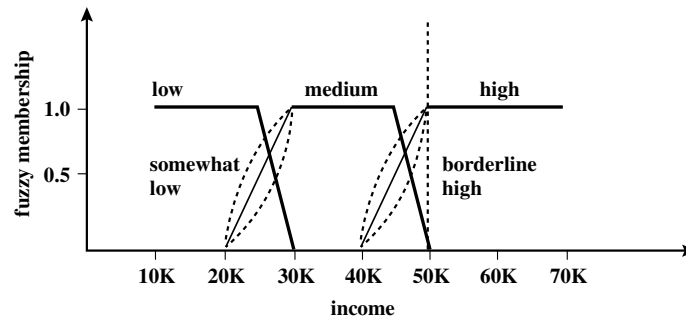


Figure 6.25: Fuzzy truth values for *income*, representing the degree of membership of *income* values with respect to the categories  $\{low, medium, high\}$ . Each category represents a fuzzy set. Note that a given income value,  $x$ , can have membership in more than one fuzzy set. The membership values of  $x$  in each fuzzy set do not have to total to 1. [TO EDITOR The figure must be redone: 1) The two upwards sloping lines (i.e., leading the ‘medium’ and ‘high’ curves) must be AS THICK as the rest of the lines defining ‘medium’ and ‘high’. 2) Remove the dashed lines (4 curved ones and 1 vertical one). 3) Show ticks and labels for the value 0 on both axes. 4) Remove labels ‘somewhat low’ and ‘borderline high’. 5) Please italicize *income*. ]

Fuzzy set theory is also known as **possibility theory**. It was proposed by Lotfi Zadeh in 1965 as an alternative to traditional two-value logic and probability theory. It lets us work at a high level of abstraction and offers a means for dealing with imprecise measurement of data. Most importantly, fuzzy set theory allows us to deal with vague or inexact facts. For example, being a member of a set of high incomes is inexact (e.g., if \$50K is high, then what about \$49K? Or \$48K?) Unlike the notion of traditional “crisp” sets where an element either belongs to a set  $S$  or its complement, in fuzzy set theory, elements can belong to more than one fuzzy set. For example, the income value \$49K belongs to both the *medium* and *high* fuzzy sets, but to differing degrees. Using fuzzy set notation and following Figure 6.25, this can be shown as

$$m_{medium\_income}(\$49K) = 0.15 \text{ and } m_{high\_income}(\$49K) = 0.96,$$

where  $m$  denotes the membership function, operating on the fuzzy sets of *medium\_income* and *high\_income*, respectively. In fuzzy set theory, membership values for a given element,  $x$ , (e.g., such as for \$49K) do not have to sum to 1. This is unlike traditional probability theory, which is constrained by a summation axiom.

Fuzzy set theory is useful for data mining systems performing rule-based classification. It provides operations for combining fuzzy measurements. Suppose that in addition to the fuzzy sets for *income*, we defined the fuzzy sets *junior\_employee* and *senior\_employee* for the attribute *years\_employed*. Suppose also that we have a rule that, say, tests *high\_income* and *senior\_employee* in the rule antecedent (IF part) for a given employee,  $x$ . If these two fuzzy measures are ANDed together, the minimum of their measure is taken as the measure of the rule. In other words,

$$m_{(high\_income \text{ AND } senior\_employee)}(x) = \min(m_{high\_income}(x), m_{senior\_employee}(x)).$$

This is akin to saying that a chain is as strong as its weakest link. If the two measures are ORed, the maximum of their measure is taken as the measure of the rule. In other words,

$$m_{(high\_income \text{ OR } senior\_employee)}(x) = \max(m_{high\_income}(x), m_{senior\_employee}(x)).$$

Intuitively, this is like saying that a rope is as strong as its strongest strand.

Given a tuple to classify, more than one fuzzy rule may apply. Each applicable rule contributes a vote for membership in the categories. Typically, the truth values for each predicted category are summed, and these sums are combined. Several procedures exist for translating the resulting fuzzy output into a *defuzzified* or crisp value

that is returned by the system.

Fuzzy logic systems have been used in numerous areas for classification, including market research, finance, health care, and environmental engineering.

## 6.11 Prediction

“What if we would like to predict a continuous value, rather than a categorical label?” Numeric prediction is the task of predicting continuous (or ordered) values for given input. For example, we may like to predict the salary of college graduates with 10 years of work experience, or the potential sales of a new product given its price. By far, the most widely used approach for numeric prediction (hereafter referred to as prediction) is **regression**, a statistical methodology that was developed by Sir Frances Galton (1822-1911), a mathematician who was also a cousin of Charles Darwin. In fact, many texts use the terms “regression” and “numeric prediction” synonymously. However, as we have seen, some classification techniques (such as backpropagation, support vector machines and  $k$ -nearest neighbor classifiers) can be adapted for prediction. In this section, we discuss the use of regression techniques for prediction.

Regression analysis can be used to model the relationship between one or more *independent* or **predictor** variables and a *dependent* or **response** variable (which is continuous-valued). In the context of data mining, the predictor variables are the attributes of interest describing the tuple (i.e., making up the attribute vector). In general, the values of the predictor variables are known. (Techniques exist for handling cases where such values may be missing.) The response variable is what we want to predict—it is what we referred to in Section 6.1 as the predicted attribute. Given a tuple described by predictor variables, we want to predict the associated value of the response variable. Regression analysis is a good choice when all of the predictor variables are continuous-valued as well. Many problems can be solved by *linear regression*, and even more can be tackled by applying transformations to the variables so that a nonlinear problem can be converted to a linear one. For reasons of space, we cannot give a fully detailed treatment of regression. Instead, this section provides an intuitive introduction to the topic. Section 6.11.1 discusses straight-line regression analysis (which involves a single predictor variable) and multiple linear regression analysis (which involves two or more predictor variables). Section 6.11.2 provides some pointers on dealing with nonlinear regression. Section 6.11.3 mentions other regression-based methods such as generalized linear models, Poisson regression, log-linear models, and regression trees.

Several software packages exist to solve regression problems. Examples include SAS (<http://www.sas.com>), SPSS (<http://www.spss.com>), and S-Plus (<http://www.insightful.com>). Another useful resource is the book *Numerical Recipes in C* by Press, Flannery, Teukolsky, and Vetterling and its associated source code.

### 6.11.1 Linear Regression

**Straight-line regression analysis** involves a response variable,  $y$ , and a single predictor variable,  $x$ . It is the simplest form of regression, and models  $y$  as a linear function of  $x$ . That is,

$$y = b + wx, \quad (6.48)$$

where the variance of  $y$  is assumed to be constant, and  $b$  and  $w$  are **regression coefficients** specifying the Y-intercept and slope of the line, respectively. The regression coefficients,  $w$  and  $b$ , can also be thought of as weights, so that we can equivalently write,

$$y = w_0 + w_1x. \quad (6.49)$$

These coefficients can be solved for by the **method of least squares**, which estimates the best-fitting straight line as the one that minimizes the error between the actual data and the estimate of the line. Let  $D$  be a training set consisting of values of predictor variable,  $x$ , for some population and their associated values for response variable,  $y$ .

The training set contains  $|D|$  data points of the form  $(x_1, y_1), (x_2, y_2), \dots, (x_{|D|}, y_{|D|})$ .<sup>11</sup> The regression coefficients can be estimated using this method with the following equations:

$$w_1 = \frac{\sum_{i=1}^{|D|} (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{|D|} (x_i - \bar{x})^2} \quad (6.50)$$

$$w_0 = \bar{y} - w_1 \bar{x} \quad (6.51)$$

where  $\bar{x}$  is the mean value of  $x_1, x_2, \dots, x_{|D|}$ , and  $\bar{y}$  is the mean value of  $y_1, y_2, \dots, y_{|D|}$ . The coefficients  $w_0$  and  $w_1$  often provide good approximations to otherwise complicated regression equations.

$x$ <i>years experience</i>	$y$ <i>salary (in \$1000s)</i>
3	30
8	57
9	64
13	72
3	36
6	43
11	59
21	90
1	20
16	83

Table 6.7: Salary data.

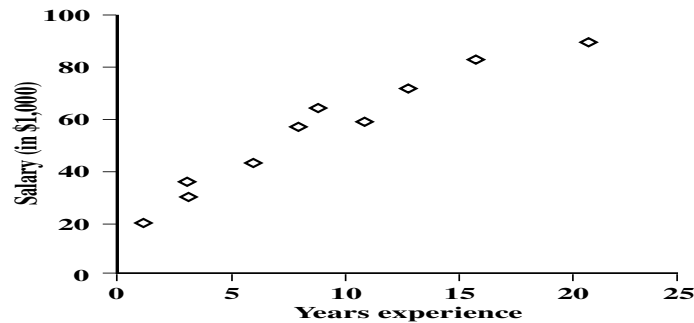


Figure 6.26: Plot of the data in Table 6.7 for Example 6.11. Although the points do not fall on a straight line, the overall pattern suggests a linear relationship between  $x$  (*years experience*) and  $y$  (*salary*). [TO EDITOR Italicize *years experience* and *salary*. Change vertical axis to read “*salary (in 1000s)*”, i.e., add “s” after “1000” to match heading in associated table.]

**Example 6.11 Straight-line regression using the method of least squares.** Table 6.7 shows a set of paired data where  $x$  is the number of years of work experience of a college graduate and  $y$  is the corresponding salary of the graduate. The 2-D data can be graphed on a *scatter plot*, as in Figure 6.26. The plot suggests a

<sup>11</sup>Note that earlier, we had used the notation  $(\mathbf{X}_i, y_i)$  to refer to training tuple  $i$  having associated class label  $y_i$ , where  $\mathbf{X}_i$  was an attribute (or feature) *vector* (that is,  $\mathbf{X}_i$  was described by more than one attribute). Here, however, we are dealing with just one predictor variable. Since the  $\mathbf{X}_i$  here are 1-dimensional, we use the notation  $x_i$  over  $\mathbf{X}_i$  in this case.



linear relationship between the two variables,  $x$  and  $y$ . We model the relationship that salary may be related to the number of years of work experience with the equation  $y = w_0 + w_1x$ .

