Mining interesting itemsets

Ardian Kristano Poernomo

2011

Ardian, K. P. (2011). Mining interesting itemsets. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/45201>

<https://doi.org/10.32657/10356/45201>

MINING INTERESTING ITEMSETS

Ardian Kristanto Poernomo

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfilment of the requirement for the degree of
Doctor of Philosophy

2011

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Dr. Vivekanand Gopalkrishnan) Thesis Advisor

Approved for the University Committee on Graduate Studies.

Acknowledgments

Pleasant and grueling. That's how I describe my four years journey in fulfilling this study. Many ups and downs, frustrations and delights, it was really a long and windy road covered by flower blooms. Having said that, I would like to thank all who have helped me to end this journey, to pass through the winds, to let me see the flower blooms. All thanks and praises to God for his blessings through this journey.

Firstly, I would like to thank my advisor, Dr. Vivekanand Gopalkrishnan, for getting me started, providing me with directions and guidance, and consistent encouragements throughout this study. Even more, he has become my life mentor, and also my friend.

I would also thank the examiners and panels, Dr. Zhang Jun, Dr. Gao Cong, Dr. Tan Ah-Hwee, and two anonymous external examiners, for dedicating their time and efforts to review my thesis. Their suggestions are very useful to improve the quality of this thesis. Special thanks also to Dr. Sourav Saha Bhowmick, for his guidance and suggestions.

I would also like to thank my colleagues: Alex, Kelvin, Hoang Vu, Meiqun, Meishan, Linhong, Wumin, Quang Huy, my friends: Bram, Prima, Maggy, Ilham, the lab technicians, Mr. Lai Chee Keong and Mr. Chua Chiew Song, and others whom I unintentionally missed.

Finally, I am grateful to my wife, my mother, my sister, and in laws for their continuous encouragement and support.

Abstract

Data mining aims to discover knowledge in large databases. The desired knowledge, normally represented as patterns, are deemed interesting if they benefit some applications. Therefore, the objective of data mining can be translated to finding interesting patterns from observational data. In this thesis, we focus on the simplest form of patterns, which is a set of features (items), also called an itemset.

In a high level, data mining process can be split into three parts. The first is to define the notion of interesting patterns. The solution of this sub-task is highly application and domain dependent. Having the mathematical formulation of interesting patterns, the next subtask is to find/enumerate those interesting patterns. As the number of interesting patterns are usually too much, the last subtask is to present those patterns in an interpretable and concise form. This thesis focuses on the second and the third sub-tasks, which we call as the *enumeration problem* and the *summarization problem* respectively.

Analyzing the current status, we observe some ironic facts regarding these problems. For the enumeration problem, we observe tremendous efforts for developing efficient algorithms for enumerating *frequent* itemsets, and also for *defining* alternative interestingness measures other than frequency. However, there is currently no efficient algorithm for enumerating general interesting itemsets, especially when the interestingness measure does not satisfy the anti-monotone property. For the summarization problem, whilst the original motivating goal was to improve the interpretability of interesting itemsets, most of the current works are concerned solely about the *conciseness* of the summary, instead of its actual interpretability. Our goal is to bridge the efficiency gap between the algorithms for enumerating frequent itemsets and those for interesting itemsets, and to improve quality of the summary, with respect to its interpretability, conciseness, and accuracy.

To that end, this thesis achieves the following technical contributions. We propose general algorithms for calculating the interestingness of an itemset, with only a little

more cost than calculating its frequency. We also propose algorithms for enumerating itemsets when the notion of interestingness does not satisfy the anti-monotone property. In short, our proposed algorithms have advanced the state-of-the-art for many variants of the interesting itemset enumeration problem, and hence narrowed the gap between enumerating interesting itemsets and frequent itemsets. For the summarization problem, we tackle several popular problem variants, and manage to design algorithms that produce interpretable summaries, without sacrificing accuracy or conciseness.

Many previous researches rely on approximate algorithms for dealing with large databases. With our innovations, those problems would be solved in more efficient and effective ways. Analysts would also be able to analyze the data better, and hence take better decisions, with our interpretable and accurate summaries.

Contents

Acknowledgments	iii
Abstract	iv
1 Introduction	1
1.1 Background	1
1.1.1 Data Mining / Pattern Mining	1
1.1.2 Interesting Itemset Mining	2
1.2 The Enumeration Problem	4
1.2.1 Problems and Challenges	5
1.2.2 Contributions	7
1.3 The Summarization Problem	8
1.3.1 Analysis Methodology	8
1.3.2 Problems and Challenges	10
1.3.3 Contributions	11
1.4 Organization	12
2 Enumerating Interesting Itemsets	13
2.1 Preliminaries	13
2.1.1 Interestingness Measure	14
2.1.2 Problem Variants	14
2.1.3 Itemset Enumeration Framework	15
2.1.4 Basic Notations	15
2.1.5 Organization of the Chapter	16
2.2 Candidate Generation	16
2.2.1 Monotonicity Properties	17
2.2.2 Algorithms for Measures with the Anti-Monotone Property . .	20
2.2.3 Algorithms for Measures with the Weak Anti-Monotone Property	27

2.2.4	Algorithms for Measures with the (Supweak) Monotone Property	30
2.2.5	Algorithms for Measures Without any Monotonicity Properties	31
2.3	Interestingness Computation	32
2.3.1	Detailed Support	33
2.3.2	Algorithms for Computing Exact Support	34
2.3.3	Algorithms for Computing Subset Support	36
2.3.4	Algorithms for Computing Partial Support	40
2.4	Summary of Algorithms	44
2.5	Applications	45
2.5.1	Entropy	46
2.5.2	Density	48
2.5.3	Bond	50
2.5.4	Derivability	50
2.5.5	Expected Support in Bernoulli Noise	51
2.5.6	Summary of Interestingness Measures	53
2.6	Experimental Results	53
2.6.1	Computing Partial Supports and Subset Supports	54
2.6.2	Applications Requiring Partial Supports	59
2.6.3	Applications Requiring Subset Supports	62
2.7	Concluding Remarks	65
3	Enumerating Correlated Itemsets	67
3.1	Preliminaries	67
3.1.1	Motivation	67
3.1.2	Related Work	69
3.1.3	Basic Definitions and Notations	69
3.1.4	Differences with the General Framework	70
3.2	Algorithms	71
3.2.1	Monotonicity Property of Correlation	71
3.2.2	Enumerating Correlated Itemsets	72
3.2.3	Integrating Closed Constraint	73
3.2.4	Integrating Support Constraint	74
3.3	Experimental Results	76
3.3.1	Without Support Constraint	76
3.3.2	With Support Constraint	77
3.3.3	Case Study on Additional Constraints	79

3.4	Concluding Remarks	79
4	Enumerating Fault-Tolerant Frequent Itemsets	81
4.1	Preliminaries	81
4.1.1	Motivation	81
4.1.2	Basic Definitions and Notations	82
4.1.3	Problem Variants	84
4.1.4	Differences with the General Framework	84
4.2	Constant Transaction Relaxation ($\varepsilon_t^c \neq \infty$)	85
4.2.1	Integrating Minimum Item Support Threshold	85
4.3	Constant Relaxation ($\varepsilon_t^c, \varepsilon_i^c, \varepsilon_p^c \neq \infty$)	86
4.3.1	Candidate Generation	87
4.3.2	Interestingness Computation	87
4.4	Proportional Relaxation ($\varepsilon_t, \varepsilon_i, \varepsilon_p < 1$)	92
4.4.1	Candidate Generation	93
4.4.2	Calculating the Fault-Tolerant Support	98
4.5	Experimental Results	100
4.5.1	Constant Transaction Relaxation	101
4.5.2	Constant Relaxation	101
4.5.3	Proportional Relaxation	102
4.6	Concluding Remarks	106
5	Duality between Mining Frequent Itemsets and Fault-Tolerant Fre-	107
	quent Itemsets	
5.1	Preliminaries	108
5.1.1	Motivation	108
5.1.2	Contributions	108
5.1.3	Basic Definitions and Notations	109
5.1.4	Organization	110
5.2	Intuitive (Infeasible) Approach	110
5.2.1	Properties of the Transformed Database	111
5.2.2	Issues of the Item Transformation	112
5.3	Proposed Approach	112
5.3.1	Inverse Frequent Itemset Mining	113
5.3.2	Porting Naïve Algorithm for Duality Problem	115
5.3.3	Deriving the Weight of Itemsets	117
5.3.4	Optimality of the Size of the Transformed Database	119

5.3.5	Properties of the Transformed Database	119
5.3.6	Transformation Algorithm	120
5.4	Concluding Remarks	120
6	Summarizing Frequent Itemsets	123
6.1	Preliminaries	124
6.1.1	Related Works	124
6.2	Summary Form: CP-Summary	127
6.3	Notations and Problem Definition	127
6.4	Algorithm	129
6.4.1	Finding the Best Splitting Items	130
6.4.2	Verifying and Forming Valid Profiles	134
6.4.3	Time Complexity Analysis	135
6.4.4	Quality Analysis	136
6.5	Experimental Results	136
6.5.1	Result Quality	136
6.5.2	Comparison with Other Techniques	139
6.6	Concluding Remarks	140
7	Summarizing Subspace Clusters	143
7.1	Preliminaries	143
7.1.1	Related Works	144
7.2	Summary Form	146
7.3	Notations and Problem Definition	147
7.4	Approach	149
7.4.1	Analysis of Compromising Recall and Precision	149
7.4.2	Algorithms	151
7.5	Experimental Results	154
7.5.1	Effectiveness	155
7.5.2	Efficiency	156
7.6	Concluding Remarks	157
8	Summarizing Itemsets for Class Description	159
8.1	Preliminaries	159
8.2	Summary Form	160
8.3	Notations and Problem Definition	161
8.4	Algorithm	163

8.4.1	Main Method	164
8.4.2	Finding the Best Sentence	164
8.4.3	Finding Optimal Values	165
8.4.4	Complexity Analysis	170
8.5	Experimental Results	170
8.5.1	Impact of Partial Coverage	172
8.5.2	Comparison with C4.5	172
8.5.3	Sensitivity to Parameters	174
8.5.4	Case Study	175
8.6	Related Works	176
8.7	Concluding Remarks	177
9	Conclusion	179
9.1	Summary	179
9.1.1	The Enumeration Problem	179
9.1.2	The Summarization Problem	180
9.2	Future Work	181
A	Experimental Setting	185
B	Proof of Equation (5.4)	186
	Bibliography	189

List of Tables

2.1	Variants of the interesting itemset enumeration problem.	15
2.2	Monotonicity properties of interestingness measures.	17
2.3	Sample interesting itemsets for measures satisfying the anti-monotone property	20
2.4	Sample interesting itemsets for measures satisfying the weak anti-monotone property	27
2.5	Step-by-step illustration of the QIE algorithm.	39
2.6	Information required by algorithms for calculating detailed supports and their time and space complexities.	45
2.7	Information provided by candidate generation algorithms.	45
2.8	Interestingness measures along with their monotonicity properties and required detailed supports.	53
3.1	Comparison between correlation and support of sample itemsets in Zoo data.	68
3.2	Top-10 correlated itemsets in Zoo data, projected to four attributes. .	79
4.1	Impact of relaxation criteria on the anti-monotone property of FT-support.	93
4.2	Impact of relaxation criteria on the anti-monotone property of FT-support, and on the complexity of interestingness computation. . . .	100
4.3	Detailed snapshot of the efficiency of the optimal and approximate bound for FT-support.	103
6.1	Detailed snapshot of the results of K-Regression and Hierarchical summarization algorithms.	140
7.1	Number of patterns produced by the Concurrent and Consecutive algorithms, varying σ	155

7.2	Number of patterns produced by the Concurrent and Consecutive algorithms, varying ε	155
8.1	Impact of allowing partial coverage for concept description.	171
A.1	Transactional databases from FIMI.	185
A.2	Categorical datasets from UCI repository. All numerical attributes in the Pendigits dataset are discretized into two bins with equal width. .	185

List of Figures

1.1	Sample transactional database with 4 transactions (transactions 1, 2, 3, and 4), and 4 items (item a , b , c , and d).	2
1.2	The enumeration and the summarization problems. The symbol μ represents the interestingness of an itemset, and the symbol ν represents the goodness of the summary with respect to the interesting itemsets.	4
1.3	The framework for enumerating interesting itemsets.	5
2.1	Data structures for enumerating frequent itemsets.	35
2.2	Sample transactional database as a running example (repeated).	46
2.3	Enumerating frequent itemsets with partial supports.	54
2.4	Enumerating frequent itemsets with subset supports.	55
2.5	Efficiency of calculating exact support, partial support, and subset support.	55
2.6	Enumerating maximal frequent itemsets.	56
2.7	Enumerating top- K frequent itemsets.	57
2.8	Enumerating frequent itemsets assuming weak anti-monotone property.	57
2.9	Enumerating maximal frequent itemsets assuming weak anti-monotone property.	58
2.10	Enumerating top- K frequent itemsets assuming weak anti-monotone property.	59
2.11	Enumerating dense itemsets. Figures (a) to (c) plot the scalability of each algorithm with respect to s , when σ is fixed. Figures (d) to (e) plot the scalability when σ varies.	60
2.12	Enumerating frequent itemsets in presence of Bernoulli noise. Figures (a) to (c) plot the scalability of each algorithm with respect to σ , when p and q are fixed. Figures (d) to (e) plot the scalability when p and q vary.	61
2.13	Enumerating large bond itemsets.	62