Given the above data, we compute  $\bar{x} = 9.1$  and  $\bar{y} = 55.4$ . Substituting these values into Equations (6.50) and (6.51), we get

$$w_1 = \frac{(3 - 9.1)(30 - 55.4) + (8 - 9.1)(57 - 55.4) + \cdots + (16 - 9.1)(83 - 55.4)}{(3 - 9.1)^2 + (8 - 9.1)^2 + \cdots + (16 - 9.1)^2} = 3.5$$

$$w_0 = 55.4 - (3.5)(9.1) = 23.6$$

Thus, the equation of the least squares line is estimated by  $y = 23.6 + 3.5x$ . Using this equation, we can predict that the salary of a college graduate with, say, 10 years of experience is \$58.6K. ■

**Multiple linear regression** is an extension of straight-line regression so as to involve more than one predictor variable. It allows response variable  $y$  to be modeled as a linear function of, say,  $n$  predictor variables or attributes,  $A_1, A_2, \dots, A_n$ , describing a tuple,  $\mathbf{X}$ . (That is,  $\mathbf{X} = (x_1, x_2, \dots, x_n)$ .) Our training data set,  $D$ , contains data of the form  $(\mathbf{X}_1, y_1), (\mathbf{X}_2, y_2), \dots, (\mathbf{X}_{|D|}, y_{|D|})$ , where the  $\mathbf{X}_i$  are the  $n$ -dimensional training tuples with associated class labels,  $y_i$ . An example of a multiple linear regression model based on two predictor attributes or variables,  $A_1$  and  $A_2$ , is

$$y = w_0 + w_1x_1 + w_2x_2 \quad (6.52)$$

where  $x_1$  and  $x_2$  are the values of attributes  $A_1$  and  $A_2$ , respectively, in  $\mathbf{X}$ . The method of least squares shown above can be extended to solve for  $w_0$ ,  $w_1$ , and  $w_2$ . The equations, however, become long and are tedious to solve by hand. Multiple regression problems are instead commonly solved with the use of statistical software packages, such as SAS, SPSS, and S-Plus (see references above.)

### 6.11.2 Nonlinear Regression

*“How can we model data that does not show a linear dependence? For example, what if a given response variable and predictor variable have a relationship that may be modeled by a polynomial function?”* Think back to the straight-line linear regression case above where dependent response variable,  $y$ , is modeled as a linear function of a single independent predictor variable,  $x$ . What if we can get a more accurate model using a nonlinear model, such as a parabola or some other higher order polynomial? **Polynomial regression** is often of interest when there is just one predictor variable. It can be modeled by adding polynomial terms to the basic linear model. By applying transformations to the variables, we can convert the nonlinear model into a linear one that can then be solved by the method of least squares.

**Example 6.12 Transformation of a polynomial regression model to a linear regression model.** Consider a cubic polynomial relationship given by

$$y = w_0 + w_1x + w_2x^2 + w_3x^3. \quad (6.53)$$

To convert this equation to linear form, we define new variables:

$$x_1 = x \quad x_2 = x^2 \quad x_3 = x^3 \quad (6.54)$$

Equation (6.53) can then be converted to linear form by applying the above assignments, resulting in the equation  $y = w_0 + w_1x_1 + w_2x_2 + w_3x_3$ , which is easily solved by the method of least squares using software for regression analysis. Note that polynomial regression is a special case of multiple regression. That is, the addition of high-order terms like  $x^2$ ,  $x^3$ , and so on, which are simple functions of the single variable,  $x$ , can be considered equivalent to adding new independent variables. ■

In Exercise 15, you are asked to find the transformations required to convert a nonlinear model involving a power function into a linear regression model.

Some models are intractably nonlinear (such as the sum of exponential terms, for example) and cannot be converted to a linear model. For such cases, it may be possible to obtain least square estimates through extensive calculations on more complex formulae.

Various statistical measures exist for determining how well the proposed model can predict  $y$ . These are described in Section 6.12.2. Obviously, the greater the number of predictor attributes is, the slower the performance is. Prior to applying regression analysis, it is common to perform attribute subset selection (Section 2.5.2) to eliminate attributes that are unlikely to be good predictors for  $y$ . In general, regression analysis is quite accurate for prediction, except when the data contain outliers. Outliers are data points that are highly inconsistent with the remaining data (e.g., they may be way out of the expected value range). Outlier detection is discussed in Chapter 7. Such techniques must be used with caution, however, so as not to remove data points that are valid although they may vary greatly from the mean.

### 6.11.3 Other Regression-Based Methods

Linear regression is used to model continuous-valued functions. It is widely used, owing largely to its simplicity. “Can it also be used to predict categorical labels?” **Generalized linear models** represent the theoretical foundation on which linear regression can be applied to the modeling of categorical response variables. In generalized linear models, the variance of the response variable,  $y$ , is a function of the mean value of  $y$ , unlike in linear regression, where the variance of  $y$  is constant. Common types of generalized linear models include **logistic regression** and **Poisson regression**. Logistic regression models the probability of some event occurring as a linear function of a set of predictor variables. Count data frequently exhibit a Poisson distribution and are commonly modeled using Poisson regression.

**Log-linear models** approximate *discrete* multidimensional probability distributions. They may be used to estimate the probability value associated with data cube cells. For example, suppose we are given data for the attributes *city*, *item*, *year*, and *sales*. In the log-linear method, all attributes must be categorical; hence continuous-valued attributes (like *sales*) must first be discretized. The method can then be used to estimate the probability of each cell in the 4-D base cuboid for the given attributes, based on the 2-D cuboids for *city* and *item*, *city* and *year*, *city* and *sales*, and the 3-D cuboid for *item*, *year*, and *sales*. In this way, an iterative technique can be used to build higher-order data cubes from lower-order ones. The technique scales up well to allow for many dimensions. Aside from prediction, the log-linear model is useful for data compression (since the smaller-order cuboids together typically occupy less space than the base cuboid) and data smoothing (since cell estimates in the smaller-order cuboids are less subject to sampling variations than cell estimates in the base cuboid).

Decision tree induction can be adapted so as to predict continuous (ordered) values, rather than class labels. There are two main types of trees for prediction—*regression trees* and *model trees*. **Regression trees** were proposed as a component of the CART learning system. (Recall that the acronym CART stands for *Classification and Regression Trees*). Each regression tree leaf stores a continuous-valued prediction, which is actually the average value of the predicted attribute for the training tuples that reach the leaf. Since the terms “regression” and “numeric prediction” are used synonymously in statistics, the resulting trees were called “regression trees”, even though they did not use any regression equations. By contrast, in **model trees**, each leaf holds a regression model—a multivariate linear equation for the predicted attribute. Regression and model trees tend to be more accurate than linear regression when the data are not represented well by a simple linear model.

## 6.12 Accuracy and Error Measures

Now that you may have trained a classifier or predictor, there may be many questions going through your mind. For example, suppose you used data from previous sales to train a classifier to predict customer purchasing behavior. You would like an estimate of how accurately the classifier can predict the purchasing behavior of future customers,

that is, future customer data on which the classifier has not been trained. You may even have tried different methods to build more than one classifier (or predictor) and now wish to compare their accuracy. But what is accuracy? How can we estimate it? Are their strategies for increasing the accuracy of a learned model? These questions are addressed in the next few sections. Section 6.12.1 describes measures for computing classifier accuracy. Predictor error measures are given in Section 6.12.2. We can use these measures in techniques for accuracy estimation, such as the *holdout*, *random subsampling*, *k-fold cross-validation*, and *bootstrap* methods (Section 6.13). In Section 6.14, we'll learn some "tricks" for increasing model accuracy, such as *bagging* and *boosting*. Finally, Section 6.15 discusses **model selection** (i.e., choosing one classifier or predictor over another).

### 6.12.1 Classifier Accuracy Measures

Using training data to derive a classifier or predictor and then to estimate the accuracy of the resulting learned model can result in misleading overoptimistic estimates due to overspecialization of the learning algorithm to the data. (We'll say more on this in a moment!) Instead, accuracy is better measured on a test set, consisting of class-labeled tuples that were not used to train the model. The **accuracy** of a classifier on a given test set is the percentage of test set tuples that are correctly classified by the classifier. In the pattern recognition literature, this is also referred to as the overall **recognition rate** of the classifier, that is, it reflects how well the classifier recognizes tuples of the various classes.

We can also speak of the **error rate** or **misclassification rate** of a classifier,  $M$ , which is simply  $1 - \text{Acc}(M)$ , where  $\text{Acc}(M)$  is the accuracy of  $M$ . If we were to use the training set to estimate the error rate of a model, this quantity is known as the **resubstitution error**. This error estimate is optimistic of the true error rate (and similarly, the corresponding accuracy estimate is optimistic) because the model is not tested on any samples that it has not already seen.

<i>Classes</i>	<i>buys_computer = yes</i>	<i>buys_computer = no</i>	<i>Total</i>	<i>Recognition (%)</i>
<i>buys_computer = yes</i>	6,954	46	7,000	99.34
<i>buys_computer = no</i>	412	2,588	3,000	86.27
<i>Total</i>	7,366	2,634	10,000	95.52

Figure 6.27: A confusion matrix for the classes *buys\_computer = yes* and *buys\_computer = no* where an entry is row  $i$  and column  $j$  shows the number of tuples of class  $i$  that were labeled by the classifier as class  $j$ . Ideally, the non-diagonal entries should be zero or close to zero.