2.14	Enumerating low entropy itemsets.	63
2.15	Enumerating non-derivable itemsets. Notice the absolute minimum support and linear scale in both Mushroom and Chess datasets. . . .	63
2.16	Enumerating frequent itemsets in the presence of Bernoulli noise. Figures (a) to (c) plot the scalability of each algorithm with respect to σ , when p and q are fixed. Figures (d) to (e) plot the scalability when p and q vary. The specified values of p and q are the <i>mean</i> of the noise for each item, drawn by normal distribution.	64
3.1	Enumerating top- K correlated itemsets.	77
3.2	Enumerating frequent closed top- K correlated itemsets. The value of K is set to 10 in Figures (a) to (f).	78
4.1	Application of FT-itemsets for subspace clustering.	82
4.2	Approximating and tightening the bounds for FT-support.	95
4.3	Enumerating fault-tolerant frequent itemsets with constant transaction relaxation. The value of σ^i is fixed to 0.5σ in BMS-Webview-1 and Mushroom datasets, and 0.8σ in Chess dataset.	101
4.4	Impact of pre-processing on the interestingness computation module.	102
4.5	Efficiency of ILP with proportional relaxation and Iterative + constant relaxation.	103
4.6	Efficiency of the optimal and approximate bound of FT-support. . . .	104
4.7	Impact of the maximum desired FTFI.	105
4.8	Sensitivity with respect to minimum support and error tolerances. . .	105
5.1	Sample transactional database as a running example (repeated). . . .	110
5.2	Item transformation with $\varepsilon = 1$	111
6.1	Tree-structure of a <i>CP-Summary</i> , with five <i>C-Profiles</i> , with bases $\emptyset$, a , b , c , and ab	130
6.2	Splitting problem for constructing the prefix-tree.	132
6.3	Evaluation of CP-Summary in real datasets.	137
6.4	Evaluation of CP-Summary on synthetic datasets.	139
6.5	Quality of Hierarchical and K-Regression summarization algorithms. Note that CP-Summary always produces 100% precision, recall, and F_1 .139	

7.1	One-cluster summaries of (a) the input clusters, provided by (b) and (c) the consecutive approaches, and (d) the proposed approaches, as shown by green-thatched regions.	146
7.2	Spuriousness and coverage for representing pattern $p = (Y_p \times X_p)$ by pattern $r = (Y_r \times X_r)$. The green-thatched region represents uncovered patterns, and the red-dotted region represents spurious patterns. . . .	147
7.3	Efficiency of concurrent and consecutive algorithms, varying σ and ε	157
8.1	Plots of x (true positives) vs. y (false positives) for various values of F_β	165
8.2	Finding maximum y -intercept through a set of points, given a gradient.	168
8.3	F_1 measure of PUB and C4.5 algorithms.	173
8.4	Sensitivity of PUB with respect to b and w	174

Notations

Itemset	A set of items
$\mu(X)$	Interestingness of itemset X
$\nu(R)$	Quality of summary R
σ	Minimum interestingness threshold
K	Number of resultant itemsets
Candidate	Itemsets whose interestingness is calculated
D	The database
T	Set of all transactions
I	Set of all items
n	Number of transactions or objects
m	Number of items
$I(y)$	Set of items contained by transaction y
$T(x)$	Set of transactions containing item x
$\mathcal{S}(X)$	The supporting transactions of itemset X
k -itemset	Itemsets of size k
k -subset	Subsets of size k
$sup(X), rsup(X)$	The support and relative support of itemset X
FI, FIM	Frequent itemset, FI Mining
MII	Maximal interesting itemsets
SS	Subset support

$sup_Y(X)$	The number of transactions containing all items in Y but none in $X \setminus Y$
PS	Partial support
$sup_{\geq k}(X)$, $sup_{=k}(X)$, $sup_{\leq k}(X)$	The number of transactions containing at least, exactly, and at most k items in X
O , $n = O $	Set of all objects
A , $t = A $	Set of all attributes
$A_i \in A$, D_i	An attribute, domain of that attribute
$A_i(o) \in D_i$	The value of object o in attribute A_i
$\hat{sup}(X)$, $r\hat{sup}(X)$	The estimated support and relative support of itemset X
CFI	Closed frequent itemsets
$ub(X)$	The upper bound of $\mu(X)$
FTFI, FTFIM	Fault-tolerant frequent itemsets, FTFI Mining
FT-support, $sup_{FT}(X)$	Fault-tolerant support of itemset X
$\mathcal{S}_{FT}(X)$	Maximum sized transaction-set that FT-supports X
ε	Error tolerance
$sup^Y(X)$	Number of transactions containing X in transaction-set Y
$\ X\ _k$	All k -subsets of X
$\mathcal{FI}(D)$, $\mathcal{FTFI}(D)$	Set of FIs and FTFIs in database D

List of Publications

Most of the content of this thesis has been presented in the following publications:

1. Ardian Kristanto Poernomo, Vivekanand Gopalkrishnan: PUB: A Class Description Technique Based on Partial Coverage of Subspace. ICDM 2009: 920–925
2. Ardian Kristanto Poernomo, Vivekanand Gopalkrishnan: CP-summary: a Concise Representation for Browsing Frequent Itemsets. KDD 2009: 687–696
3. Ardian Kristanto Poernomo, Vivekanand Gopalkrishnan: Towards Efficient Mining of Proportional Fault-Tolerant Frequent itemsets. KDD 2009: 697–706
4. Ardian Kristanto Poernomo, Vivekanand Gopalkrishnan: Efficient Computation of Partial-Support for mining Interesting Itemsets. SDM 2009: 1014–1025
5. Ardian Kristanto Poernomo, Vivekanand Gopalkrishnan: Mining Statistical Information of Frequent Fault-Tolerant Patterns in Transactional Databases. ICDM 2007: 272–281
6. Ardian Kristanto Poernomo, Vivekanand Gopalkrishnan: Efficiently Mining Top- K Correlated Itemsets in Categorical Databases. Technical Report CAIS-2010/02, Nanyang Technological University, Singapore, February 2010.
7. Ardian Kristanto Poernomo, Vivekanand Gopalkrishnan: Summarizing Subspace Clusters by Compromising Coverage and non-Spuriousness. Technical Report CAIS-2010/03, Nanyang Technological University, Singapore, March 2010.
8. Ardian Kristanto Poernomo, Vivekanand Gopalkrishnan: Duality between the Fault-Tolerant Frequent Itemset and Frequent Itemset Mining Problems. Technical Report CAIS-2008/09, Nanyang Technological University, Singapore, September 2008.

9. Ardian Kristanto Poernomo, Calin Garboni, Toon Calders, Vivekanand Gopalkrishnan, Bart Goethals, Céline Robardet: Representation of the Support for Itemsets. *under preparation for publication*.

Publications 1–5 have been published in peer-reviewed conferences, Publications 6–8 are published as technical report, and Publication 9 is currently under preparation for a journal submission.

Publications 3, 4, 5, 6, 8, and 9 deal with the enumeration problems. Publication 5 proposes a framework to enumerate interesting itemsets, with an algorithm to enumerate fault-tolerant frequent itemsets with constant relaxation. Publication 4 uses this framework to enumerate interesting itemsets, and proposes an efficient algorithm for calculating the partial supports of an itemset. Publication 9 generalizes Publication 4 to also include the algorithm for calculating the subset-supports of an itemset. These works are extended in Chapter 2 to handle various problem settings (enumerating *maximal* and *top-K* interesting itemsets) and non-anti-monotone interestingness measures.

Publication 3 uses the framework and algorithms in Publications 4, 5, and 9 to enumerate fault-tolerant frequent itemsets. In addition, Publication 3 proposes an upper-bound function to effectively bound the search space, and an efficient algorithm for calculating the fault-tolerant support. Similarly, Publication 6 proposes an efficient algorithm to enumerate correlated itemsets. These two publications form the body of Chapters 4 and 3 respectively.

Publication 8 proposes a solution for the duality problem to reduce the problem of mining fault-tolerant frequent itemsets to mining frequent itemsets. This result allows fault-tolerant frequent itemsets to be enumerated by using frequent itemset mining implementations, and is presented in Chapter 5.

Publications 1, 2, and 7 deal with the summarization problem, where we propose algorithms to produce interpretable summaries, without sacrificing their conciseness, accuracy, and coverage.

Publication 2 proposes an algorithm to summarize frequent itemsets using *conditional profile*. The produced summary has no false negative or positive, and provides error guarantee of the estimation error. Publication 7 proposes an algorithm to summarize subspace clusters, that compromises both recall and precision. Finally, Publication 1 proposes an algorithm to describe a concept in the database, by expanding the coverage of each itemset (sentence) to cover objects (transactions) partially containing the itemset. These three publications are the bases of Chapters 6, 7, and 8 respectively.

Chapter 1

Introduction

1.1 Background

1.1.1 Data Mining / Pattern Mining

In this era where storage is cheap, we are heavily fed with massive amounts of data. On the one hand, it is beneficial to have massive data, since it provides a lot of information. On the other hand, such massive data is infeasible to manually analyze. This limitation leads to the birth of a problem to discover *knowledge* in a database, which is called *data mining*. This mined knowledge can generally be presented as *patterns*. A pattern is deemed *interesting* if it benefits an application. Hence, data mining can be translated into a problem to find interesting patterns from observational data.

There are many applications in the real world where data mining finds itself useful. In bio-medical research, scientists might be interested in finding the similarity between human and mouse genes, a supermarket owner might be interested in ways to boost the sales of particular products, and an insurance agent might be interested in determining whether a potential customer should be rejected or accepted. We can solve these tasks by mining the *interesting patterns*, which are formed as: a set of genes which behave similarly in both species, associated products, and properties to distinguish prospective customers, respectively. Each of these three tasks is an example of famous data mining problems, namely clustering, association rule mining, and classification, respectively.

Tid	Itemset
1	a, b
2	c, d
3	b
4	a, c, d

(a) Table form

	a	b	c	d
1				
2				
3				
4				

(b) Matrix form

Figure 1.1: Sample transactional database with 4 transactions (transactions 1, 2, 3, and 4), and 4 items (item a , b , c , and d).

1.1.2 Interesting Itemset Mining

In order to apply a pattern mining technique, we first need to formulate the *interesting pattern*. Desirable patterns may be of various types, viz., sets, sequences, strings, trends, trees, graphs, and other complex structures. The measure of interestingness may also vary depending on the applications.

Transactional Database – Itemsets

In this thesis, we narrow down our scope to the simplest form of pattern, which is a set or also called an itemset (a set of items). In the pioneering work on itemset mining [4], the notion of *item* relates to a product in a market basket database, and an itemset is simply a collection of products in the supermarket. In this application, we normally store the database as a set of transactions, where each transaction contains a set of items/products bought by a customer. Such a database is denoted as a *transactional database*, as illustrated in Figure 1.1. This database consists of 4 transactions and 4 items. Transaction 1 contains items a and b , transaction 2 contains items c and d , and so on.

The utility of interesting itemsets is ubiquitous. For example, within the domain of supermarket data analysis, interesting itemsets can be considered as a group of items which profit if they are sold in a discounted package, or put together in nearby shelves. Conceptually, interesting itemsets can be used to express a wide variety of requirements to group a set of items.

Due to its simplicity and expressiveness, the concept of transactions and items has been used in broader contexts. For example, we can use this concept to analyze web-log data and text data, where the items are the web-pages accessed in each session,

and words appearing in each sentence or paragraph respectively. More generally, we can treat any interesting *concept/event* as an item. For example, if we are interested with the “*gender*” of a patient in a medical database, we can treat “*gender = male*” and “*gender = female*” as items. Any numerical features can also be mapped into an item via discretization, e.g., “*age < 21*” [97]. It is not hard to imagine the usefulness of a set of these features. One famous example is to derive a rule for classification [50]. This notion of interesting itemsets have been widely applied in many real applications, such as bioinformatics [13], document clustering [28], and network intrusion detection [47].

Interestingness of Itemsets

Before we can mine the interesting itemsets, we need to formulate their interestingness measure. The most common one is the *frequency* or the *support* of an itemset, which is calculated as the number of transactions containing the itemset. Frequent itemsets are interesting or important as they appear significantly often in the database.

An itemset is a set of items. Statistically, a set is interesting if all elements in the set are correlated, i.e., they do not appear by chance. Following this argument, we can relate the interestingness of an itemset with the *correlation* between items [17, 59]. Another common measure of interestingness is based on *surprisingness* [51, 88], i.e., how much the appearance of this itemset deviates from the expectation, given some underlining assumptions. In some applications, we can measure the *utility* of an itemset, such as expected profit or accuracy. In those cases, an itemset is interesting if it has a high utility, which is commonly referred as *actionable* [96]. For a comprehensive survey on the definitions of the interestingness of itemsets, please refer to [29].

Having the interestingness measure, the distinction between interesting and non-interesting itemsets is usually determined by a simple cut-off. That is, an itemset is interesting if its interestingness value is greater than a threshold, and not if otherwise. These distinctions between interesting and uninteresting itemsets are referred as *interestingness criteria*.

Interesting Itemset Mining Problems

The basic mining problem, which we term as *the enumeration problem*, aims to find all interesting itemsets given an interestingness criterion. The application of this problem is straight-forward, i.e., as the notion of interestingness should reflect the applicability

Figure 1.2: The enumeration and the summarization problems. The symbol μ represents the interestingness of an itemset, and the symbol ν represents the goodness of the summary with respect to the interesting itemsets.

of the itemsets, having those interesting itemsets are useful. For example, all itemsets with positive *expected profit* are useful.

However, in some cases, the number of interesting itemsets are too large. These results, though useful, are impossible to manage, especially if they are intended to be manually analyzed. To resolve this issue, analysts need to specify a stricter interestingness criteria, for example by increasing the interestingness threshold. However, doing so decreases the comprehensiveness of the mined knowledge, i.e., many should-be-interesting-itemsets begin to be considered uninteresting. This limitation motivates *the summarization problem*, to represent those abundant knowledge into an interpretable form.