The *confusion matrix* is a useful tool for analyzing how well your classifier can recognize tuples of different classes. A confusion matrix for two classes is shown in Figure 6.27. Given  $m$  classes, a **confusion matrix** is a table of at least size  $m$  by  $m$ . An entry,  $CM_{i,j}$  in the first  $m$  rows and  $m$  columns indicates the number of tuples of class  $i$  that were labeled by the classifier as class  $j$ . For a classifier to have good accuracy, ideally most of the tuples would be represented along the diagonal of the confusion matrix, from entry  $CM_{1,1}$  to entry  $CM_{m,m}$ , with the rest of the entries being close to zero. The table may have additional rows or columns to provide totals or recognition rates per class.

	$C_1$	$C_2$
$C_1$	true positives	false negatives
$C_2$	false positives	true negatives

Figure 6.28: A confusion matrix for positive and negative tuples. [TO EDITOR Please display 'actual class' in bold to left of  $C_1$  and  $C_2$  of first column (these are the rows). Please display 'predicted class' in bold above the  $C_1$  and  $C_2$  of columns 2 and 3 (that is, the columns are the predicted class).]

Given two classes, we can talk in terms of **positive tuples** (tuples of the main class of interest, e.g., *buys\_computer*

= *yes*) versus **negative tuples** (e.g., *buys\_computer* = *no*)<sup>12</sup>. **True positives** refer to the positive tuples that were correctly labeled by the classifier, while **true negatives** are the negative tuples that were correctly labeled by the classifier. **False positives** are the negative tuples that were incorrectly labeled (e.g., tuples of class *buys\_computer* = *no* for which the classifier predicted *buys\_computer* = *yes*). Similarly, **false negatives** are the positive tuples that were incorrectly labeled (e.g., tuples of class *buys\_computer* = *yes* for which the classifier predicted *buys\_computer* = *no*). These terms are useful when analyzing a classifier's ability and are summarized in Figure 6.28.

*“Are there alternatives to the accuracy measure?”* Suppose that you have trained a classifier to classify medical data tuples as either “*cancer*” or “*not\_cancer*”. An accuracy rate of, say, 90% may make the classifier seem quite accurate, but what if only, say, 3–4% of the training tuples are actually “*cancer*”? Clearly, an accuracy rate of 90% may not be acceptable—the classifier could be correctly labelling only the “*not\_cancer*” tuples, for instance. Instead, we would like to be able to access how well the classifier can recognize “*cancer*” tuples (the positive tuples) and how well it can recognize “*not\_cancer*” tuples (the negative tuples). The **sensitivity** and **specificity** measures can be used, respectively, for this purpose. Sensitivity is also referred to as the *true positive (recognition) rate* (that is, the proportion of positive tuples that are correctly identified), while specificity is the *true negative rate* (that is, the proportion of negative tuples that are correctly identified). In addition, we may use **precision** to access the percentage of tuples labeled as “*cancer*” that actually are “*cancer*” tuples. These measures are defined as

$$\text{sensitivity} = \frac{t\_pos}{pos} \quad (6.55)$$

$$\text{specificity} = \frac{t\_neg}{neg} \quad (6.56)$$

$$\text{precision} = \frac{t\_pos}{(t\_pos + f\_pos)} \quad (6.57)$$

where *t\_pos* is the number of true positives (“*cancer*” tuples that were correctly classified as such), *pos* is the number of positive (“*cancer*”) tuples, *t\_neg* is the number of true negatives (“*not\_cancer*” tuples that were correctly classified as such), *neg* is the number of negative (“*not\_cancer*”) tuples, and *f\_pos* is the number of false positives (“*not\_cancer*” tuples that were incorrectly labeled as “*cancer*”). It can be shown that accuracy is a function of sensitivity and specificity:

$$\text{accuracy} = \text{sensitivity} \frac{pos}{(pos + neg)} + \text{specificity} \frac{neg}{(pos + neg)}. \quad (6.58)$$

The true positives, true negatives, false positives, and false negatives are also useful in assessing the costs and benefits (or risks and gains) associated with a classification model. The cost associated with a false negative (such as, incorrectly predicting that a cancerous patient is not cancerous) is far greater than that of a false positive (incorrectly yet conservatively labeling a noncancerous patient as cancerous). In such cases, we can outweigh one type of error over another by assigning a different cost to each. These cost may consider the danger to the patient, financial costs of resulting therapies, and other hospital costs. Similarly, the benefits associated with a true positive decision may be different than that of a true negative. Up to now, to compute classifier accuracy, we have assumed equal costs and essentially divide the sum of true positives and true negatives by the total number of test tuples. Alternatively, we can incorporate costs and benefits by instead computing the average cost (or benefit) per decision. Other applications involving cost-benefit analysis include loan application decisions and target marketing mailouts. For example, the cost of loaning to a defaulter greatly exceeds that of the lost business incurred by denying a loan to a nondefaulter. Similarly, in an application that tries to identify households that are likely to respond to mailouts of certain promotional material, the cost of mailouts to numerous households that do not respond may outweigh the cost of lost business from not mailing to households that would have responded. Other costs to consider in the overall analysis include the costs to collect the data and to develop the classification tool.

*“Are there other cases where accuracy may not be appropriate?”* In classification problems, it is commonly assumed that all tuples are uniquely classifiable, that is, that each training tuple can belong to only one class.

<sup>12</sup>In the machine learning and pattern recognition literature, these are referred to as *positive samples* and *negatives samples*, respectively.

Yet, owing to the wide diversity of data in large databases, it is not always reasonable to assume that all tuples are uniquely classifiable. Rather, it is more probable to assume that each tuple may belong to more than one class. How then can the accuracy of classifiers on large databases be measured? The accuracy measure is not appropriate, since it does not take into account the possibility of tuples belonging to more than one class.

Rather than returning a class label, it is useful to return a probability class distribution. Accuracy measures may then use a **second guess** heuristic, whereby a class prediction is judged as correct if it agrees with the first or second most probable class. Although this does take into consideration, to some degree, the nonunique classification of tuples, it is not a complete solution.

### 6.12.2 Predictor Error Measures

*“How can we measure predictor accuracy?”* Let  $D^T$  be a test set of the form  $(\mathbf{X}_1, y_1), (\mathbf{X}_2, y_2), \dots, (\mathbf{X}_d, y_d)$ , where the  $\mathbf{X}_i$  are the  $n$ -dimensional test tuples with associated known values,  $y_i$ , for a response variable,  $y$ , and  $d$  is the number of tuples in  $D^T$ . Since predictors return a continuous value rather than a categorical label, it is difficult to say *exactly* whether the predicted value,  $y'_i$ , for  $\mathbf{X}_i$  is correct. Instead of focussing on whether  $y'_i$  is an “exact” match with  $y_i$ , we instead look at how far off the predicted value is from the actual known value. **Loss functions** measure the error between  $y_i$  and the predicted value,  $y'_i$ . The most common loss functions are:

$$\text{Absolute error : } |y_i - y'_i| \quad (6.59)$$

$$\text{Squared error : } (y_i - y'_i)^2 \quad (6.60)$$

Based on the above, the **test error (rate)**, or **generalization error**, is the average loss over the test set. Thus, we get the following error rates.

$$\text{Mean absolute error : } \frac{\sum_{i=1}^d |y_i - y'_i|}{d} \quad (6.61)$$

$$\text{Mean squared-error : } \frac{\sum_{i=1}^d (y_i - y'_i)^2}{d} \quad (6.62)$$

The mean squared-error exaggerates the presence of outliers, while the mean absolute error does not. If we were to take the square root of the mean squared error, the resulting error measure is called the **root mean-squared error**. This is useful in that it allows the error measured to be of the same magnitude as the quantity being predicted.

Sometimes, we may want the error to be relative to what it would have been if we had just predicted  $\bar{y}$ , the mean value for  $y$  from the training data,  $D$ . That is, we can normalize the total loss by dividing by the total loss incurred from always predicting the mean. Relative measures of error include:

$$\text{Relative absolute error} = \frac{\sum_{i=1}^d |y_i - y'_i|}{\sum_{i=1}^d |y_i - \bar{y}|} \quad (6.63)$$

$$\text{Relative squared error :} = \frac{\sum_{i=1}^d (y_i - y'_i)^2}{\sum_{i=1}^d (y_i - \bar{y})^2} \quad (6.64)$$

where  $\bar{y}$  is the mean value of the  $y_i$ 's of the training data, that is  $\bar{y} = \frac{\sum_{i=1}^d y_i}{d}$ . We can take the root of the relative squared error to obtain the **root relative squared-error** so that the resulting error is of the same magnitude as the quantity predicted.

In practice, the choice of error measure does not greatly affect prediction model selection.

### 6.13 Evaluating the Accuracy of a Classifier or Predictor

How can we use the above measures to obtain a reliable estimate of classifier accuracy (or predictor accuracy in terms of error)? Holdout, random subsampling, cross-validation, and the bootstrap are common techniques for assessing accuracy based on randomly sampled partitions of the given data. The use of such techniques to estimate accuracy increases the overall computation time, yet is useful for model selection.

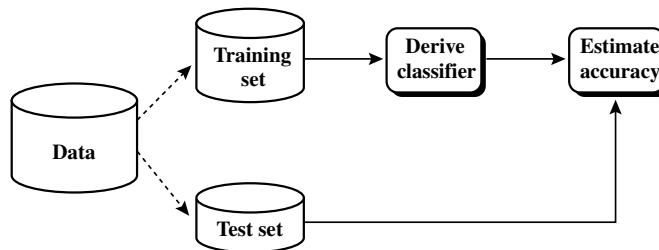


Figure 6.29: Estimating accuracy with the holdout method. [TO EDITOR PLEASE CHANGE “Derive classifier” TO “Derive model”.]

#### 6.13.1 Holdout Method and Random Subsampling

The **holdout** method is what we have alluded to so far in our discussions about accuracy. In this method, the given data are randomly partitioned into two independent sets, a *training set* and a *test set*. Typically, two thirds of the data are allocated to the training set, and the remaining one third is allocated to the test set. The training set is used to derive the model, whose accuracy is estimated with the test set (Figure 6.29). The estimate is pessimistic since only a portion of the initial data is used to derive the model.