These two problems are illustrated in Figure 1.2. The enumeration problem takes the database and the interestingness criterion μ as its inputs, and returns all interesting itemsets. The summarization problem takes the database, the interestingness criterion μ , and the summarization measure ν as its inputs, and produces the *best* summary as the output. Parameter μ defines the interesting itemsets, and parameter ν defines the goodness of the summary, to represent the set of all interesting itemsets.

1.2 The Enumeration Problem

The objective of this problem is to mine/enumerate all interesting itemsets, based on an interestingness measure μ . An itemset X is considered more interesting than another itemset Y , if $\mu(X) > \mu(Y)$. Normally, an itemset X is considered interesting if $\mu(X)$ is not less than a given threshold σ .

The interesting itemset enumerating algorithm can be described in a high level

Figure 1.3: The framework for enumerating interesting itemsets.

framework consisting of two modules, viz., the candidate generation and the interestingness computation modules. The candidate generation module prompts an itemset X to the interestingness computation module, to check whether the respective itemset is interesting. The results of the computation are then fed-back to the candidate generation module, to decide the next itemsets to be prompted. This framework is illustrated in Figure 1.3, and are discussed in a much greater detail in Chapter 2.

1.2.1 Problems and Challenges

The pioneering instance of the interesting itemset mining problem is to mine *frequent itemsets*, i.e., where $\mu(X)$ is defined as the number of occurrences of X in the database [4]. This problem has now been extremely well-studied with great success, indicated by the existence of extremely efficient algorithms for enumerating frequent itemsets in varying problem settings. For instance, one of the current fastest algorithms (implementations) can enumerate millions of frequent itemsets from a database with millions of entries within seconds [11]. A thorough survey on the developments of the frequent itemset mining algorithms can be found in [34]. These huge research and development efforts for optimizing the enumeration of frequent itemsets indicate the importance of finding efficient solutions to this problem.

There are also tremendous research efforts to find the *best* interestingness measure, rather than frequency. In the past decades, alternate measures based on correlation [17, 59], surprisingness/derivability [22, 94, 88], predictability/uncertainty [39], and actionability [96], have been proposed. There is even a survey on the interestingness measures of itemsets [29]. These efforts indicate the importance of extending the notion of *frequent* itemsets to determine interesting itemsets.

Unfortunately, extending *frequent* itemset mining algorithms for mining *interesting* itemsets defined by general interestingness measures is difficult. Looking deeper into the individual enumeration module, generating candidate frequent itemsets is

easy because frequency satisfies the *anti-monotone* property [5], i.e., the frequency of an itemset is never greater than that of its subsets. With this property, we can enumerate all frequent itemsets by generating the itemsets from those with smallest number of items (start with the empty-itemset that contains zero item), and keep extending itemsets which are frequent. When the interestingness measure does not satisfy the anti-monotone property, we cannot use the same algorithm as there can be interesting itemsets with no interesting subsets. In such cases, literature often suggests to use heuristic and incomplete algorithms, to integrate a minimum frequency threshold, or even to enforce the anti-monotone property. This leads us to ask,

ENUMERATION PROBLEM 1: *How to efficiently generate candidates for enumerating interesting itemsets, when the interestingness measure does not satisfy the anti-monotone property?*

Similarly, calculating the frequency of an itemset can be done efficiently due to its simplistic nature, which simply counts the number of transactions containing all items in the set. This simplicity allows many sophisticated techniques to be applied [16]. However, these techniques are not applicable for other interestingness measures. Many recent publications [85, 40, 89] suggested to calculate the interestingness of an itemset naïvely, by iterating every transaction and examining the *status* of that itemset in the respective transaction. From our experiments in Chapter 2, we find that this naïve algorithm is over thousand times slower than the state-of-the-art algorithm for calculating the frequency of an itemset. This limitation leads us to the following research problem,

ENUMERATION PROBLEM 2: *How to calculate the interestingness of an itemset efficiently?*

Considering that the frequent itemset mining problem has been very well studied, a promising alternative to boost the efficiency of enumerating interesting itemsets is by reducing it to the frequent itemset mining problem. That is, we modify the database in such a way, that enumerating *frequent* itemsets in the *transformed* database produces equivalent results with enumerating *interesting* itemsets in the *original* database. With this transformation, we can enumerate interesting itemsets using the existing frequent itemset mining algorithms, and even the implementations. This requirement leads us to another research problem,

ENUMERATION PROBLEM 3: *Can the general interesting itemset mining problem be*

1.2. THE ENUMERATION PROBLEM

7

reduced to the frequent itemset mining problem?

1.2.2 Contributions

Our objective is to advance the methodology for enumerating interesting itemsets, by answering the mentioned research problems. With respect to this objective, we make the following contributions:

- We present a thorough and systematic study for various itemset enumeration problems with varying properties (Section 2.2). In particular, we consider the problems to enumerate all interesting itemsets, maximal interesting itemsets, and top- K interesting itemsets, and relate them with the monotonicity properties of the interestingness measures. This contribution addresses the difficulty of various itemset mining problems, including the case when the interestingness measure does not satisfy the anti-monotone property (Enumeration Problem 1).

With this study and categorization, we can see which optimization techniques are applicable for each problem variant, and pick the best techniques to form the most efficient algorithm. In some variants, our study allows us to propose a simple modification of the algorithms that can boost the efficiency of the state-of-the-art algorithm.

- We show that most interestingness measures proposed so far, can be formulated as functions of *detailed supports*, which is the number of transactions containing *certain parts* of the itemset. Having this, the problem of calculating the interestingness measure can be reduced to the problem of calculating the detailed supports (Section 2.3). This contribution addresses the challenge to efficiently calculate the interestingness of an itemset.

Following up this observation, we propose efficient algorithms for calculating the detailed supports of itemset. Our proposed algorithms are only slightly less efficient than that for calculating the exact support, and hence, can significantly improve the efficiency for enumerating interesting itemsets. Our experimental evaluations show that the performance improvement may reach two or three orders of magnitudes [74] (Enumeration Problem 2).

- We demonstrate that enumerating interesting itemsets without the anti-monotone property is not necessarily intractable, by proposing efficient (tractable) and complete algorithms for two non-anti-monotone measures, namely proportional

fault-tolerant frequent itemsets [76] (Chapter 4) and correlated itemsets [77] (Chapter 3). These two measures were chosen because of their usability and difficulty. In particular, mining interesting itemsets with these measures were previously considered intractable [85, 12, 48] (Instances of Enumeration Problems 1 and 2).

- We propose a solution for the duality problem for enumerating fault-tolerant frequent itemsets, with constant transaction relaxation [72] (Chapter 5). With this result, we can enumerate fault-tolerant frequent itemsets with frequent itemset mining implementations. This contribution demonstrates the extensibility of frequent itemset mining algorithms to mine interesting itemsets (An instance of Enumeration Problem 3).

1.3 The Summarization Problem

As mentioned before, the number of interesting itemsets can be abundant, which limits their interpretability. The summarization problem aims to present the interesting itemsets into an interpretable form. Normally, this can be achieved by (slightly) compensating the completeness/accuracy of the desired knowledge. This objective is analogous with the document summarization problem, where we want to concisely represent a long document into a much shorter paragraph or article. An ideal summary should be concise and yet contain information similar to the original document. Naturally, a summary or a representation should contain a set of patterns, where each individual pattern exhibits the knowledge of some interesting itemsets.

Formally, given a database, an interestingness criteria μ (and the threshold σ), and the summarization measure ν , the objective of the summarization problem is to find a summary or a representation R that maximizes $\nu(R)$. The function $\nu(R)$ represents the quality of the summary, which can be formulated as a function of diversity, conciseness, spuriousness, and coverage depending on the application [29]. Normally, $\nu(R)$ is formulated as a set of constraints and one objective function [103, 104]. For example, analysts might desire a summary with at most K patterns (conciseness constraint), while maximizing the accuracy, or vice versa.

1.3.1 Analysis Methodology

As the sheer volume of the database prohibits manual analysis, analysts prefer to see and interpret a concise summary of the interestingness knowledge from the data.

1.3. THE SUMMARIZATION PROBLEM

9

In most real world scenarios, analysts interpret the summary by scanning and analyzing each pattern in it consecutively. As this interpretation depends upon human analyzability, we cannot simply measure it using complexity analysis.

Interpretability

The analyst should be able to infer the interesting knowledge in the database by simply reading the summary. Naturally, we cannot expect the analyst to remember all previously read patterns or perform complex mathematical operations on them to infer some additional knowledge. For example, it would be difficult to derive the maximum entropy distribution of an itemset, assuming we know the frequency of all its two-subsets, by scanning the summary [88]. Based on these requirements, we postulate the properties for a summary to be interpretable:

Property 1: *Each summary pattern should be interpretable by itself, i.e., it should be semantically meaningful.*

Property 2: *The analysis process should not require concurrent analyses of several summary patterns.*

Having satisfied these two properties, the data analyst can scan the patterns and analyze each pattern individually. One example of a common summarization technique that satisfies these properties is in the document summarization problem, where the summary is presented in bullet or point form. In this case, each point is interpretable by itself, and its content is independent of other points. After scanning all points, the analyst should be able to infer the content of the document.

Besides allowing independent and easy inference, an interpretable summary should also be concise. That is, the summary should not be too large (contain too many patterns), as it will be tedious and expensive to analyze. By optimizing conciseness, we indirectly prevent redundant itemsets from appearing in the summary.

Accuracy

Besides being interpretable, a summary should also be *accurate*. Which means, the knowledge inferred by a summary should be similar to the original data. From this perspective, a summary can be treated as an information-retrieval system of the interesting itemsets. If the summary is R , and the knowledge inferred by a pattern $r \in R$ is $f(r)$, the overall knowledge inferred by the summary can be computed as $\bigcup_{r \in R} f(r)$.

Using the information-retrieval terminologies, the summary should have high precision and recall. In other words, the summary should not contain many misleading information (high precision), and should comprehensively capture the information (high recall). Normally, there is a trade-off between precision and recall, i.e., we can optimize precision by only representing few patterns, and we can optimize recall by representing a lot of patterns. The most common way to combine these two measures is by computing the F_1 measure, which is the harmonic mean of precision and recall.

1.3.2 Problems and Challenges

The problem of *concisely representing* frequent itemsets has become popular in the past few decades. However, most approaches ignore the interpretability requirements, by instead *archiving* or *indexing* the itemsets. The objective of archiving is to form a representation, such that the input patterns can be extracted back [22, 94]. Unlike a summary, an archive is obviously not interpretable, as demonstrated by the usage of *Huffman encoding* to summarize a document, or *cosine transformation* to summarize an image. The objective of indexing is to efficiently answer the interestingness query of an itemset with reasonable accuracy [63]. In this sense, an index is not interpretable since it relies on query, and by itself does not present the interesting itemsets.

In order to produce an interpretable and effective summary, we need to decide the appropriate *form* of the summary patterns. Having the form, we then need to design the summarization algorithm to produce the best summary.

Clearly, different applications require different knowledge, and hence different summarization schemes. In this thesis, we tackle three popular variants of the interesting itemsets summarization problem, as follows:

SUMMARIZATION PROBLEM 1: *Summarizing interesting itemsets, where the knowledge is formed as frequent itemsets and their frequency.*

This problem is the most popular variant of the itemset summarization problem. However, as mentioned before, most of the existing results do not produce an interpretable summary [22, 94]. Among those which are interpretable, none of them can produce an accurate and concise summary [61, 24]. Therefore, it is important to solve this well-grounded problem. This research problem is discussed in Chapter 6.

SUMMARIZATION PROBLEM 2: *Summarizing interesting itemsets, where the knowledge is formed as itemsets and their associated transactions.*

This is another popular variant of the itemset summarization problem [103, 57].

1.3. THE SUMMARIZATION PROBLEM

11

In this problem, the main issue of existing solutions is that they do not relate the solutions with the information-retrieval measures, viz., precision and recall. Therefore, all previous works consider either precision (by maintaining perfect recall) or recall, but not both. This is not effective as perfection is generally not required in real application. By maintaining perfect recall, it is likely that the summary would have low precision and conciseness, and vice versa. Our objective is to design a summarization scheme that compensates both precision and recall effectively, in order to produce the most concise summary. This research problem is discussed in Chapter 7.

Summarization Problem 3: *Summarizing interesting itemsets, where the knowledge is formed as itemsets (feature-sets) representing objects in a class.*

A set of objects (transactions) can be described by a set of features (itemset), which are commonly satisfied by those objects, and not satisfied by other objects. The objective of this problem is to find a set of itemsets that accurately describe all objects in a class. The natural solution of this problem is to build a classifier. However, existing classification techniques emphasize too much on accuracy, while ignoring the interpretability of the description. Our objective is to design a very concise and interpretable, yet accurate description of a concept. This research problem is discussed in Chapter 8.

1.3.3 Contributions

A desirable summary should be interpretable, concise, and accurate. In this thesis, we propose a summary form and the algorithm to construct the summary for the three aforementioned problems:

- For the Summarization Problem 1, we propose a new summary form called *conditional profile*, which appends the pattern profile by a *base* specifying the condition (details in the respective chapter). Our algorithm produces a summary which has no false positives or negatives and provides an error guarantee on the accuracy, as specified by the user [73].
- For the Summarization Problem 2, we manage to effectively compromise the precision and recall of the summary. Our proposed algorithm is much more effective than the approaches which compromise each property separately [78].
- For the Summarization Problem 3, we expand the coverage of each itemset (sentence) to cover objects (transactions) partially containing the itemset. Our

proposed algorithm can accurately describe a class in a database with very few sentences [75].

1.4 Organization

This thesis discusses two main problems, namely the itemset enumeration problem and the itemset summarization problem. The itemset enumeration problem is discussed in Chapters 2 to 5. In particular, Chapter 2 discusses the basic enumeration problem, which includes the mining variants, the properties of interesting itemsets, and the calculation of interestingness function. In other words, this chapter presents the *basic framework* to mine interesting itemsets with various properties. This chapter also discusses several applications (functions) which can be trivially derived by the mentioned algorithms.

However, for some sophisticated measures, simply applying the *basic framework* would not yield efficient solution. That is, there could be several specifically-tailored pruning schemes to boost the efficiency. This thesis discusses two of such functions, namely correlated itemsets and fault-tolerant frequent itemsets (FTFI), in Chapters 3 and 4 respectively. Finally, Chapter 5 discusses the reduction from the fault-tolerant frequent itemset mining to the frequent itemset mining problem.

The itemset summarization problem is discussed in Chapters 6 to 8. In particular, we discuss the problem of summarizing frequent itemsets in Chapter 6, summarizing subspace-clusters in Chapter 7, and summarizing itemsets for describing a class in Chapter 8. In each chapter, we discuss the motivation, the challenges, and our proposed solutions for each respective problem.

Finally, this thesis is concluded in Chapter 9 with future research directions.

Chapter 2

Enumerating Interesting Itemsets

This chapter discusses the problem of enumerating interesting itemsets. The solution framework for this general problem consists of two modules, viz., the candidate generation and the interestingness computation modules [71, 74, 70]. This framework has been highly optimized for the specific problem to enumerate *frequent itemsets*, due to the anti-monotone property and the ease for calculating the frequency or support. However, applying those optimization techniques for enumerating *interesting itemsets* is difficult, because general interestingness measures lack anti-monotone property, and are also difficult to compute.

In this chapter, we classify the enumeration problems into three variants, viz., to enumerate *all* interesting itemsets, *maximal* interesting itemsets, and *top-K* interesting itemsets. We also analyze the properties of the interestingness measures, with respect to the problem variants and the modules of the solution framework. This categorization allows us to find the best optimization techniques for tackling each problem variant.

We also propose efficient algorithms for *calculating the interestingness* of an itemset. These algorithms operate at a cost that is only slightly higher than what is required for calculating its support.

2.1 Preliminaries

This section presents the basic concepts for the itemset enumeration problem, which includes the problem variants and the enumeration framework.

2.1.1 Interestingness Measure

In order to find the useful knowledge, or interesting patterns, in a database, we first need to formulate the notion of interesting itemsets. As mentioned in the introduction, there are various interestingness measures, which can be defined based on correlation, uncertainty, actionability, etc.

In this chapter, our objective is not to define the best interestingness measure, nor to tackle specific measures. Instead, we aim to analyze useful properties of interestingness measures, which can be exploited to speed up the enumeration process. Having these useful properties, we can enumerate interesting itemsets by simply analyzing the properties of the interestingness measure.

For the sake of generality, we assume that there is an *interestingness measure* $\mu(X)$ that specifies how interesting an itemset is, i.e., the larger $\mu(X)$, the more interesting the itemset. One famous instance is when analysts are interested in *frequent* itemset. In this case, $\mu(X)$ is the support of X .

2.1.2 Problem Variants

There are two general variants of the itemset enumeration problem. The first is to enumerate all interesting itemsets, and the second is to enumerate the top- K most interesting itemsets. For the former problem, the user specifies the interestingness threshold σ , and the system should return all itemsets with $\mu(X) \geq \sigma$. For the latter, the user specifies the desired number of itemsets K , and the system should return the K most interesting itemsets. We denote the former as the *threshold* variant, and the latter as the *top- K* variant.

The top- K variant is generally considered more useful, as the number of desired itemsets K is easier and more intuitive to set than the threshold [27, 38]. Moreover, we can always terminate the program early if the interestingness of the returned itemset is already too low. The advantage of the threshold variant is that it is usually easier to enumerate, i.e., it allows more efficient algorithms.

In some applications, the knowledge provided by an itemset subsumes that of its subsets. Therefore, knowing that an itemset X is interesting implies that all itemsets $Y \subset X$ are also interesting. For example, knowing that all pairs of items in X are highly correlated implies all pairs of items in $Y \subset X$ are also highly correlated. Another famous example is regarding the frequentness of an itemset, i.e., if an itemset is frequent (with respect to a threshold), then all its subsets are also frequent. In these cases, analysts are interested only in interesting itemsets, which have no interesting

Mining Target	Required Parameter
Interesting Itemsets	σ
Maximal Interesting Itemsets (MII)	σ
Top- K Interesting Itemsets	K (and σ)

Table 2.1: Variants of the interesting itemset enumeration problem.

supersets. Such itemsets are denoted as *maximal interesting itemsets*. Considering this maximality criterion, we have two sub-variants of the threshold variant itemset enumeration problem, viz., to enumerate all interesting itemsets and to enumerate maximal interesting itemsets. For easier presentation, we separate the variant with and without maximal restrictions, hence resulting in three problem variants as presented in Table 2.1.

As the analysis of some algorithms requires the number of interesting itemsets, we denote this number as K , which is consistent with the notion of top- K interesting itemsets.

2.1.3 Itemset Enumeration Framework

Interesting itemset enumeration algorithms can be described by a high-level framework that consists of two sub-modules, viz., *candidate generation* and *interestingness computation*. The *candidate generation* module prompts the itemsets whose interestingness should be computed, and depending on whether the prompted itemsets are interesting, updates the set of *candidate itemsets*. A candidate generation module might not prompt each itemset one-by-one, instead, it may prompt a bunch of itemsets for the interestingness to be calculated. The *interestingness computation* module, as named, computes the interestingness of a given itemset, based on the desired measure.

2.1.4 Basic Notations

We deal with a transactional database D which contains of a set of n transactions T over a set of m items $I = \{a, b, c, \dots\}$. For brevity, we write a set $\{a, b, c\}$ as a sequence of characters abc . A function $I(y), y \in T$ represents the itemset contained by transaction y . Similarly, a function $T(x), x \in I$ represents the transaction-set (tidset) containing item x . Itemset with size k is denoted as k -itemset. A k -subset of an itemset X is a subset of X which consists of k items. A subset of itemset X

with size $|X| - 1$ is denoted as its direct subset, and a superset of X with size $|X| + 1$ is denoted as its direct superset. The density of a database D is calculated as the total number of items in all transactions, divided by the maximum possible number of items. When D is presented in a matrix form as in Figure 1.1(b), the notion of density can be seen as the proportion between number of cells with value ‘1’ and the total number of cells.

A transaction y *supports* an itemset X if it contains all items in X , or mathematically, $I(y) \supseteq X$. The supporting transactions or the cover of an itemset X , denoted as $\mathcal{S}(X)$, is the set of all transactions that contains X , or mathematically, $\mathcal{S}(X) = \{y | I(y) \supseteq X\}$. The (absolute) support of an itemset X , $sup(X)$, is calculated as the number of transactions that support X , which is $|\mathcal{S}(X)|$. The relative support of an itemset X , $rsup(X)$, is calculated as the proportion between $sup(X)$ and the number of transactions n .

2.1.5 Organization of the Chapter

This chapter is organized as follows. We first analyze individual modules, viz., the candidate generation and the interestingness computation modules, regarding the properties of the interestingness measures affecting the respective module, and also the algorithms for each property, in Sections 2.2 and 2.3 respectively. The properties of these algorithms are then summarized in Section 2.4. Section 2.5 presents how several interestingness measures can be presented and analyzed by our categorization/framework. This section is followed by the empirical evaluations of our algorithms for various applications in Section 2.6.

2.2 Candidate Generation

There are two properties that affect the design of the candidate generation algorithm. The first is the mining target. As mentioned before, there are three variants of interesting itemset enumeration problem, viz., all interesting itemsets, maximal interesting itemsets, and top- K interesting itemsets. The second is the monotonicity property of the measure μ .

This section aims to discuss the algorithms for all those variants, and is organized as follows. Firstly, we discuss the monotonicity properties of interesting itemsets. Next, we discuss the algorithms for enumerating interesting itemsets with each monotonicity property, with each problem variant.

2.2. CANDIDATE GENERATION

17

Property	Characteristic						
Anti-monotone [4]		<i>all</i>		<i>subsets</i>		<i>less</i>	
Monotone	the	<i>all</i>		<i>subsets</i>		<i>more</i>	
Weak anti-monotone [15]	interestingness	<i>some</i>	direct	<i>subsets</i>	is no	<i>less</i>	than the
Weak monotone	of	<i>some</i>		<i>subsets</i>		<i>more</i>	itemset
Supweak monotone		<i>some</i>		<i>supersets</i>		<i>less</i>	
Supweak anti-monotone		<i>some</i>		<i>supersets</i>		<i>more</i>	

Table 2.2: Monotonicity properties of interestingness measures.

2.2.1 Monotonicity Properties

In order to design an efficient algorithm for enumerating interesting itemsets, the first thing to do is to analyze the property of the interestingness measure. In this thesis, we discuss the monotonicity properties of itemsets, namely strong/weak/supweak monotone and anti-monotone properties, resulting in six combinations in total. If the context is clear, the term *strong* can be omitted. These measures are formally defined as follows.

Definition 2.1. [Strong anti-monotone] A measure μ satisfies the strong anti-monotone property if $\mu(X)$ is upper bounded by $\mu(Y)$, for all Y subsets of X , or mathematically, $\mu(X) \leq \mu(Y) \forall Y \subset X$. Equivalently, $\mu(X)$ is lower bounded by $\mu(Z)$, for all Z supersets of X .

Definition 2.2 (Strong monotone). A measure μ satisfies the strong monotone property if $\mu(X)$ is lower bounded by $\mu(Y)$, for all Y subsets of X , or mathematically, $\mu(X) \geq \mu(Y) \forall Y \subset X$. Equivalently, $\mu(X)$ is upper bounded by $\mu(Z)$, for all Z supersets of X .

In short, the *anti-monotone* property means that the larger the itemset (the more the number of items), the less interesting the itemset. Similarly, the *monotone property* means that: the larger the itemset, the more interesting the itemset. The term *strong* means that the restriction, which is the lower and the upper bound of interestingness, is applied to *all* subsets and (consequently) *all* supersets of each itemset. These properties are standard mathematic terminologies, and were first introduced in data mining in [4].

Next, we define the *weak* properties as follows.

Definition 2.3 (Weak anti-monotone). A measure μ satisfies the weak anti-monotone property if $\mu(X)$ is upper bounded by the *maximum* interestingness of its subset with

one item less. Mathematically, $\mu(X) \leq \max_{a \in X} \mu(X \setminus a)$. In other words, for every itemset X , there is an item $a \in X$ such that $\mu(X) \leq \mu(X \setminus a)$.

Definition 2.4 (Weak monotone). A measure μ satisfies the weak anti-monotone property if $\mu(X)$ is lower bounded by the *minimum* interestingness of its subset with one item less. Mathematically, $\mu(X) \geq \min_{a \in X} \mu(X \setminus a)$. In other words, for every itemset X , there is an item $a \in X$ such that $\mu(X) \geq \mu(X \setminus a)$.

The *weak* property was coined in [105] and [85], where they describe a definition with the weak anti-monotone property. For example, [85] named a measure *weak density* because it satisfies the weak anti-monotone property. However, in those papers, the term “weak anti-monotone” has not been introduced. This definition is formalized in [15] with the name *loose anti-monotone*. To the best of our knowledge, the *weak monotone* property has never been discussed in literature, and is included here for the sake of completeness.

Lastly, we define the *supweak* properties as follows.

Definition 2.5 (Supweak anti-monotone). A measure μ satisfies the supweak anti-monotone property if $\mu(X)$ is lower bounded by the *minimum* interestingness of its superset with one item more. Mathematically, $\mu(X) \geq \min_{a \notin X} \mu(Xa)$. In other words, for every itemset X , there is an item $a \notin X$ such that $\mu(Xa) \leq \mu(X)$.

Definition 2.6 (Supweak monotone). A measure μ satisfies the supweak monotone property if $\mu(X)$ is upper bounded by the *minimum* interestingness of its superset with one item more. Mathematically, $\mu(X) \leq \max_{a \notin X} \mu(Xa)$. In other words, for every itemset X , there is an item $a \notin X$ such that $\mu(Xa) \geq \mu(X)$.

The *supweak* property is the counterpart of *weak* properties. That is, while the *weak* property restricts the existence of more/less interesting direct subset (subset with one item less), the *supweak* property restricts the existence of more/less interesting direct superset (superset with one item more). This property is introduced in this thesis, and its applicability is demonstrated in Chapter 3, where we prove that the *correlation* of an itemset satisfies the *supweak monotone* property. All these properties are summarized in Table 2.2.

Other Related Properties

Several interesting properties such as *succinct* and *convertible* [67, 68, 66] are related to the monotonicity properties of itemsets. These properties are briefly discussed

2.2. CANDIDATE GENERATION

19

here to maintain the coherence of this chapter. A good survey on these properties is presented in [15], and a system to mine interesting itemsets with these properties is presented in [14].

A measure is considered *succinct* if its interestingness, i.e., whether it satisfies the interestingness criteria, can be determined by observing a single item. The example of measure satisfying this property is *minimum frequency of each item in the itemset*, i.e., $f(X) = \min_{x \in X} \text{sup}(X)$. With this measure, an itemset is not interesting if it contains any single item which is not frequent. We can mine interesting itemsets with this property by removing all uninteresting (or infrequent, in the previous case) items. The succinct property is a special case of the anti-monotone property.

The *convertible anti-monotone* measures deals with *ordered items*. That is, if we sort the items in non-increasing order of interestingness, extending an uninteresting itemset with another item *with larger index* will never produce interesting itemsets. Similarly, removing the item with the largest index from an interesting itemset will always produce interesting itemset. One example of this measure is the average frequency of items, when we require it to be no less than a minimum threshold μ . Interesting itemsets with convertible anti-monotone property can be mined by first ordering the items, and then using depth-first enumeration (Section 7) as in the case of anti-monotone. The convertible anti-monotone property is the generalization of the anti-monotone property, and the specialization of the weak anti-monotone property [15].