**Random subsampling** is a variation of the holdout method in which the holdout method is repeated  $k$  times. The overall accuracy estimate is taken as the average of the accuracies obtained from each iteration. (For prediction, we can take the average of the predictor error rates.)

### 6.13.2 Cross-validation

In  **$k$ -fold cross-validation**, the initial data are randomly partitioned into  $k$  mutually exclusive subsets or “folds,”  $D_1, D_2, \dots, D_k$ , each of approximately equal size. Training and testing is performed  $k$  times. In iteration  $i$ , partition  $D_i$  is reserved as the test set, and the remaining partitions are collectively used to train the model. That is, in the first iteration, subsets  $D_2, \dots, D_k$  collectively serve as the training set in order to obtain a first model, which is tested on  $D_1$ ; the second iteration is trained on subsets  $D_1, D_3, \dots, D_k$  and tested on  $D_2$ ; and so on. [NEW Unlike the holdout and random subsampling methods above, here, each sample is used the same number of times for training and once for testing.] For classification, the accuracy estimate is the overall number of correct classifications from the  $k$  iterations, divided by the total number of tuples in the initial data. For prediction, the error estimate can be computed as the total loss from the  $k$  iterations, divided by the total number of initial tuples.

**Leave-one-out** is a special case of  $k$ -fold cross-validation where  $k$  set to the number of initial tuples. That is, only one sample is “left out” at a time for the test set. In **stratified cross-validation**, the folds are stratified so that the class distribution of the tuples in each fold is approximately the same as that in the initial data.

In general, stratified 10-fold cross-validation is recommended for estimating accuracy (even if computation power allows using more folds) due to its relatively low bias and variance.

### 6.13.3 Bootstrap

Unlike the accuracy estimation methods mentioned above, the **bootstrap method** samples the given training tuples uniformly *with replacement*. That is, each time a tuple is selected, it is equally likely to be selected again and re-added to the training set. For instance, imagine a machine that randomly selects tuples for our training set. In *sampling with replacement*, the machine is allowed to select the same tuple more than once.

There are several bootstrap methods. A commonly used one is the **.632 bootstrap**, which works as follows. Suppose we are given a data set of  $d$  tuples. The data set is sampled  $d$  times, with replacement, resulting in a *bootstrap sample* or training set of  $d$  samples. It is very likely that some of the original data tuples will occur more than once in this sample. The data tuples that did not make it into the training set end up forming the test set. Suppose we were to try this out several times. As it turns out, on average, 63.2% of the original data tuples will end up in the bootstrap, and the remaining 36.8% will form the test set (hence, the name, .632 bootstrap.)

“Where does the figure, 63.2%, come from?” Each tuple has a probability of  $1/d$  of being selected, so the probability of not being chosen is  $(1 - 1/d)$ . We have to select  $d$  times, so the probability that a tuple will not be chosen during this whole time is  $(1 - 1/d)^d$ . If  $d$  is large, the probability approaches  $e^{-1} = 0.368$ .<sup>13</sup> Thus, 36.8% of tuples will not be selected for training and thereby end up in the test set, and the remaining 63.2% will form the training set.

We can repeat the sampling procedure  $k$  times, where in each iteration, we use the current test set to obtain an accuracy estimate of the model obtained from the current bootstrap sample. The overall accuracy of the model is then estimated as

$$Acc(M) = \sum_{i=1}^k (0.632 \times Acc(M_i)_{test\_set} + 0.368 \times Acc(M_i)_{train\_set}), \quad (6.65)$$

where  $Acc(M_i)_{test\_set}$  is the accuracy of the model obtained with bootstrap sample  $i$  when it is applied to test set  $i$ .  $Acc(M_i)_{train\_set}$  is the accuracy of the model obtained with bootstrap sample  $i$  when it is applied to the original set of data tuples. The bootstrap method works well with small data sets.

<sup>13</sup> $e$  is the base of natural logarithms, that is,  $e = 2.718$ .

## 6.14 Ensemble Methods—Increasing the Accuracy

In Section 6.3.3, we saw how pruning can be applied to decision tree induction to help improve the accuracy of the resulting decision trees. Are there *general* strategies for improving classifier and predictor accuracy?

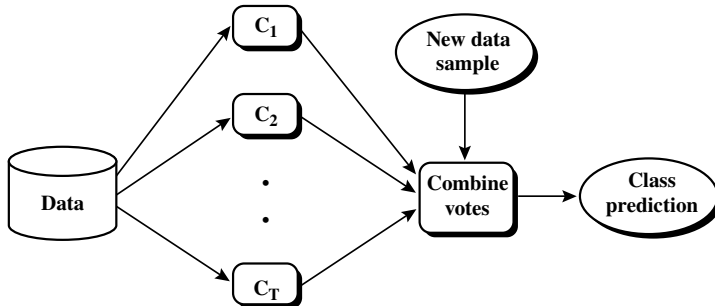


Figure 6.30: Increasing model accuracy: Bagging and boosting each generate a set of classification or prediction models,  $M_1, M_2, \dots, M_k$ . Voting strategies are used to combine the predictions for a given unknown tuple. [TO EDITOR PLEASE CHANGE  $C_1, C_2, \dots, C_T$  to  $M_1, M_2, \dots, M_k$ . ALSO CHANGE “Class prediction” TO “Prediction”.]

The answer is yes. *Bagging* and *boosting* are two such techniques (Figure 6.30). They are examples of **ensemble methods**, or methods that use a *combination* of models. Each combines a series of  $k$  learned models (classifiers or predictors),  $M_1, M_2, \dots, M_k$ , with the aim of creating an improved composite model,  $M^*$ . Both bagging and boosting can be used for classification as well as prediction.

### 6.14.1 Bagging

We first take an intuitive look at how bagging works as a method of increasing accuracy. For ease of explanation, we will assume at first that our model is a classifier. Suppose that you are a patient and would like to have a diagnosis made based on your symptoms. Instead of asking one doctor, you may choose to ask several. If a certain diagnosis occurs more than any of the others, you may choose this as the final or best diagnosis. That is, the final diagnosis is made based on a majority vote, where each doctor gets an equal vote. Now replace each doctor by a classifier, and you have the basic idea behind bagging. Intuitively, a majority vote made by a large group of doctors may be more reliable than a majority vote made by a small group.

Given a set,  $D$ , of  $d$  tuples, bagging works as follows. For iteration  $i$  ( $i = 1, 2, \dots, k$ ), a training set,  $D_i$ , of  $d$  tuples is sampled with replacement from the original set of tuples,  $D$ . Note that the term bagging stands for *bootstrap aggregation*. Each training set is a bootstrap sample, as described in Section 6.13.3. Since sampling with replacement is used, some of the original tuples of  $D$  may not be included in  $D_i$ , while others may occur more than once. A classifier model,  $M_i$ , is learned for each training set,  $D_i$ . To classify an unknown tuple,  $\mathbf{X}$ , each classifier,  $M_i$ , returns its class prediction, which counts as one vote. The bagged classifier,  $M^*$ , counts the votes and assigns the class with the most votes to  $\mathbf{X}$ . Bagging can be applied to the prediction of continuous values by taking the average value of each prediction for a given test tuple. The algorithm is summarized in Figure 6.31.

The bagged classifier often has significantly greater accuracy than a single classifier derived from  $D$ , the original training data. It will not be considerably worse and is more robust to the effects of noisy data. The increased accuracy occurs because the composite model reduces the variance of the individual classifiers. For prediction, it was theoretically proven that a bagged predictor will *always* have improved accuracy over a single predictor derived from  $D$ .



**Algorithm: Bagging.** The bagging algorithm—create an ensemble of models (classifiers or predictors) for a learning scheme where each model gives an equally-weighted prediction.

**Input:**

- $D$ , a set of  $d$  training tuples;
- $k$ , the number of models in the ensemble;
- a learning scheme (e.g., decision tree algorithm, backpropagation, etc.)

**Output:** A composite model,  $M^*$ .

**Method:**

- (1) for  $i = 1$  to  $k$  do // create  $k$  models:
- (2)     create bootstrap sample,  $D_i$ , by sampling  $D$  with replacement;
- (3)     use  $D_i$  to derive a model,  $M_i$ ;
- (4) endfor

**To use the composite model on a tuple,  $X$ :**

- (1) if classification then
- (2)     let each of the  $k$  models classify  $X$  and return the majority vote;
- (3) if prediction then
- (4)     let each of the  $k$  models predict a value for  $X$  and return the average predicted value;

Figure 6.31: Bagging.

### 6.14.2 Boosting

We now look at the ensemble method of boosting. As in the previous section, suppose that as a patient, you have certain symptoms. Instead of consulting one doctor, you choose to consult several. Suppose you assign weights to the “value” or worth of each doctor’s diagnosis, based on the accuracies of previous diagnoses they have made. The final diagnosis is then a combination of the weighted diagnoses. This is the essence behind boosting.

In **boosting**, weights are assigned to each training tuple. A series of  $k$  classifiers is iteratively learned. After a classifier  $M_i$  is learned, the weights are updated to allow the subsequent classifier,  $M_{i+1}$ , to “pay more attention” to the training tuples that were misclassified by  $M_i$ . The final boosted classifier,  $M^*$ , combines the votes of each individual classifier, where the weight of each classifier’s vote is a function of its accuracy. The boosting algorithm can be extended for the prediction of continuous values.

**Adaboost** is a popular boosting algorithm. Suppose we would like to boost the accuracy of some learning method. We are given  $D$ , a data set of  $d$  class-labeled tuples,  $(\mathbf{X}_1, y_1), (\mathbf{X}_2, y_2), \dots, (\mathbf{X}_d, y_d)$ , where  $y_i$  is the class label of tuple  $\mathbf{X}_i$ . Initially, **Adaboost** assigns each training tuple an equal weight of  $1/d$ . Generating  $k$  classifiers for the ensemble requires  $k$  rounds through the rest of the algorithm. In round  $i$ , the tuples from  $D$  are sampled to form a training set,  $D_i$ , of the same size. Each tuple’s chance of being selected is based on its weight. Sampling with replacement is used—the same tuple may be selected more than once. A classifier model,  $M_i$ , is derived from the training tuples of  $D_i$ . Its error is then calculated using  $D_i$  as a test set. The weights of the training tuples are then adjusted according to how they were classified. If a tuple was incorrectly classified, its weight is increased. If a tuple was correctly classified, its weight is decreased. A tuple’s weight reflects how hard it is to classify—the higher the weight, the more often it has been misclassified. These weights will be used to generate the training samples for the classifier of the next round. The basic idea is that when we build a classifier, we want it to focus more on the misclassified tuples of the previous round. Some classifiers may be better at classifying some “hard” tuples than others. In this way, we build a series of classifiers that complement each other. The algorithm is summarized in Figure 6.32.

**Algorithm: Adaboost.** A boosting algorithm—create an ensemble of classifiers. Each one gives a weighted vote.

**Input:**

- $D$ , a set of  $d$  class-labeled training tuples;
- $k$ , the number of rounds (one classifier is generated per round);
- a classification learning scheme.

**Output:** A composite model.

**Method:**

```

(1) initialize the weight of each tuple in  $D$  to  $1/d$ ;
(2) for  $i = 1$  to  $k$  do // for each round:
(3)     sample  $D$  with replacement according to the tuple weights to obtain  $D_i$ ;
(4)     use training set  $D_i$  to derive a model,  $M_i$ ;
(5)     compute  $error(M_i)$ , the error rate of  $M_i$  (Equation 6.66)
(6)     if  $error(M_i) > 0.5$  then
(7)         reinitialize the weights to  $1/d$ 
(8)         go back to step 3 and try again;
(9)     endif
(10)    for each tuple in  $D_i$  that was correctly classified do
(11)        multiply the weight of the tuple by  $error(M_i)/(1 - error(M_i))$ ; // update weights
(12)    endfor
(13) endfor

```

**To use the composite model to classify tuple,  $X$ :**

```

(1) initialize weight of each class to 0;
(2) for  $i = 1$  to  $k$  do // for each classifier:
(3)      $w_i = \log \frac{1 - error(M_i)}{error(M_i)}$ ; // weight of the classifier's vote
(4)      $c = M_i(X)$ ; // get class prediction for  $X$  from  $M_i$ 
(5)     add  $w_i$  to weight for class  $c$ 
(6) endfor
(7) return the class with the largest weight;

```

Figure 6.32: Adaboost, a boosting algorithm.

Now, let's look at some of the math that's involved in the algorithm. To compute the error rate of model  $M_i$ , we sum the weights of each of the tuples in  $D_i$  that  $M_i$  misclassified. That is,

$$error(M_i) = \sum_j^d w_j \times err(\mathbf{X}_j), \quad (6.66)$$

where  $err(\mathbf{X}_j)$  is the misclassification error of tuple  $\mathbf{X}_j$ : If the tuple was misclassified, then  $err(\mathbf{X}_j)$  is 1. Otherwise, it is 0. If the performance of classifier  $M_i$  is so poor that its error exceeds 0.5, then we abandon it. Instead, we try again by generating a new  $D_i$  training set, from which we derive a new  $M_i$ .

The error rate of  $M_i$  affects how the weights of the training tuples are updated. If a tuple in round  $i$  was correctly classified, its weight is multiplied by  $error(M_i)/(1 - error(M_i))$ . Once the weights of all of the correctly classified tuples are updated, the weights for all tuples (including the misclassified ones) are normalized so that their sum remains the same as it was before. To normalize a weight, we multiply it by the sum of the old weights, divided by the sum of the new weights. As a result, the weights of misclassified tuples are increased and the weights of correctly classified tuples are decreased, as described above.

*“Once boosting is complete, how is the ensemble of classifiers used to predict the class label of a tuple,  $\mathbf{X}$ ?”*

Unlike bagging, where each classifier was assigned an equal vote, boosting assigns a weight to each classifier's vote, based on how well the classifier performed. The lower a classifier's error rate, the more accurate it is, and therefore, the higher its weight for voting should be. The weight of classifier  $M_i$ 's vote is

$$\log \frac{1 - \text{error}(M_i)}{\text{error}(M_i)} \quad (6.67)$$

For each class,  $c$ , we sum the weights of each classifier that assigned class  $c$  to  $\mathbf{X}$ . The class with the highest sum is the “winner” and is returned as the class prediction for tuple  $\mathbf{X}$ .

“How does boosting compare with bagging?” Because of the way boosting focusses on the misclassified tuples, it risks overfitting the resulting composite model to such data. Therefore, sometimes the resulting “boosted” model may be less accurate than a single model derived from the same data. Bagging is less susceptible to model overfitting. While both can significantly improve accuracy in comparison to a single model, boosting tends to achieve greater accuracy.

## 6.15 Model Selection

Suppose that we have generated two models,  $M_1$  and  $M_2$  (for either classification or prediction), from our data. We have performed 10-fold cross validation to obtain a mean error rate for each. How can we determine which model is best? It may seem intuitive to select the model with the lowest error rate, however, the mean error rates are just *estimates* of error on the true population of future data cases. There can be considerable variance between error rates within any given 10-fold cross validation experiment. Although the mean error rates obtained for  $M_1$  and  $M_2$  may appear different, that difference may not be statistically significant. What if any difference between the two may just be attributed to chance? This section addresses these questions.

### 6.15.1 Estimating Confidence Intervals

To determine if there is any “real” difference in the mean error rates of two models, we need to employ a *test of statistical significance*. In addition, we would like to obtain some confidence limits for our mean error rates so that we can make statements like “any observed mean will not vary by +/- two standard errors 95% of the time for future samples”, or, “one model is better than the other by a margin of error of +/- 4%”.

What do we need in order to perform the statistical test? Suppose that for each model, we did 10-fold cross validation, say, 10 times, each time using a different 10-fold partitioning of the data. Each partitioning is independently drawn. We can average the 10 error rates obtained each for  $M_1$  and  $M_2$ , respectively, to obtain the mean error rate for each model. For a given model, the individual error rates calculated in the cross validations can be considered as different, independent samples from a probability distribution. [OLD: In this situation, the means of independently drawn samples][NEW: In general, they] follow a *t distribution with  $k-1$  degrees of freedom* where, here,  $k = 10$ . (This distribution looks very similar to a normal, or Gaussian, distribution even though the functions defining the two are quite different. Both are unimodal, symmetric, and bell-shaped.) This allows us to do hypothesis testing where the significance test used is the **t-test**, or **Student's t-test**. Our hypothesis is that the two models are the same, or in other words, that the difference in mean error rate between the two is zero. If we can reject this hypothesis (referred to as the *null hypothesis*), then we can conclude that the difference between the two models is statistically significant, in which case we can select the model with the lower error rate.

In data mining practice, we may often employ a single test set, that is, the same test set can be used for both  $M_1$  and  $M_2$ . In such cases, we do a **pairwise comparison** of the two models *for each* 10-fold cross validation round. That is, for the  $i$ th round of 10-fold cross validation, the same cross-validation partitioning is used to obtain an error rate for  $M_1$  and an error rate for  $M_2$ . Let  $\text{err}(M_1)_i$  (or  $\text{err}(M_2)_i$ ) be the error rate of model  $M_1$  (or  $M_2$ ) on round  $i$ . The error rates for  $M_1$  are averaged to obtain a mean error rate for  $M_1$ , denoted  $\overline{\text{err}}(M_1)$ . Similarly, we can obtain  $\overline{\text{err}}(M_2)$ . The variance of the difference between the two models is denoted  $\text{var}(M_1 - M_2)$ . The *t*-test computes the *t-statistic with  $k - 1$  degrees of freedom* for  $k$  samples. In our example we have  $k = 10$  since,

here, the  $k$  samples are our error rates obtained from 10 10-fold cross validations for each model. The  $t$ -statistic for pairwise comparison is computed as follows:

$$t = \frac{\overline{err}(M_1) - \overline{err}(M_2)}{\sqrt{var(M_1 - M_2)/k}}, \quad (6.68)$$

where

$$var(M_1 - M_2) = \frac{1}{k} \sum_{i=1}^k [err(M_1)_i - err(M_2)_i - (\overline{err}(M_1) - \overline{err}(M_2))]^2. \quad (6.69)$$

To determine whether or not  $M_1$  and  $M_2$  are significantly different, we compute  $t$  and select a significance level,  $sig$ . In practice, a significance level of 5% or 1% is typically used. We then consult a table for the  $t$  distribution, available in standard textbooks on statistics. This table is usually shown arranged by degrees of freedom as rows and significance levels as columns. Suppose we want to ascertain whether the difference between  $M_1$  and  $M_2$  is significantly different for 95% of the population, or  $sig = 5\%$  or 0.05. We need to find the  $t$  distribution value corresponding to  $k - 1$  degrees of freedom (or 9 degrees of freedom for our example) from the table. However, since the  $t$  distribution is symmetric, typically only the upper percentage points of the distribution are shown. Therefore, we look up the table value for  $z = sig/2$ , which in this case is 0.025, where  $z$  is also referred to as a confidence limit. If  $t > z$  or  $t < -z$ , then our value of  $t$  lies in the rejection region, within the tails of the distribution. This means that we can reject the null hypothesis that the means of  $M_1$  and  $M_2$  are the same and conclude that there is a statistically significant difference between the two models. Otherwise, if we cannot reject the null hypothesis, we then conclude that any difference between  $M_1$  and  $M_2$  can be attributed to chance.