The *convertible monotone* property is the converse of the *convertible anti-monotone* property. That is, extending interesting itemsets with another item with larger index will always produce interesting itemsets, and removing items with the largest index from non-interesting itemsets will always produce non-interesting itemsets. Interesting itemsets with the convertible monotone property can be mined in a similar way as those with the monotone property, after sorting the items (Section 2.2.4).

As these properties are the special cases of monotonicity properties, our analyses and algorithms in this chapter can be directly applied for enumerating interesting itemsets with these properties.

Level	Itemset
1	a, b, c, d
2	ab, ac, bc, ad
3	abc

Table 2.3: Sample interesting itemsets for measures satisfying the anti-monotone property

2.2.2 Algorithms for Measures with the Anti-Monotone Property

In general, there are three algorithms for candidate generation, when the interestingness measure satisfies anti-monotone property. The first is called *Apriori* or *Breadth-First* (BF). This algorithm generates the candidate itemsets level-by-level, where the *level* corresponds to the number of items in the itemset. In brief, this algorithm generates all candidates in the same level, verifies their *interestingness*, and then generates the candidates in the next level, and so on.

The second is called *Depth-First* (DF). This algorithm repeatedly splits the search space into those containing an item, say item a , and those not containing it. To enumerate all interesting itemsets containing item a , this algorithm repeats the same procedure by re-splitting the search space into those containing another item, say item b , and those not containing it, and so on.

The third is called *Reversed-Depth-First* (RDF). RDF algorithm is essentially the DF algorithm in the reverse order. In other words, this algorithm first enumerate all interesting itemsets that does *not* contain item a , before those containing item a . The significance of the reverse-ordering is that an itemset is always traversed later than its subsets.

To illustrate the algorithms, we use interesting itemsets in Table 2.3.

Apriori / Breadth-First (BF)

The Apriori algorithm [5] generates the candidate level-by-level, as presented in Algorithm 2.1. First, this algorithm stores all interesting items (or 1-itemsets) in level one. Because of the anti-monotone property of the interestingness measure, an itemset can possibly be interesting only if all its subsets are interesting. The candidates k -itemsets are then generated as those whose *all* direct subsets (in level $k - 1$) are interesting.

2.2. CANDIDATE GENERATION

21

Algorithm 2.1: Apriori/BF Algorithm

```

1  $k \leftarrow 1$ ;
2 initialize  $C_1 \leftarrow \{a | \mu(a) \geq \sigma\}$ ;
3 while  $C_k$  is not empty do
4    $I_k \leftarrow \{X \in C_k | \mu(X) \geq \sigma\}$ ;
5   pruning: Remove items which do not appear in any itemset  $X \in I_k$ 
6    $C_{k+1} \leftarrow$  itemsets of size  $k + 1$ , whose all subsets are in  $I_k$ ;
7    $k \leftarrow k + 1$ ;

```

Algorithm 2.2: Depth-First Algorithm

```

1 ENUMERATE(itemset  $X$ , item[ ]  $Tail$ )
2 begin
3    $Tail' \leftarrow \{a \in Tail | \mu(Xa) \geq \sigma\}$ ;
4   for item  $a$  in  $Tail'$  do
5     ENUMERATE( $Xa$ ,  $Tail' \setminus \{a\}$ );
6    $Tail' \leftarrow Tail' \setminus \{a\}$ ;

```

Example 2.1. Consider interesting itemsets in Table 2.3. Initially C_1 contains all interesting items, a , b , c , and d . We do not need to consider uninteresting itemsets, e.g., item e , because according to the anti-monotone property (Definition 2.1, there is no interesting itemset containing item e . After processing the 1-itemset, we initialize the candidate interesting 2-itemsets, C_2 , as all 2-itemsets containing only a , b , c , d , which are: ab , ac , ad , bc , bd , and cd , and then filter only the interesting ones: ab , ac , bc , and ad . The candidate 3-itemsets, C_3 , is then set as 3-itemsets whose all subsets are interesting, which is abc .

One pruning strategy is presented in Line 5 to reduce the database size, by removing all items which do not appear in any interesting itemset in the current level. This is useful, as removing items means reducing the size of the database. It is natural that working with smaller databases is more efficient than working with larger databases. For example, by removing items, many transactions might become empty which can be removed trivially. With fewer transactions, it is obvious that iterating the database takes less time.

Depth-First (DF)

The Depth-First (DF) algorithm splits the generated candidates into two parts, viz., those containing an item, say item a , and those not containing it. To generate candidates containing item a , the algorithm again splits the search space into two parts, i.e., those containing item b and those not containing it. Generating candidates not containing item a can simply be done by removing this item from the database. This algorithm is presented in Algorithm 2.2, where the parameter X is the *current* itemset (the algorithm currently generates all candidate itemsets containing itemset X), and the parameter $Tail$ represents the set of all items that can extend X . Initially, this function is called with X as an empty set and $Tail$ as all interesting items. This algorithm first filters only interesting extension in $Tail$ (Line 3). For each interesting extension, lets say item a , DF generates all candidates containing item a by recursively calling the ENUMERATE function (Line 5), and then removes this item from the extension (Line 6). Note that the itemset X is always interesting.

Example 2.2. Consider interesting itemsets in Table 2.3. Assuming that the items are ordered lexicographically, the DF algorithm first calls ENUMERATE(a), where $Tail = \{b, c, d\}$. Next, it calls ENUMERATE(ab), where $Tail = \{c, d\}$. As abd is not interesting, it is filtered in Line 3 (Algorithm 2.2). Next, it calls ENUMERATE(abc) with empty $Tail$. After mining interesting itemsets containing item a , the algorithm then calls ENUMERATE(b), with $Tail = \{c, d\}$. Compared with the Apriori algorithm, this algorithm checks the interestingness of several itemsets, which are not considered by the Apriori algorithm, such as itemsets abd and acd .

This way of removing the element (item) in the $Tail$ after being enumerated is called *tail extension*. Regardless of the order of the items in the $Tail$, the DF algorithm always enumerates complete interesting itemsets without duplication. However, the order of the items does affect the efficiency. In most cases, the optimal efficiency is achieved by ordering the tail in increasing order of support [16].

There are two famous DF algorithms for enumerating frequent itemsets, namely ECLAT [107] and FP-Growth [37]. We do not discuss the details of these two algorithms, as they are specific to the frequent itemset mining problem. However, some key concepts of the algorithms are sketched when we discuss the algorithm for calculating the exact support of itemsets in Section 2.3.2.

Compared with Apriori, DF has several advantages and disadvantages. One disadvantage is that DF generates more candidates than Apriori. For example, DF may calculate the interestingness of itemset ab even though itemset b is not interesting,

Algorithm 2.3: Reversed-Depth-First Algorithm

```

1 ENUMERATE(itemset  $X$ , item[ ]  $Tail$ )
2 begin
3    $Tail \leftarrow \{a \in Tail \mid \mu(Xa) \geq \sigma \wedge \forall b \in X, \mu(Xa \setminus b) \geq \sigma\};$ 
4    $Tail' \leftarrow \emptyset;$ 
5   for item  $a$  in  $Tail$  do
6     ENUMERATE( $Xa$ ,  $Tail'$ );
7      $Tail' \leftarrow Tail' \cup \{a\};$ 

```

and only because a is interesting. To formalize, for DF, an itemset is a candidate if all its *prefixes* are interesting, where the order of items in the itemset is defined by the algorithm. This is less effective than Apriori, where the candidates are those whose all *subsets* are interesting.

Despite generating more candidates, DF is usually faster than Apriori because of the following advantages. First, while generating candidates, DF only extends the *current* itemset by one item, unlike Apriori which completes each level before proceeding to the next level. These extension operations allow faster computation of interestingness, which we will show later in Section 2.3. Second, DF shrinks the database faster than Apriori, as an item is removed right after the enumeration (Line 6). Whereas, Apriori removes items per level k , and only if the respective items never appear in any interesting itemset of size $k - 1$.

Reversed Depth-First (RDF)

The Reversed Depth-First (RDF) algorithm [20] is similar to DF, and is presented in Algorithm 2.3. The difference between the two is that, unlike DF, RDF enumerates the itemset *not* containing an item a before those containing item a . Interestingly, by changing this order, it ensures that all the subsets are enumerated before the itemset itself. This property allows the algorithm to avoid calculating the interestingness of an itemset, if any of its subsets are not interesting (Line 3).

Example 2.3. Consider interesting itemsets in Table 2.3. Assuming that the items are ordered lexicographically, the RDF algorithm first calls ENUMERATE(a) with $Tail = \emptyset$, retrieving interesting itemset a . In the next iteration, it calls ENUMERATE(b) with $Tail = \{a\}$, followed by ENUMERATE(ab) with empty $Tail$. Next, the algorithm calls ENUMERATE(c) with $Tail = \{a, b\}$. Observe that all interesting subsets of $Tail$ have been previously enumerated.

Algorithm 2.4: GENMAX: Enumerating Maximal Interesting Itemsets

```

1 ENUMERATE(itemset  $X$ , item[ ]  $Tail$ , projected MII  $\mathcal{M}$ )
2 begin
3    $Tail \leftarrow \{a \in Tail \mid \mu(Xa) \geq \sigma\};$ 
4    $Tail' \leftarrow Tail;$ 
5   if  $Tail' = \emptyset \wedge \mathcal{M} = \emptyset$  then
6     output  $X$ ;
7     add  $X$  to the list of MII (update  $\mathcal{M}$  in every recurrence level);
8     return;
9   for item  $a$  in  $Tail$  do
10    if  $\exists Y \in \mathcal{M} \mid Tail' \subseteq Y$  then
11      return;
12     $Tail' \leftarrow Tail' \setminus \{a\}$ 
13     $\mathcal{M}' \leftarrow \{Y \in \mathcal{M} \mid a \in Y\};$ 
14    ENUMERATE( $Xa$ ,  $Tail'$ ,  $\mathcal{M}'$ );

```

This algorithm has the combined advantages of Apriori and DF, i.e., it generates the candidates by extending the current itemset by one item, reducing the database aggressively (as in DF), and produces a small number of candidates (as in Apriori). The drawback is in the large memory requirement to store all interesting itemsets, typically in a hash-table (otherwise, RDF is equivalent to DF), and non-negligible time taken to retrieve those interesting itemsets. In most cases, the benefits of pruning more candidates is not significant compared to the cost of maintaining and accessing the hash-table. The main advantage of RDF over DF is in the guarantee that the subsets are enumerated before the itemset, as some sophisticated interestingness computation algorithms require the support/interestingness of the subsets (as explained in Section 2.3).

Enumerating Maximal Interesting Itemsets (MII)

An interesting itemset X is considered maximal if it has no interesting superset. One way to enumerate maximal interesting itemsets is by post-processing all interesting itemsets, i.e., we first enumerate all interesting itemsets, and then filter-out those which are not-maximal by traversing the lattice of interesting itemsets [16]. However, this approach is inefficient as the number of MIIs is usually much less than the number of interesting itemsets. It is desirable to traverse only the MIIs, or to minimize the number of candidates generated.

2.2. CANDIDATE GENERATION

25

The state-of-the-art algorithm for enumerating maximal interesting itemsets is GENMAX [31], which is illustrated in Algorithm 2.4. GENMAX uses depth-first enumeration, and maintains the set of all maximal interesting itemsets, projected to the current itemset. Because of the depth-first traversal, an itemset is maximal if and only if (1) it is not a subset of previously found maximal itemsets, and (2) there is no item that can extend this itemset. This maximality criterion is checked in Line 5. Whenever a maximal itemset is found, GENMAX updates the list of maximal itemsets in all recursion levels. This algorithm is improved in [32], by using a prefix-tree to speed-up the subset checking (Line 10).

Example 2.4. Consider interesting itemsets in Table 2.3, and assume that the items are ordered lexicographically. While enumerating the itemsets, the algorithm calls $\text{ENUMERATE}(a)$ with $Tail = \{b, c, d\}$, $\text{ENUMERATE}(ab)$ with $Tail = \{c\}$ (itemset abd is not interesting), and $\text{ENUMERATE}(abc)$ with empty $Tail$. In this phase, we know that abc is interesting and maximal. In the following iterations, let's say when the algorithm iterates itemset ac or bc , we do not need to calculate their interestingness as they are not maximal. Another pruning scheme is also applicable when the algorithm calls $\text{ENUMERATE}(b)$ with $Tail = \{c\}$ (bd is not interesting). In this case, since $b \cup Tail$ is a subset of a maximal interesting itemset, which is abc , we can terminate the search immediately.

As GENMAX is based on DF enumeration, we can remove an item after enumerating the interesting itemsets containing that item. This database shrinking procedure allows the algorithm to prune the search space if the current itemset and the *remaining items* cannot form any maximal interesting itemsets (Line 11). Moreover, because of the anti-monotone property, if an itemset X is interesting, then all itemsets $Y \subset X$ are also interesting. Therefore, once an itemset X is found, we do not need to calculate the interestingness of any generated itemset $Y \subset X$.

The objective of projecting the maximal interesting itemsets to the current itemset is to speed-up the subset-checking procedure, as only few (and small) maximal itemsets need to be checked. We can further reduce the size of projected maximal itemsets using the database shrinking technique, i.e., by removing the removed item from the list of all maximal interesting itemsets.

Note that it is impossible to extend the Apriori and RDF algorithms for enumerating MIIs without traversing all interesting itemsets, as these algorithms ensure that all subsets of an itemset are enumerated before that itemset.