If two test sets are available instead of a single test set, then a nonpaired version of the  $t$ -test is used, where the variance between the means of the two models is estimated as

$$var(M_1 - M_2) = \sqrt{\frac{var(M_1)}{k_1} + \frac{var(M_2)}{k_2}}, \quad (6.70)$$

and  $k_1$  and  $k_2$  are the number of cross-validation samples (in our case, 10-fold cross validation rounds) used for  $M_1$  and  $M_2$ , respectively. When consulting the table of  $t$  distribution, the number of degrees of freedom used is taken as the minimum number of degrees of the two models.

### 6.15.2 ROC Curves

**ROC curves** are a useful visual tool for comparing two classification models. The name ROC stands for *Receiver Operating Characteristic*. ROC curves come from signal detection theory that was developed during WWII for the analysis of radar images. An ROC curve shows the trade-off between the true positive rate or sensitivity (proportion of positive tuples that are correctly identified) and the false positive rate (proportion of negative tuples that are incorrectly identified as positive) for a given model. That is, given a two class problem, it allows us to visualize the trade-off between the rate at which the model can accurately recognize ‘yes’ cases versus the rate at which it mistakenly identifies ‘no’ cases as ‘yes’ for different “portions” of the test set. Any increase in the true positive rate occurs at the cost of an increase in the false positive rate. The area under the ROC curve is a measure of the accuracy of the model.

In order to plot an ROC curve for a given classification model,  $M$ , the model must be able to return a probability or ranking for the predicted class of each test tuple. That is, we need to rank the test tuples in decreasing order, where the one the classifier thinks is most likely to belong to the positive or ‘yes’ class appears at the top of the list. Naive Bayesian and backpropagation classifiers are appropriate, while others, such as decision tree classifiers, can easily be modified so as to return a class probability distribution for each prediction. The vertical axis of an ROC curve represents the true positive rate. The horizontal axis represents to the false positive rate. An ROC curve for  $M$  is plotted as follows. Starting at the bottom left hand corner (where the true positive rate and false positive rate are both 0), we check the actual class label of the tuple at the top of the list. If we have a true

positive (that is, a positive tuple that was correctly classified), then on the ROC curve, we move up and plot a point. If, instead, the tuple really belongs to the ‘no’ class, we have a false positive. On the ROC curve, we move right and plot a point. This process is repeated for each of the test tuples, each time moving up on the curve for a true positive or towards the right for a false positive.

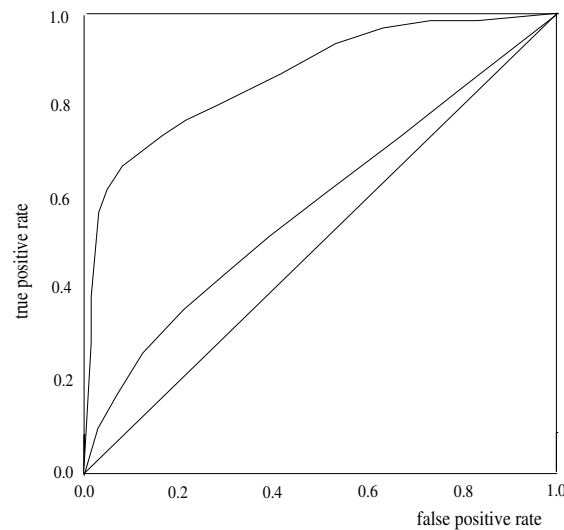


Figure 6.33: The ROC curves of two classification models.

Figure 6.33 shows the ROC curves of two classification models. The plot also shows a diagonal line. The closer the ROC curve of a model is to the diagonal line, the less accurate is the model. For every true positive of such a model, we are just as likely to encounter a false positive. If the model is really good, initially we are more likely to encounter true positives as we move down the ranked list. Thus, the curve would move steeply up from zero. Later, as we start to encounter fewer and fewer true positives, and more and more false positives, the curve eases off and becomes more horizontal.

To assess the accuracy of a model, we can measure the area under the curve. Several software packages are able to perform such calculation. The closer the area is to 0.5, the less accurate the corresponding model is. A model with perfect accuracy will have an area of 1.0.

## 6.16 Summary

- Classification and prediction are two forms of data analysis that can be used to extract models describing important data classes or to predict future data trends. While **classification** predicts categorical labels (classes), **prediction** models continuous-valued functions.
- Preprocessing of the data in preparation for classification and prediction can involve **data cleaning** to reduce noise or handle missing values, **relevance analysis** to remove irrelevant or redundant attributes, and **data transformation**, such as generalizing the data to higher-level concepts or normalizing the data.
- Predictive accuracy, computational speed, robustness, scalability, and interpretability are five **criteria** for the evaluation of classification and prediction methods.
- **ID3**, **C4.5**, and **CART** are greedy algorithms for the induction of **decision trees**. Each algorithm uses an attribute selection measure to select the attribute tested for each nonleaf node in the tree. **Pruning** algorithms attempt to improve accuracy by removing tree branches reflecting noise in the data. Early decision tree algorithms typically assume that the data are memory resident—a limitation to data mining

on large databases. Several scalable algorithms, such as **SLIQ**, **SPRINT**, and **RainForest**, have been proposed to address this issue.

- **Naive Bayesian classification** and **Bayesian belief networks** are based on Bayes theorem of posterior probability. Unlike naive Bayesian classification (which assumes class conditional independence), Bayesian belief networks allow class conditional independencies to be defined between subsets of variables.
- A **rule-based classifier** uses a set of IF-THEN rules for classification. Rules can be extracted from a decision tree or directly from the training data.
- **Backpropagation** is a neural network algorithm for classification that employs a method of gradient descent. It searches for a set of weights that can model the data so as to minimize the mean squared distance between the network's class prediction and the actual class label of data tuples. Rules may be extracted from trained neural networks in order to help improve the interpretability of the learned network.
- A **Support Vector Machine (SVM)** is an algorithm for the classification of both linear and nonlinear data. It transforms the original data in a higher dimension, from where it can find a hyperplane for separation of the data using essential training tuples called **support vectors**.
- **Association mining** techniques, which search for frequently occurring patterns in large databases, can be adapted for classification.
- Decision tree classifiers, Bayesian classifiers, classification by backpropagation, support vector machines, and classification based on association are all examples of **eager learners** in they use training tuples to construct a generalization model and in this way are ready for classifying new tuples. This contrasts with **lazy learners** or **instance-based** methods of classification, such as nearest neighbor classifiers and case-based reasoning classifiers, which store all of the training tuples in pattern space and wait until presented with a test tuple before performing generalization. Hence, lazy learners require efficient indexing techniques.
- In **genetic algorithms**, populations of rules “evolve” via operations of crossover and mutation until all rules within a population satisfy a specified threshold. **Rough set theory** can be used to approximately define classes that are not distinguishable based on the available attributes. **Fuzzy set** approaches replace “brittle” threshold cutoffs for continuous-valued attributes with degree of membership functions.
- Linear, nonlinear, and generalized linear models of **regression** can be used for prediction. Many nonlinear problems can be converted to linear problems by performing transformations on the predictor variables. Unlike decision trees, regression trees and model trees are used for prediction. In regression trees, each leaf stores a continuous-valued prediction. In model trees, each leaf holds a regression model.
- **Stratified  $k$ -fold cross-validation** is a recommended method for accuracy estimation. **Bagging** and **boosting** methods can be used to increase overall accuracy by learning and combining a series of individual models. For classifiers, **sensitivity**, **specificity**, and **precision** are useful alternatives to the accuracy measure, particularly when the main class of interest is in the minority. There are many measures of predictor error, such as the **mean squared-error**, the **mean absolute error**, the **relative squared-error**, and the **relative absolute error**. **Significance tests** and **ROC curves** are useful for model selection.
- There have been numerous comparisons of the different classification and prediction methods, and the matter remains a research topic. No single method has been found to be superior over all others for all data sets. Issues such as accuracy, training time, robustness, interpretability, and scalability must be considered and can involve trade-offs, further complicating the quest for an overall superior method. Empirical studies show that the accuracies of many algorithms are sufficiently similar that their differences are statistically insignificant, while training times may differ substantially. For classification, most neural network and statistical methods involving splines tend to be more computationally intensive than most decision tree methods.

## 6.17 Exercises

1. Briefly outline the major steps of *decision tree classification*.
2. Why is *tree pruning* useful in decision tree induction? What is a drawback of using a separate set of tuples to evaluate pruning?
3. Given a decision tree, you have the option of (a) converting the decision tree to rules and then pruning the resulting rules, or (b) pruning the decision tree and then converting the pruned tree to rules. What advantage does (a) have over (b)?
4. It is important to calculate the worst-case computational complexity of the decision tree algorithm. Given data set  $D$ , the number of attributes  $n$ , and the number of training tuples  $|D|$ , show that the computational cost of growing a tree is at most  $n \times |D| \times \log(|D|)$ .
5. Why is *naive Bayesian classification* called “naive”? Briefly outline the major ideas of naive Bayesian classification.
6. Given a 5 GB data set with 50 attributes (each containing 100 distinct values) and 512 MB of main memory in your laptop, outline an efficient method that constructs decision trees in such large data sets. Justify your answer by rough calculation of your main memory usage.
7. Rainforest is an interesting scalable algorithm for decision-tree induction. Develop a scalable naive Bayesian classification algorithm that requires just a single scan of the entire data set for most databases. Discuss whether such an algorithm can be refined to incorporate *boosting* to further enhance its classification accuracy.
8. Compare the advantages and disadvantages of *eager* classification (e.g., decision tree, Bayesian, neural network) versus *lazy* classification (e.g.,  $k$ -nearest neighbor, case-based reasoning).
9. Design an efficient method that performs effective naive Bayesian classification over an *infinite* data stream (i.e., you can scan the data stream only once). If we wanted to discover the *evolution* of such classification schemes (e.g., comparing the classification scheme at this moment with earlier schemes, such as one from a week ago), what modified design would you suggest?
10. What is *association-based classification*? Why is association-based classification able to achieve higher classification accuracy than a classical decision-tree method? Explain how association-based classification can be used for text document classification.
11. The following table consists of training data from an employee database. The data have been generalized. For example, “31 ... 35” for *age* represents the age range of 31 to 35. For a given row entry, *count* represents the number of data tuples having the values for *department*, *status*, *age*, and *salary* given in that row.

<i>department</i>	<i>status</i>	<i>age</i>	<i>salary</i>	<i>count</i>
sales	senior	31...35	46K...50K	30
sales	junior	26...30	26K...30K	40
sales	junior	31...35	31K...35K	40
systems	junior	21...25	46K...50K	20
systems	senior	31...35	66K...70K	5
systems	junior	26...30	46K...50K	3
systems	senior	41...45	66K...70K	3
marketing	senior	36...40	46K...50K	10
marketing	junior	31...35	41K...45K	4
secretary	senior	46...50	36K...40K	4
secretary	junior	26...30	26K...30K	6

Let *status* be the class label attribute.

- (a) How would you modify the ID3 algorithm to take into consideration the *count* of each generalized data tuple (i.e., of each row entry)?

- (b) Use your modified version of ID3 to construct a decision tree from the given data.
  - (c) Given a data tuple having the values “*systems*”, “*26...30*”, and “*46-50K*” for the attributes *department*, *age*, and *salary*, respectively, what would a naive Bayesian classification of the *status* for the tuple be?
  - (d) Design a multilayer feed-forward neural network for the given data. Label the nodes in the input and output layers.
  - (e) Using the multilayer feed-forward neural network obtained above, show the weight values after one iteration of the backpropagation algorithm given the training instance “(*sales, senior, 31...35, 46K...50K*)”. Indicate your initial weight values and biases, and the learning rate used.
12. The *support vector machine (SVM)* is a highly accurate classification method. However, SVM classifiers suffer from slow processing when training with a large set of data tuples. Discuss how to overcome this difficulty and develop a scalable SVM algorithm for efficient SVM classification in large datasets.
  13. Write an algorithm for *k-nearest neighbor classification* given  $k$  and  $n$ , the number of attributes describing each tuple.
  14. The following table shows the midterm and final exam grades obtained for students in a database course.

$x$	$y$
<i>Midterm exam</i>	<i>Final exam</i>
72	84
50	63
81	77
74	78
94	90
86	75
59	49
83	79
65	77
33	52
88	74
81	90

- (a) Plot the data. Do  $x$  and  $y$  seem to have a linear relationship?
  - (b) Use the *method of least squares* to find an equation for the prediction of a student’s final exam grade based on the student’s midterm grade in the course.
  - (c) Predict the final exam grade of a student who received an 86 on the midterm exam.
15. Some *nonlinear regression* models can be converted to linear models by applying transformations to the predictor variables. Show how the nonlinear regression equation  $y = \alpha X^\beta$  can be converted to a linear regression equation solvable by the method of least squares.
  16. What is *boosting*? State why it may improve the accuracy of decision tree induction.
  17. Show that accuracy is a function of *sensitivity* and *specificity*, that is, prove Equation (6.58).
  18. Suppose that we would like to select between two prediction models,  $M_1$  and  $M_2$ . We have performed 10 rounds of 10-fold cross validation on each model, where the same data partitioning in round  $i$  is used for both  $M_1$  and  $M_2$ . The error rates obtained for  $M_1$  are 30.5, 32.2, 20.7, 20.6, 31.0, 41.0, 27.7, 26.0, 21.5, 26.0. The error rates for  $M_2$  are 22.4, 14.5, 22.4, 19.6, 20.7, 20.4, 22.1, 19.4, 16.2, 35.0. Comment on whether one model is significantly better than the other considering a significance level of 1%.
  19. It is difficult to assess classification *accuracy* when individual data objects may belong to more than one class at a time. In such cases, comment on what criteria you would use to compare different classifiers modeled after the same data.



## 6.18 Bibliographic Notes

Classification from machine learning, statistics, and pattern recognition perspectives has been described in many books, such as Weiss and Kulikowski [WK91], Michie, Spiegelhalter, and Taylor [MST94], Russel and Norvig [RN95], Langley [Lan96], Mitchell [Mit97], Hastie, Tibshirani, and Friedman [HTF01], Duda, Hart, and Stork [DHS01], Alpaydin [Alp04], Tan, Steinbach, and Kumar [TSK05], and Witten and Frank [WF05]. Many of these books describe each of the basic methods of classification discussed in this chapter, as well as practical techniques for the evaluation of classifier performance. Edited collections containing seminal articles on machine learning can be found in Michalski, Carbonell, and Mitchell [MCM83, MCM86], Kodratoff and Michalski [KM90], Shavlik and Dietterich [SD90], and Michalski and Tecuci [MT94]. For a presentation of machine learning with respect to data mining applications, see Michalski, Bratko, and Kubat [MBK98].

The C4.5 algorithm is described in a book by Quinlan [Qui93]. The CART system is detailed in *Classification and Regression Trees* by Breiman, Friedman, Olshen, and Stone [BFOS84]. Both books give an excellent presentation of many of the issues regarding decision tree induction. C4.5 has a commercial successor, known as C5.0, which can be found at [www.rulequest.com](http://www.rulequest.com). ID3, a predecessor of C4.5, is detailed in [Qui86]. It expands on pioneering work on concept learning systems, described by Hunt, Marin, and Stone [HMS66]. Other algorithms for decision tree induction include FACT (Loh and Vanichsetakul [LV88]), QUEST (Loh and Shih [LS97]), PUBLIC (Rastogi and Shim [RS98]), and CHAID (Kass [Kas80] and Magidson [Mag94]). INFERULE (Uthurusamy, Fayyad, and Spangler [UFS91]) learns decision trees from inconclusive data, where probabilistic rather than categorical classification rules are obtained. KATE (Manago and Kodratoff [MK91]) learns decision trees from complex structured data. Incremental versions of ID3 include ID4 (Schlimmer and Fisher [SF86a]) and ID5 (Utgoff [Utg88]), the latter of which is extended in Utgoff, Berkman, and Clouse [UBC97]. An incremental version of CART is described in Crawford [Cra89]. BOAT (Gehrke, Ganti, Ramakrishnan, and Loh [GGRL99]), a decision tree algorithm that addresses the scalability issue in data mining, is also incremental. Other decision tree algorithms that address scalability include SLIQ (Mehta, Agrawal, and Rissanen [MAR96]), SPRINT (Shafer, Agrawal, and Mehta [SAM96]), RainForest (Gehrke, Ramakrishnan, and Ganti [GRG98]), and earlier approaches, such as [Cat91, CS93a, CS93b]. The integration of attribution-oriented induction with decision tree induction is proposed in Kamber, Winstone, Gong, et al. [KWG<sup>+</sup>97]. For a comprehensive survey of many salient issues relating to decision tree induction, such as attribute selection and pruning, see Murthy [Mur98].

For a detailed discussion on attribute selection measures, see Kononenko and Hong [KH97]. Information gain was proposed by Quinlan [Qui86] and is based on pioneering work on information theory by Shannon and Weaver [SW49]. The gain ratio, proposed as an extension to information gain, is described as part of C4.5 [Qui93]. The gini index was proposed for CART [BFOS84]. The G-statistic, based on information theory, is given in Sokal and Rohlf [SR81]. Comparisons of attribute selection measures include Buntine and Niblett [BN92], Fayyad and Irani [FI92], Kononenko [Kon95], Loh and Shih [LS97], and Shih [Shi00]. Fayyad and Irani [FI92] show limitations of impurity-based measures such as information gain and gini index. They propose a class of attribute selection measures called C-SEP (Class SEPARation), which outperform impurity-based measures in certain cases. Kononenko [Kon95] notes that attribute selection measures based on the minimum description length principle have the least bias towards multivalued attributes. Martin and Hirschberg [MH95] proved that the time complexity of decision tree induction increases exponentially with respect to tree height in the worst case, and under fairly general conditions, in the average case. Fayad and Irani [FI90] found that shallow decision trees tend to have many leaves and higher error rates for a large variety of domains. Attribute (or feature) construction is described in Liu and Motoda [LM98, Le98]. Examples of systems with attribute construction include BACON by Langley, Simon, Bradshaw, Zytkow [LSBZ87], Stagger by Schlimmer [Sch86], FRINGE by Pagallo [Pag89], and AQ17-DCI by Bloedorn and Michalski [BM98].

There are numerous algorithms for decision tree pruning, including cost complexity pruning (Breiman, Friedman, Olshen, and Stone [BFOS84]), reduced error pruning (Quinlan [Qui87]), and pessimistic pruning (Quinlan [Qui86]). PUBLIC (Rastogi and Shim [RS98]) integrates decision tree construction with tree pruning. MDL-based pruning methods can be found in Quinlan and Rivest [QR89], Mehta, Agrawal, and Rissanen [MRA95], and Rastogi and Shim [RS98]. Other methods include Niblett and Bratko [NB86], and Hosking, Pednault, and Sudan [HPS97]. For an empirical comparison of pruning methods, see Mingers [Min89] and Malerba, Floriana, and Semeraro [MFS95]. For a survey on simplifying decision trees, see Breslow and Aha [BA97].

There are several examples of rule-based classifiers. These include AQ15 (Hong, Mozetic, and Michalski [HMM86]), CN2 (Clark and Niblett [CN89]), ITRULE (Smyth and Goodman [SG92]), [NEW RISE (Domingos [Dom94]), IREP (Furnkranz and Widmer [FW94]), RIPPER (Cohen [Coh95]),] FOIL (Quinlan [NEW and Cameron-Jones] [Qui90, QCJ93]), and Swap-1 (Weiss and Indurkha [WI98]). For the extraction of rules from decision trees, see Quinlan [Qui87, Qui93]. Rule refinement strategies that identify the most interesting rules among a given rule set can be found in Major and Mangano [MM95].