Algorithm 2.5: Enumerating top- K Interesting Itemsets

```

1 Insert all items to the priority-queue  $pq$ ;
2  $Result \leftarrow \emptyset$ ;
3 for  $|Result| < K$  do
4    $X \leftarrow pq.pop()$ ;
5    $Result \leftarrow Result \cup X$ ;
6   for  $a \notin X$  do
7     if  $\forall b \in Xa, Xa \setminus b \in Result$  then
8        $pq.push(Xa)$ ;

```

Enumerating Top- K Interesting Itemsets

The simplest way to enumerate the top- K interesting itemsets is by performing the usual enumeration with $\sigma = 0$, and keep updating the threshold σ to the interestingness value of the current K -th most interesting itemsets [99]. However, as this approach is not efficient, as it may traverse many non-interesting itemsets. The ideal way is to mine the itemsets in the decreasing order of interestingness, so that the top- K interesting itemsets are mined in exactly K iterations.

The algorithm for enumerating top- K interesting itemsets in decreasing order of interestingness was proposed in [27], and is presented in Algorithm 2.5. This algorithm stores all candidate itemsets in the priority queue, which prioritizes the more interesting itemsets, and breaks the ties by favoring the smaller itemset. In each iteration, this algorithm extends the current itemset with all other items. Because of the anti-monotone property, we only need to check the interestingness of those itemsets whose all subsets are already enumerated. For example, as an itemset abc is never more interesting than ab , ac , and bc , we can wait to enumerate abc after all its subsets are enumerated. This checking also guarantees that the interestingness of each candidate is calculated exactly once. Using the same reasoning, we can observe that it is impossible for this algorithm to miss any interesting itemset, and hence this algorithm is correct.

Example 2.5. Consider interesting itemsets in Table 2.3. Initially, all interesting items (a , b , c , and d) are put inside the priority queue. In the first iteration, we take the first item (lets say item a), and extend it with all other items, that is, ab , ac , and ad . However, all these extensions are not considered, as these itemsets (lets say itemset ab) have interesting subsets which have not been processed, i.e., b . In the next iteration, we take item b from the priority-queue, extend it with all other items,

Level	Itemset
1	a, b, c
2	ab, ac, ad, bd
3	bcd

Table 2.4: Sample interesting itemsets for measures satisfying the weak anti-monotone property

Algorithm 2.6: W-Apriori algorithm

```

1  $k \leftarrow 1$ ;
2 initialize  $C_1 \leftarrow \{a | \mu(a) \geq \sigma\}$ ;
3 while  $C_k$  is not empty do
4    $I_k \leftarrow \{X \in C_k | \mu(X) \geq \sigma\}$ ;
5   pruning: Remove items which are not appear in any itemset  $X \in I_k$ ;
6    $C_{k+1} \leftarrow$  itemsets of size  $k + 1$ , whose any one subset is in  $I_k$ ;
7    $k \leftarrow k + 1$ ;

```

resulting in itemsets ab , bc , and bd . In this case, we put ab inside the priority-queue, but not bc and bd .

As each iteration outputs one itemset, the time complexity of this algorithm is linear to K . The space requirement is also linear to K since we only need to store at most K itemsets, or equivalently, $K - |Result|$ itemsets in the priority queue.

2.2.3 Algorithms for Measures with the Weak Anti-Monotone Property

Unlike the anti-monotone property, the weak anti-monotone property only guarantees that at least one direct subset of an interesting itemset X is at least as interesting as X . For illustrating the algorithms for mining interesting itemsets with measures satisfying the weak anti-monotone property, we use sample interesting itemsets in Table 2.4.

The pioneering algorithm for enumerating interesting itemset with weak anti-monotone property extends the Apriori algorithm (Algorithm 2.1), by setting the *candidates* in level k as itemsets whose any subset in level $k - 1$ are interesting [105, 39, 15]. We denote this algorithm as *w-Apriori*, and present it in Algorithm 2.6. The correctness of this algorithm is trivial, as the candidate generation exactly follows the definition of weak anti-monotone property.

Algorithm 2.7: W-DF algorithm

```

1 ENUMERATE(itemset  $X$ , item[ ]  $Tail$ )
2 begin
3   for  $item\ a\ in\ Tail$  do
4     if  $\mu(Xa) \geq \sigma \wedge Xa.parent = X$  then
5        $\quad$ ENUMERATE( $Xa, Tail \setminus a$ );

```

Example 2.6. Consider interesting itemsets in Table 2.4. Initially, all interesting items, namely a , b , and c , are put to C_1 . Next, all extensions of C_1 (with one item) are put to C_2 , that is, C_2 contains itemsets: $\{ab, ac, ad, bc, bd, cd\}$. After filtering only the interesting ones, C_2 becomes $\{ab, ac, ad, bd\}$. As before, we store all extensions of C_2 into C_3 , that is: $\{abc, abd, acd, bcd\}$, and then filter into $\{bcd\}$. Lastly, C_4 stores all extensions of itemsets in C_3 , which is $abcd$, which is then filtered out.

We can also modify depth-first enumeration algorithm for mining with weak anti-monotone property. We denote this algorithm as *w-DF*, which is presented in Algorithm 2.7. The difference between DF and w-DF is that w-DF does not remove an item a from the *Tail* after enumerating all itemsets containing a . It is because there might be an interesting itemset that contains a , which has no interesting prefix that contains a . For example, it is possible that itemset ab is interesting, but itemset a is not interesting.

However, this modification raises a problem as an itemset can be traversed multiple times. For example, itemset abc can be traversed by extending itemset ab by item c , or extending itemset bc by item a . This issue can be handled by defining the parent-child relationship between interesting itemsets (Line 4), which is also called *reverse-search enumeration* [9]. This relationship can be illustrated as an edge in the enumeration tree.

One way to define the *parent* of an itemset is to set it as its interesting direct subset which is enumerated earliest. Verifying this notion of parent-child can be implemented by storing all interesting itemsets in a hash-table. For each itemset Xa , we first check its existence in the hash-table. If it exists, then this itemset is interesting and has already been enumerated, and hence the enumeration can be pruned. For the other case when Xa does not exist in the hash-table, we can either directly check the interestingness of Xa , or first check whether X is the parent of Xa (the conditions in the LHS and RHS of Line 4 respectively). For the former, we can ignore Xa if it is not interesting, and proceed the enumeration otherwise. For the latter, we check the

2.2. CANDIDATE GENERATION

29

number of interesting direct subsets of Xa . If it is exactly one, then X is the parent of Xa , and hence we can proceed to check its interestingness. The former technique might compute the interestingness of an uninteresting itemset multiple times, while the latter technique computes the interestingness of all candidates exactly once, but might perform the parent-checking multiple times.

Example 2.7. Consider interesting itemsets in Table 2.4, and assume that the items are ordered lexicographically. First, the algorithm calls $\text{ENUMERATE}(a)$ with $\text{Tail} = \{b, c, d\}$, followed with $\text{ENUMERATE}(ab)$ with $\text{Tail} = \{c, d\}$. Note that in each recursive call, we store the respective interesting itemset in a hash-table. As there is no interesting extension, the algorithm backtracks, and enumerates ac and ad . Next, the algorithm calls $\text{ENUMERATE}(b)$ with $\text{Tail} = \{a, c, d\}$. When it encounters itemsets ab , it notices that ab is already inside the hash-table, and hence it is ignored. The procedure continues, until it encounters item d , calls $\text{ENUMERATE}(bd)$ with $\text{Tail} = \{a, c\}$, and then $\text{ENUMERATE}(bcd)$ with $\text{Tail} = \{a\}$.

As mentioned above, we can either check the interestingness of an itemset, or calculate the number of interesting direct subsets, in order to minimize double counting. For example, when we extend itemset a to ab , we can either calculate $\mu(ab)$ directly, or check the number of interesting direct subset of ab , which is already enumerated. Assuming the latter, there is only one such a subset, which is a , and hence we know that ab has not been enumerated. Another example is when we extend itemset c to bc . In this case, there are two subsets of bc which are already enumerated, namely b and c . Therefore, we can prune the search immediately without calculating $\mu(bc)$.

We can also ensure that the interestingness of all candidates are computed exactly once without the need to do parent checking, by storing all candidates in the hash-table, regardless they are interesting or not. This way, we can simply check whether the respective candidate is inside the hash table. However, by doing this, the hash-table becomes extremely large, and hence the overall algorithm becomes inefficient.

Comparing the w-DF and w-Apriori algorithms, unlike the case when the measure satisfies the anti-monotone property, here both algorithms generate the same candidates. Moreover, w-Apriori stores all candidates in memory, similar with the variant of w-DF that stores all candidates in the hash-table, which is inefficient. To add, w-DF allows much more efficient interestingness computation because of the extension operation. Based on the above arguments, we conclude that w-DF is more promising, which is interesting as the w-Apriori algorithm is far more popular [105, 14]. This argument is empirically validated by our empirical evaluation.

Enumerating Maximal Interesting Itemsets

For mining maximal interesting itemsets, we can use the same strategy as the case with anti-monotone property. However, some non-maximal itemsets could be considered maximal during the enumeration. For example, itemset ab might not be extendable by any single item (abc and abd are not interesting), but itemset $abcd$ is interesting. This is possible, for instance, when bcd is interesting. In this case, itemset ab will mistakenly be treated as a maximal interesting itemset. This case can be efficiently handled by updating the list of maximal interesting itemsets whenever we extend the current itemset. For example, whenever we extend X to Xa , we remove any (currently treated as) maximal itemsets which are subsets of Xa . Those itemsets always contain item a (otherwise, they are already removed), which can be used to limit the checking.

Enumerating Top- K interesting Itemsets

We can also use similar algorithm with that for anti-monotone case for enumerating top- K interesting itemset with weak anti-monotone property. The only difference is that, instead of checking whether all the subsets are present, we check whether exactly one subset is present. This way of checking is consistent with the *parent-child* relationship defined earlier for enumerating all interesting itemsets, i.e., the parent of an itemset is its earliest enumerated interesting direct subset.

Example 2.8. Consider interesting itemsets in Table 2.4, and assume that the items are ordered lexicographically. Firstly, we store itemsets a , b , c , and d in the priority queue. Next, taking an itemset a , we extend it with all possible items, yielding itemsets ab , ac , and ad , and put those which are interesting and has exactly one previously enumerated direct subset into the priority queue. In this case, all the above itemsets are put inside the priority queue. Next, when we take itemset b and extend it into ab , we see that ab has two direct subsets which have already been enumerated, and hence, it is ignored.

2.2.4 Algorithms for Measures with the (Supweak) Monotone Property

The anti-monotone property guarantees that an itemset is not more interesting than all its subsets, whereas, the monotone property guarantees that an itemset is not

more interesting than all its supersets. From this observation, we can reuse the algorithms for enumerating interesting itemsets with an anti-monotone measure, to mine interesting itemsets with a monotone measure, by reversing the enumeration process. That is, we consider the *complement* of the itemsets. For example, if we consider itemset ab in the anti-monotone case, here we consider itemset $I \setminus ab$. Consequently, extending an item means removing an item, checking whether all subsets are frequent becomes checking whether all supersets are frequent, etc. This modification works for all algorithms and all mining variants.

While the complement of anti-monotone is monotone, the complement of weak anti-monotone is supweak monotone. That is, the weak anti-monotone property guarantees the existence of a non-less-interesting direct subset, and the supweak monotone property guarantees the existence of a non-less-interesting direct superset. Therefore, all algorithms for mining with weak anti-monotone property are admissible for those with supweak monotone property, by using the complement of the itemset.

2.2.5 Algorithms for Measures Without any Monotonicity Properties

All previous algorithms for all problem variants generate the candidates by traversing only the interesting itemsets, and keep extending them with an item. However, without any monotonicity properties, an interesting itemset might have no interesting direct subsets. Therefore, it is impossible to apply the same procedure, for completely enumerating interesting itemsets. With this limitation, many previous works consider the enumeration problem without any monotonicity properties hopeless, and then propose an incomplete greedy/heuristic strategy. The common strategy is by assuming the existence of some properties [105, 100]. However, this technique in general cannot guarantee, or even provide an error guarantee for, the completeness of the enumeration.

Pruning Strategy

One strategy to completely enumerate interesting itemsets without any monotonicity property is by deriving an upper bound function $ub(X)$ which never underestimates the interestingness of X , and satisfies any specified monotonicity properties, usually the anti-monotone property. As $ub(X)$ is, say, anti-monotone, enumerating all itemsets with $ub(X) \geq \sigma$ can be done using any standard enumeration algorithm. And as

$ub(X) \geq \mu(X)$, no itemset with $\mu(X) \geq \sigma$ is missed. Therefore, we can enumerate all interesting itemsets by post-processing itemsets with $ub(X) \geq \sigma$.

Usually, the function $ub(X)$ depends only on the items in X . In some cases, we can tighten $ub(X)$ by considering the *possible extensions* of X , i.e., we can derive a tighter bound for X by analyzing which items can extend X . Sometimes, we can even remove several transactions from consideration to tighten the bound.

The above mentioned techniques are abstract and not applicable in their present form, since different interestingness functions has different upper bounds. In this thesis, we present two strategies for enumerating *correlated* itemsets and *fault-tolerant frequent* itemsets, in Chapters 3 and 4 respectively.

Enumerating Top- K Interesting Itemsets

Another difficult task when there is no monotonicity property is for enumerating the top- K interesting itemsets. This is because our enumeration is based on the upper bound, but our notion of interesting itemsets is based on μ .

Usually, $ub(X)$ is highly correlated with $\mu(X)$. Therefore, we suggest to enumerate the itemsets in descending order of the upper bound, by keeping all itemsets with upper bound no less than the current K -th interesting itemsets (w.r.t. μ). As we enumerate itemsets in the order of $ub(X)$, while we are interested in the top- K itemsets according to $\mu(X)$, we may enumerate and store more than K itemsets in the priority queue, unlike the case when some monotonicity properties are satisfied.