Thorough presentations of Bayesian classification can be found in Duda, Hart, and Stork [DHS01], Weiss and Kulikowski [WK91], and Mitchell [Mit97]. For an analysis of the predictive power of naive Bayesian classifiers when the class conditional independence assumption is violated, see Domingos and Pazzani [DP96]. Experiments with kernel density estimation for continuous-valued attributes, rather than Gaussian estimation, have been reported for naive Bayesian classifiers in John [Joh97]. For an introduction to Bayesian belief networks, see Heckerman [Hec96]. For a thorough presentation of probabilistic networks, see Pearl [Pea88]. Solutions for learning the belief network structure from training data given observable variables are proposed in [CH92, Bun94, HGC95]. Algorithms for inference on belief networks can be found in Russell and Norvig [RN95] and Jensen [Jen96]. The method of gradient descent, described in Section 6.4.4 for training Bayesian belief networks, is given in Russell, Binder, Koller, and Kanazawa [RBKK95]. The example given in Figure 6.11 is adapted from Russell et al. [RBKK95]. Alternative strategies for learning belief networks with hidden variables include application of Dempster, Laird, and Rubin's [DLR77] EM (Expectation Maximization) algorithm (Lauritzen [Lau95]) and methods based on the minimum description length principle (Lam [Lam98]). Cooper [Coo90] showed that the general problem of inference in unconstrained belief networks is NP-hard. Limitations of belief networks, such as their large computational complexity (Laskey and Mahoney [LM97]), have prompted the exploration of hierarchical and composable Bayesian models (Pfeffer, Koller, Milch, and Takusagawa [PKMT99] and Xiang, Olesen, and Jensen [XOJ00]). These follow an object-oriented approach to knowledge representation.

The perceptron is a simple neural network, proposed in 1958 by Rosenblatt [Ros58], which became a landmark in early machine learning history. Its input units are randomly connected to a single layer of output linear threshold units. In 1969, Minsky and Papert [MP69] showed that perceptrons are incapable of learning concepts that are linearly inseparable. This limitation, as well as limitations on hardware at the time, dampened enthusiasm for research in computational neuronal modeling for nearly twenty years. Renewed interest was sparked following presentation of the backpropagation algorithm in 1986 by Rumelhart, Hinton, and Williams [RHW86], as this algorithm can learn concepts that are linearly inseparable. Since then, many variations for backpropagation have been proposed, involving, for example, alternative error functions (Hanson and Burr [HB88]), dynamic adjustment of the network topology (Mézard and Nadal [MN89], Fahlman and Lebiere [FL90], Le Cun, Denker, and Solla [LDS90], and Harp, Samad, and Guha [HSG90]), and dynamic adjustment of the learning rate and momentum parameters (Jacobs [Jac88]). Other variations are discussed in Chauvin and Rumelhart [CR95]. Books on neural networks include [RM86, HN90, HKP91, Fu94, CR95, Bis95, Rip96, Hay99]. Many books on machine learning, such as [Mit97, RN95], also contain good explanations of the backpropagation algorithm. There are several techniques for extracting rules from neural networks, such as [SN88, Gal93, TS93, Fu94, Avn95, LSL95, CS96b, LGT97]. The method of rule extraction described in Section 6.6.4 is based on Lu, Setiono, and Liu [LSL95]. Critiques of techniques for rule extraction from neural networks can be found in Craven and Shavlik [CS97]. Roy [Roy00] proposes that the theoretical foundations of neural networks are flawed with respect to assumptions made regarding how connectionist learning models the brain. An extensive survey of applications of neural networks in industry, business, and science is provided in Widrow, Rumelhart, and Lehr [WRL94].

Support Vector Machines (SVMs) grew out of early work by Vapnik and Chervonenkis on statistical learning theory [VC71]. The first paper on SVMs was presented by Boser, Guyon and Vapnik [BGV92]. More detailed accounts can be found in books by Vapnik [Vap95, Vap98]. Good starting points include the tutorial on SVMs by Burges [Bur98] and textbook coverage by Kecman [Kec01]. For methods for solving optimization problems, see Fletcher [Fle87] and Nocedal and Wright [NW99]. These references give additional details alluded to as “fancy math tricks” in our text, such as transformation of the problem to a Lagrangian formulation and subsequent solving using Karush-Kuhn-Tucker (KKT) conditions. For the application of SVMs to regression, see Schölkopf, Bartlett, Smola, and Williamson [SBSW99], and Drucker, Burges, Kaufman, Smola, and Vapnik [DBK<sup>+</sup>97]. Approaches to SVM for large data include the sequential minimal optimization algorithm by Platt [Pla98], decomposition approaches such as in Osuna, Freund, and Girosi [OFG97], and CB-SVM, a microclustering based SVM algorithm

for large data sets, by Yu, Yang and Han [YYH03].

Many algorithms have been proposed that adapt association rule mining to the task of classification. The CBA algorithm for associative classification was proposed by Liu, Hsu, and Ma [LHM98]. A classifier, using emerging patterns, was proposed by Dong and Li [DL99] and Li, Dong, and Ramamohanarao [LDR00]. CMAR (Classification based on Multiple Association Rules) was presented in Li, Han, and Pei [LHP01]. CPAR (Classification based on Predictive Association Rules) was proposed in Yin and Han [YH03b]. Cong, Tan, Tung, and Xu proposed a method for mining top- $k$  covering rule groups for classifying gene expression data with high accuracy [CTTX05]. Lent, Swami, and Widom [LSW97] proposed the ARCS system, which was described in Section 5.3 on mining multidimensional association rules. It combines ideas from association rule mining, clustering, and image processing, and applies them to classification. Meretakakis and Wüthrich [MW99] proposed to construct a naive Bayesian classifier by mining long itemsets.

Nearest-neighbor classifiers were introduced in 1951 by Fix and Hodges [FH51]. A comprehensive collection of articles on nearest neighbor classification can be found in Dasarathy [Das91]. Additional references can be found in many texts on classification, such as Duda, Hart, and Stork [DHS01] and James [Jam85], as well as articles by Cover and Hart [CH67] and Fukunaga and Hummels [FH87]. Their integration with attribute-weighting and the pruning of noisy instances is described in Aha [Aha92]. The use of search trees to improve nearest neighbor classification time is detailed in Friedman, Bentley, and Finkel [FBF77]. The partial distance method was proposed by researchers in vector quantization and compression. It is outlined in Gersho and Gray [GG92]. The editing method for removing “useless” training tuples was first proposed by Hart [Har68]. The computational complexity of nearest neighbor classifiers is described in Preparata and Shamos [PS85]. References on case-based reasoning (CBR) include the texts [RS89, Kol93, Lea96], as well as [AP94]. For a list of business applications, see [All94]. Examples in medicine include CASEY by Koton [Kot88] and PROTOS by Bareiss, Porter, and Weir [BPW88], while Rissland and Ashley [RA87] is an example of CBR for law. CBR is available in several commercial software products. For texts on genetic algorithms, see [Gol89, Mic92, Mit96]. Rough sets were introduced in Pawlak [Paw91]. Concise summaries of rough set theory in data mining include Ziarko [Zia91], and Cios, Pedrycz, and Swiniarski [CPS98]. Rough sets have been used for feature reduction and expert system design in many applications, including [Zia91, LP97, Swi98]. Algorithms to reduce the computation intensity in finding reducts have been proposed in [SR92]. Fuzzy set theory was proposed by Lofti Zadeh in [Zad65, Zad83]. Additional descriptions can be found in [YZ94, Kec01].

There are many good textbooks that cover the techniques of regression. Examples include [Jam85, Dob90, JW92, Dev95, HC95, NKNW96, Agr96]. The book by Press, Teukolsky, Vetterling, and Flannery [PTVF96] and accompanying source code contain many statistical procedures, such as the method of least squares for both linear and multiple regression. Recent nonlinear regression models include projection pursuit and MARS (Friedman [Fri91]). Log-linear models are also known in the computer science literature as *multiplicative models*. For log-linear models from a computer science perspective, see Pearl [Pea88]. Regression trees (Breiman, Friedman, Olshen, and Stone [BFOS84]) are often comparable in performance with other regression methods, particularly when there exist many higher-order dependencies among the predictor variables. For model trees, see Quinlan [Qui92].

Methods for data cleaning and data transformation are discussed in Kennedy, Lee, Van Roy, et al. [KLV<sup>+</sup>98], Weiss and Indurkha [WI98], Pyle [Pyl99], and Chapter 2 of this book. Issues involved in estimating classifier accuracy are described in Weiss and Kulikowski [WK91] [NEW and Witten and Frank] [WF00a]. The use of stratified 10-fold cross-validation for estimating classifier accuracy is recommended over the holdout, cross-validation, leave-one-out (Stone [Sto74]) and bootstrapping (Efron and Tibshirani [ET93]) methods, based on a theoretical and empirical study by Kohavi [Koh95]. Bagging is proposed in Breiman [Bre96]. The boosting technique of Freund and Schapire [FS97] has been applied to several different classifiers, including decision tree induction (Quinlan [Qui96]) and naive Bayesian classification (Elkan [Elk97]). Sensitivity, specificity, and precision are discussed in Frakes and Baeza-Yates [FBY92]. For ROC analysis, see Egan [Ega75] and Swets [Swe88].

The University of California at Irvine (UCI) maintains a Machine Learning Repository of data sets, for the development and testing of classification algorithms, and a Knowledge Discovery in Databases (KDD) Archive, an online repository of large data sets that encompasses a wide variety of data types, analysis tasks, and application areas. For information on these two repositories, see <http://www.ics.uci.edu/~mllearn/MLRepository.html> and <http://kdd.ics.uci.edu>.

No classification method is superior over all others for all data types and domains. Empirical comparisons of classification methods include [Qui88, SMT91, BCP93, CM94, MST94, BU95], and [LLS00].