We discuss one variant of this task in Chapter 3, where we mine top- K correlated itemsets in a categorical database.

2.3 Interestingness Computation

Previously, we discussed the algorithms for generating the candidates. In this section, we discuss how to calculate the interestingness of an itemset $\mu(X)$, having the candidate itemset X and the function μ . We also relate the computation algorithms with the enumeration algorithms, i.e., we analyze the admissibility of the computation algorithms when accompanied with specific candidate generation algorithms.

We restrict the interestingness measure μ to the family of functions which can be calculated using the database projected on X . In other words, $\mu(X)$ should not depend on any items not present in X . This family of measures subsumes most interestingness measures published so far. Because of this constraint, $\mu(X)$ can be

2.3. INTERESTINGNESS COMPUTATION

33

formulated as a function of the number of transactions that contains *specific items* of X , which we term as its *detailed support*.

Our objective is to develop efficient algorithms to calculate these detailed supports. Having the detailed supports, calculating the interestingness is straight forward. The mapping of the problem for enumerating interesting itemsets for several common interestingness measures into our framework is presented in Section 2.5.

2.3.1 Detailed Support

This subsection discusses the *detailed supports* of itemsets, which is the number of transactions containing specific parts of the itemsets. We discuss two types of detailed supports, which we term as *subset support* and *partial support*.

Subset Support (SS)

The traditional support function, or the exact support, counts the number of transactions containing all items in the itemset. We can generalize the definition, by counting the number of transaction containing a specific subset (which we call a *base*), of the given itemset. We denote such a support function as *subset-support*.

A transaction t supports an itemset X based on a subset (base) $B \subseteq X$, if t contains all items in B , and none in $X \setminus B$, or mathematically, $I(t) \cap X = B$. The subset-support of X on B , denoted as $sup_B(X)$, counts the number of transactions that support X on B . Similarly, the subset-supporting transaction of X on B , denoted as $\mathcal{S}_B(X)$, represents the set of transactions that support X on B . If the context is clear (in the presence of the base), the prefix *subset* can be omitted.

All subsets can be the base of an itemset X , and thus there are $2^{|X|}$ subset-supports of them. We can observe that for each transaction t , there is exactly one base $B \subseteq X$ such that t supports X based on B , which is $B = I(t) \cap X$. To derive the interestingness of an itemset X , we normally need all $2^{|X|}$ subset supports. Therefore, if the base is not specified, the term subset-supports refers to the set of subset-supports based on each $2^{|X|}$ subset.

Partial Support (PS)

As in the case of subset-support, we can generalize the traditional support function by allowing the transaction to contain part of the itemset. We denote such a support function as *partial support*.

A transaction t partially supports an itemset X with a criteria “ $\leq k$ ”, “ $= k$ ”, or “ $\geq k$ ” ($0 \leq k \leq |X|$) if t contains no more, equal to, or no less than k items of X , respectively. Mathematically, t partially supports X with a criteria “ $\leq k$ ” if $|I(t) \cap X| \leq k$, and so on. The partial support of X with a criteria $C \in \{\leq k, = k, \geq k\}$, denoted as $sup_C(X)$, counts the number of transactions that satisfy criteria C for an itemset X . Similarly, the partial supporting transactions of X with a criteria C , denoted as $\mathcal{S}_C(X)$, represents the set of transactions that satisfy the criteria C for X . If the context is clear (in the presence of the constraint), the prefix *partial* can be omitted.

Note that having one particular type of partial support, for example “ $\geq k$ ” for all k , is sufficient, since the remaining can be derived easily as:

$$sup_{=k}(X) = sup_{\geq k}(X) - sup_{\geq k+1}(X)$$

and

$$sup_{\leq k}(X) = n - sup_{\geq k+1}(X)$$

To derive an interestingness measure, we normally need the set of all partial supports, or one kind of partial supports with all possible k since the others can be derived from them. Therefore, if the criteria are not specified, the term partial supports refers to the set of (any one kind of) partial supports with all k between 0 and $|X|$ inclusively.

2.3.2 Algorithms for Computing Exact Support

In this section, we discuss the algorithm to calculate the exact support as the foundation of the algorithms for calculating the detailed support.

First we discuss the data structure used to represent the transactional database. There are two well-known data structure/representations, namely *horizontal structure* and *vertical structure* [95]. The former structure stores the list of items contained in each transaction, and the latter stores the list of transactions containing each item. These data structures for the database in the introduction (Figure 1.1) are illustrated in Figure 2.1.

Horizontal Data Structure

Horizontal structure can be implemented by a simple list/array of transactions, where each contains a list of items. This structure can be compressed by combining identical

Figure 2.1: Data structures for enumerating frequent itemsets.

transactions into one *weighted* transaction [91]. A more compact yet complicated (hence causing non-trivial house-keeping operations) way to store horizontal data structure is by storing them in a prefix tree [37, 16]. To calculate the support of an itemset using horizontal structure, we need to iterate the transactions and check whether the corresponding transaction contains the itemset.

Vertical Data Structure

Vertical structure can be implemented by a simple list/array of items, where each item contains a list of transaction-ids, representing the id of transactions containing the respective item. As a transaction contains an itemset if it contains all items in the set, we can find out the set of transactions containing an itemset, by intersecting the set of transactions containing each item. With this data structure, intersecting two sets containing a and b elements respectively takes $a + b$ iterations.

As the only needed operation is set-intersection, we can improve the efficiency by storing the sets as a bit-vector, i.e., the k -th bit of the vector is *on* or ‘1’ if transaction with id k contains the item/itemset, and *off* or ‘0’ otherwise. Using a bit-vector is efficient as we can do any set operations for one *word* of elements in one clock cycle. The word size w is typically either 32 or 64 in 32-bits and 64-bits machine, respectively. Therefore, if the number of transactions is n , intersecting the entire set takes $\lceil \frac{n}{w} \rceil$ iterations, regardless of the number of elements in the set. Comparing these two representations, we can observe that the former approach (by storing the list of transactions) is efficient if the database is very sparse, while the latter approach (by storing the set of transactions as a bit-vector) is relatively efficient if the database is

very dense.

Besides these two, there is a *hybrid* approach, denoted as *sparse bit-set* data structure, which has the benefit of both approaches. This approach stores the set of transactions as a list of words. A word appears in the list if at least one transaction among the w transactions it represents, exists in the set. Using this data structure, the number of words contained in the list is upper bounded by the number of elements in the list (as in the first approach) and $\lceil \frac{n}{w} \rceil$ (as in the second approach). With this data structure, intersecting two sets containing w_a and w_b words respectively takes $w_a + w_b$ iterations.

In practice, the hybrid approach is the most efficient, unless the set is extremely dense (say, more than 50% of the set is filled-up). In such a case, the approach using bit-vectors is more efficient.

Database Projection

The key for efficient calculation of itemset support is the database projection technique [107, 37]. The idea is that, for mining itemsets containing (a superset of) an itemset X , we only need to consider transactions that contains X . For example, the (absolute) support of an itemset Xab in the entire database is the same as the support of an itemset ab , if we only consider transactions containing itemset X . Despite seeming trivial, this idea is very effective.

The benefit of this approach is clear when we generate candidates using the DF or RDF approaches. Whenever we generate candidates containing an item a , we can ignore transactions not containing item a . This projection reduces the size of the database very aggressively, and hence, the set intersection if we use vertical structure, the enumeration of transactions if we use horizontal structure, and also the future projections, can be done very efficiently.

2.3.3 Algorithms for Computing Subset Support

This section discusses the algorithms for calculating all $2^{|X|}$ subset supports of an itemset X .

Naïve Algorithm

To calculate the subset supports of an itemset X , this algorithm iterates each transaction, determines the base B , and increases the subset support based on B , as illustrated in Algorithm 2.8. This algorithm has time complexity $O(n |X|)$ (assuming

Algorithm 2.8: Naïve algorithm for calculating subset support

Input: Itemset X

```

1 for transaction  $t \in T$  do
2    $B \leftarrow I(t) \cap X$ ;
3   increase  $sup_B(X)$ ;

```

Algorithm 2.9: Incremental Algorithm

Input: Itemset X , $sup_B(X)$ for all $B \subseteq X$, $B(t) = I(t) \cap X$ for all $t \in T$,
item a

```

1 for transaction  $t \in T(a)$  do
2   decrease  $sup_{B(t)}$ ;
3    $B(t) \leftarrow B(t) \cup \{a\}$ ;
4   increase  $sup_{B(t)}$ ;

```

we can check the existence of each item $a \in X$ in $I(t)$ in constant time) and space complexity $O(2^{|X|})$.

In practice, despite being similar, the naïve algorithm for calculating the subset support is much less efficient than that for calculating the exact support of an itemset. This is because, for calculating the exact support, we can prune the search directly once we find that the transaction t does not contain an item $a \in X$. Whereas, for calculating the subset support, we need to iterate all items in the itemset. Another reason for the inefficiency is that, for calculating the subset support, we cannot remove any transaction even though it does not contain any item. In other words, we cannot apply the database projection technique.

To implement this algorithm, we can use either the vertical or the horizontal data structure. With the horizontal data structure, we can simply iterate each transaction. With the vertical data structure, we can iterate all transactions containing each item, and update their base. However, as each transaction needs to be iterated/updated, we cannot use the compact structure for the vertical structure.

Incremental Algorithm

In the naïve algorithm, for every itemset, we recalculate the base supported by each transaction. Assuming that the item is inserted (and removed) one by one, we can update the base faster, without the need to recalculate from scratch. This is an extension of the naïve algorithm with the vertical data structure, and is presented in Algorithm 2.9.

In addition to the subset-supports, this algorithm maintains the base supported by each transaction, with respect to the current itemset X . The objective is to allow efficient update of the variables after inserting an item $a \notin X$. This update operation can be done efficiently by iterating the list of transactions containing a , and appending a to the current base of the iterated transaction (Line 3). As the variable $B(t)$, which stores the base supported by transaction t , changes from $I(t) \cap X$ to $I(t) \cap Xa$ (note that transaction t contains item a), the subset-supports are also updated accordingly, by decreasing $\text{sup}_{I(t) \cap X}$ (Line 2) and increasing $\text{sup}_{I(t) \cap Xa}$ (Line 4). The time complexity of this algorithm is $O(|T(a)|) = O(n)$ to maintain the base supported by every transaction and the subset-supports. The space requirement of this algorithm is $O(n + 2^{|X|})$ for storing the bases of each transaction and the subset-supports.

As this algorithm extends the naïve algorithm for using the vertical structure, it is not admissible with the horizontal data structure, as well as the compact form of the vertical data structure.

Naïve Inclusion-Exclusion (NIE) Algorithm

The subset supports of X can be computed from the support of the subsets of X using the inclusion-exclusion principle [22]. This can be done according to the following formula:

$$\text{sup}_B(X) = \sum_{B \subseteq Y \subseteq X} (-1)^{|Y \setminus X|} \text{supp}(Y) \quad (2.1)$$

For example, if $X = abc$, then we have

$$\text{sup}_a(abc) = \text{sup}(a) - \text{sup}(ab) - \text{sup}(ac) + \text{sup}(abc)$$

The NIE algorithm calculates the subset support for each base individually. The total number of iterations to calculate all subset-supports of X can be calculated as: $\sum_{B \subseteq X} 2^{|X \setminus B|} = 3^{|X|}$. Therefore, after calculating the exact support, this algorithm needs additional $O(3^{|X|})$ iterations and $O(2^{|X|})$ space. To measure the overall time, this number needs to be added by the number of iterations to calculate the exact support.

The strength of this algorithm is that, as NIE depends only on the exact supports of subsets, we can apply the database projection technique for efficiently calculating it. Therefore, this algorithm is much more efficient than the algorithms that calculate the subset-supports directly, without using the relation of the supports of subsets.

Algorithm 2.10: QIE: Quick Inclusion-Exclusion Algorithm

Input: Itemset X , $sup(B)$ for all $B \subseteq X$

```

1 for itemset  $B \subseteq X$  do
2    $sup_B(X) \leftarrow sup(B)$ ;
3 for item  $a \in I$  do
4   for itemset  $B \subseteq (X \setminus \{a\})$  do
5      $sup_B \leftarrow sup_B - sup_{Ba}$ ;
```

As the formula is based on the exact support of subsets, the NIE algorithm requires all subsets to be traversed before the current itemset, and their exact support to be stored. With this requirement, this algorithm is not admissible with DF candidate generation algorithms, and is only admissible with the variants that mine all and top- K interesting itemsets with anti-monotone property. This limitation shows the benefits of RDF over DF.

Quick Inclusion-Exclusion (QIE)

Equation (2.1) calculates the subset-support as a function of exact support. By massaging the equation, we can formulate the subset-support as a function of *another subset support* as follows:

$$sup_B(Xa) = sup_B(X) - sup_{Ba}(Xa) \quad (2.2)$$

The Quick Inclusion-Exclusion (QIE) algorithm [21] uses Equation (2.2) in a systematic manner, and is sketched in Algorithm 2.10. In brief, this algorithm initializes the subset supports of X as their exact support, and then keeps extending the itemset-part of subset support by each individual item iteratively, such that at the end, the itemset-part of all subset-supports becomes X .

Phase \ Base	$\emptyset$	a	b	ab	c	ac	bc	abc
Initialization	$\emptyset$	a	b	ab	c	ac	bc	abc
Inserting a	a	a	ab	ab	ac	ac	abc	abc
Inserting b	ab	ab	ab	ab	abc	abc	abc	abc
Inserting c	abc	abc	abc	abc	abc	abc	abc	abc

Table 2.5: Step-by-step illustration of the QIE algorithm.

For example, let the itemset be abc , and the items be iterated lexicographically.

The QIE algorithm is illustrated in Table 2.5, where each cell (phase, base) represent the itemset-part of a subset-support, at corresponding phase for the corresponding base. After the initialization phase, the itemset part equals to the base. After the first iteration when we iterate item a , the itemset-part of subset-supports which do not contain a , is then extended by a using Equation (2.2). Note that there are exactly $2^{|X|-1}$ of them. After iterating all items, the subset-supports calculations are completed.

Analyzing the time complexity, this algorithm performs $|X|$ iterations for iterating each item, and then $2^{|X|-1}$ for iterating the bases. Therefore, the time complexity is given as $O(|X| 2^{|X|-1})$, or $O(|X| 2^{|X|})$, excluding the time taken to calculate the exact support. The space requirement is $2^{|X|}$, to store all subset supports. Comparing NIE with QIE, NIE requires $O(3^{|X|})$ iterations to calculate the subset support of X , while QIE requires $O(|X| 2^{|X|})$ iterations. Therefore, we can expect QIE to be more efficient, especially when $|X|$ is large.

This algorithm has the same characteristic as NIE, i.e., it requires the exact support for all subsets. Which means, as NIE, QIE is only admissible for enumerating all and top- K interesting itemsets with anti-monotone property, but allows very fast support calculation.

2.3.4 Algorithms for Computing Partial Support

We now discuss the algorithms for calculating partial supports of an itemset. As mentioned in Section 2.3.1, knowing one kind of partial supports is sufficient. Therefore, given an itemset X , the objective is to calculate the partial supports, in one of the forms of $sup_{\leq k}(X)$, $sup_{=k}(X)$, and $sup_{\geq k}(X)$, for k between 0 and $|X|$.

Some algorithms are borrowed from those for mining fault-tolerant frequent itemsets [69], since the partial support of itemset X with $C \geq k$ is essentially identical to the *fault-tolerant support* of X with error tolerance $|X| - k$.

Naïve Algorithm

The naïve approach to calculate partial supports is similar to that of subset-supports (Algorithm 2.8). The only difference is that, instead of maintaining the itemsets contained by each transaction, we maintain the *number* of items contained by that transaction. This algorithm results in $O(n |X|)$ time complexity and $O(|X|)$ space complexity.

Algorithm 2.11: Trans-RR algorithm

```

1  $\mathcal{S}_{\geq |X|+1}(Xa) \leftarrow \mathcal{S}(Xa);$ 
2  $sup_{\geq |X|+1}(Xa) \leftarrow |\mathcal{S}_{\geq |X|+1}(Xa)|;$ 
3 for  $k \leftarrow |X|$  down to 1 do
4    $\mathcal{S}_{\geq k}(Xa) \leftarrow \mathcal{S}_{\geq k}(X) \cup (\mathcal{S}_{\geq k-1}(X) \cap T(a));$ 
5    $sup_{\geq k}(Xa) \leftarrow |\mathcal{S}_{\geq k}(Xa)|;$ 

```

This algorithm, accompanied with Apriori candidate generation, has been proposed for mining fault-tolerant frequent itemsets, with the name FT-Apriori [69].

Incremental Algorithm

The incremental algorithm for calculating subset-supports (Algorithm 2.9) can also be modified to calculate the partial supports. In this case, we maintain the number of items contained by each transaction, for every addition/removal of item. The time and space complexity of this algorithm is the same as those for subset-supports, except we only need to store $|X| + 1$ counters instead of $2^{|X|}$.

This algorithm has been proposed for mining fault-tolerant frequent itemsets within the BIAS framework [71].

Recurrence Relation of Supporting Transactions (Trans-RR)

With respect to the current itemset X , each transaction may contain 0 to $|X|$ items. After extending X with an item a , the number of items contained by each transaction t increases by 1 if t contains a , and remains the same otherwise. From this behavior, it is easy to construct a recurrence relation between the partial supporting transactions, i.e., to formulate the partial supporting transactions w.r.t. Xa from those w.r.t. X and the item a .

One of the simplest recurrence relation is proposed in [44] as follows. Let $\mathcal{S}_{\geq k}(X)$ be the set of transactions containing *at least* k items of X , and a be any item not in X , we have:

$$\mathcal{S}_{\geq k}(Xa) = \mathcal{S}_{\geq k}(X) \cup (\mathcal{S}_{\geq k-1}(X) \cap T(a)). \quad (2.3)$$

Equation (2.3) can be executed very efficiently, as it performs only one set union and one set intersection to calculate $\mathcal{S}_{\geq k}(Xa)$. As $\mathcal{S}_{\geq k}$ can be very dense, for example when k is small, [44] proposes to store this set using a bit-vector data structure. The efficiency of this data structure to intersect or union the set is independent to the density of the set. The algorithm is presented in Algorithm 2.11. This algorithm applies

Equation (2.3) for each value of k in decreasing order, to find the partial-supporting transactions of Xa . Having obtained those transactions, the partial support is simply the cardinality of the set. The time complexity to calculate $\text{sup}_{\geq k}(X)$ is $O(n)$ for one particular k , hence this algorithm runs in $O(1/w \ n \ |X|)$, where w is the word-size (either 32 or 64 for 32 and 64 bits processors respectively). Since this algorithm needs to store the partial-supporting transactions for each itemsets' prefix, its space complexity is $O(1/w \ n \ |X|^2)$.

We can also formulate the recurrence relation based on $\mathcal{S}_{=k}$. As $\mathcal{S}_{=k}$ is disjoint for each k , the total size of $\mathcal{S}_{=k}$ for all k is exactly n . The usage of horizontal data structure for this approach is proposed in [90]. As each *level* (prefix) stores n transactions, the time and space complexity becomes $O(n \ |X|)$. Regarding efficiency, this algorithm iterates all transactions whenever we extend the current itemset with an item a . Therefore, this approach is clearly inferior compared to the incremental algorithm, where we only need to iterate transactions containing item a .

Recurrence Relation of Partial Supports (Sup-RR)

The idea of recurrence relation is based on the *inclusion-exclusion principle* for calculating subset support (Equation (2.1)). Repeating the equation, the subset-support $\text{sup}_B(X)$ can be calculated as:

$$\text{sup}_B(X) = \sum_{B \subseteq Y \subseteq X} (-1)^{|Y \setminus B|} \text{sup}(Y).$$

Knowing the subset-supports of an itemset X , its partial support, $\text{sup}_{=k}(X)$, can be calculated as:

$$\text{sup}_{=k}(X) = \sum_{B \subseteq X \wedge |B|=k} \text{sup}_B(X). \quad (2.4)$$

By using only these equations, we can calculate the partial support of an itemset as a function of its subsets' support. Let $Y \subseteq X$ and $|Y| \geq k$, then combining Equations (2.1) and (2.4), we can calculate the coefficient of $\text{sup}(Y)$ in $\text{sup}_{=k}(X)$ as $(-1)^{|Y|-k} \binom{|Y|}{k}$. Let $\text{sum}_{=k}(X) = \sum_{Y \subseteq X, |Y|=k} \text{sup}(Y)$, then Equation (2.4) can be further simplified into:

$$\text{sup}_{=k}(X) = \sum_{i=k}^{|X|} (-1)^{i-k} \binom{i}{k} \text{sum}_{=i}(X).$$

Since $\text{sup}_{\geq k}(X) = \sum_{i=k}^{|X|} \text{sup}_{=i}(X)$, the coefficient of $\text{sup}(Y)$ in $\text{sup}_{\geq k}(X)$ can

2.3. INTERESTINGNESS COMPUTATION

43

be calculated as $\sum_{i=k}^{|Y|} (-1)^{|Y|-i} \binom{|Y|}{i} = (-1)^{|Y|-k} \binom{|Y|-1}{k-1}$, by the identity of binomial coefficient. Hence, a similar function can also be derived for $sup_{\geq k}(X)$, which is:

$$sup_{\geq k}(X) = \sum_{i=k}^{|X|} (-1)^{i-k} \binom{i-1}{k-1} sum_{\geq i}(X).$$

Note that the coefficient of $sup(Y)$ in $sup_{=k}(X)$ is independent of X . For example, the coefficient of $sup(ab)$ in $sup_{=1}(abc)$ is equal to the coefficient of $sup(ab)$ in $sup_{=1}(abcd)$. Having analyzed the formula, we discover a recurrence relation between partial supports, which is presented in following equation.

$$sup_{\geq k}(X) = \begin{cases} sup(X) & \text{if } k = |X| \\ \frac{(\sum_{a \in X} sup_{\geq k}(X \setminus \{a\})) - k \cdot sup_{\geq k+1}(X)}{|X| - k} & \text{otherwise} \end{cases} \quad (2.5)$$

Proof. We prove the recurrence relation using induction. The base case when $|X| = 1$, and $k = |X|$ is trivial. Next, we prove the induction phase.

First, we analyze itemsets subsumed by $sup_{\geq k}(X \setminus \{a\})$. Recalling the definition, itemsets included in this function are those containing at least k items of $X \setminus \{a\}$. Hence, considering item a , a transaction can either (1) contain *at least* $k+1$ items of X , or (2) not contain a and contains *exactly* k items of X . These 2 parts are *disjoint* and *complete*. Note that (1) is equivalent to $sup_{\geq k+1}(X)$.

Since we sum those terms for all items $a \in X$, at the end we have (3) $|X|$ multiplies of $sup_{\geq k+1}(X)$, and (4) summation of transactions containing exactly k items, each not containing one item of X . Now we count how many times each k -subset of X appears in (4). Each k -subset of X contains k items, and does not contain $|X| - k$ items. Hence, it appears exactly $|X| - k$ times in (4). Equating the numerators of equation 2.5, we have $|X| - k$ multiple of itemsets containing exactly k items, and also $|X| - k$ multiple of itemsets containing at least $k+1$ items, hence together we have $|X| - k$ multiple of itemsets containing at least k items, which is $(|X| - k) sup_{\geq k}(X)$.

To attain $sup_{\geq k}(X)$, we simply divide the result by $|X| - k$. $\square$

Based on this recurrence relation, the partial support of an itemset X can be calculated by knowing the partial support of each $(|X| - 1)$ -subset of X , and the actual support of X . Note that the nature of the recurrence relation presented here is different from that of Trans-RR. While the domain in Trans-RR is the *supporting-tidsets*, here the domain is the *partial supports* themselves. This difference affects the efficiency significantly. While Trans-RR reiterates each transaction, taking $O(n)$

Algorithm 2.12: Sup-RR algorithm

Input: Itemset X , $\text{sup}(X)$, $\text{sup}_{\geq k}(X \setminus \{a\})$ for $0 \leq k \leq |X| - 1$ and $a \in X$

- 1 $\text{sup}_{\geq |X|}(X) \leftarrow \text{sup}(X)$;
- 2 **for** $k \leftarrow |X| - 1$ **down to** 1 **do**
- 3 $\text{sup}_{\geq k}(X) = \frac{(\sum_{a \in X} \text{sup}_{\geq k}(X \setminus \{a\})) - k \text{sup}_{\geq k+1}(X)}{|X| - k}$;

iterations, Sup-RR only performs arithmetic manipulation, taking $O(|X|)$ time.

The algorithm for calculating partial support can be easily constructed from Equation (2.5), as presented in Algorithm 2.12. Analyzing the complexity, we iterate each direct subset for every k , hence the time complexity is $O(|X|^2)$. As this algorithm requires the partial support of subsets, we need to store the partial supports of all interesting itemsets, which is in the order of $O(K |X|)$.

This algorithm works in a similar fashion with NIE and QIE for computing subset supports. That is, this algorithm is only admissible for enumerating all and top- K interesting itemsets with anti-monotone property, but admits very fast support calculation.

Using Subset Support to derive Partial Support

Partial support can be formulated as an aggregate information of subset-supports. Therefore, we can calculate partial supports by post-processing subset supports as in Equation 2.4, by iterating all subset supports.

Analyzing all algorithms for calculating subset supports, the only possible improvement is by post-processing the QIE algorithm. This technique has worse time-complexity than Sup-RR, as it takes $O(|X| 2^{|X|})$ time as compared with $O(|X|^2)$ for the latter. However, on the other hand, it requires less space as it only needs to store the support of each interesting itemset, whilst Sup-RR needs to store the partial supports. Therefore, the space complexity of QIE is $O(K)$ whereas Sup-RR is $O(K |X|)$.

2.4 Summary of Algorithms

A comparative summary of all algorithms for calculating subset-supports and partial supports is presented in Table 2.6. Variable t_{ex} refers to the number of iterations to calculate the exact support of each itemset. The second column “*Require*” refers to

Algorithm	Require	Subset Support		Partial Support	
		Time	Space	Time	Space
Naïve	–	$n X $	$2^{ X }$	$n X $	$ X $
Incremental	Prefix	n	$n + 2^{ X }$	n	$n + X $
NIE	Subsets	$t_{ex} + 3^{ X }$	$K + 2^{ X }$	$t_{ex} + 3^{ X }$	$K + 2^{ X }$
QIE	Subsets	$t_{ex} + X 2^{ X }$	$K + 2^{ X }$	$t_{ex} + X 2^{ X }$	$K + 2^{ X }$
Trans-RR	Prefix	–	–	$1/w n X $	$1/w n X ^2$
Sup-RR	Subsets	–	–	$t_{ex} + X ^2$	$K X $

Table 2.6: Information required by algorithms for calculating detailed supports and their time and space complexities.

	Anti-Monotone	Weak Anti-Monotone
All, Apriori	Subset	–
All, DF	Prefix	Prefix
All, RDF	Prefix and Subset	–
Top-K	Prefix and Subset	Prefix
Maximal	Prefix	Prefix

Table 2.7: Information provided by candidate generation algorithms.

the required information in order to use the corresponding algorithm. For example, to use the incremental algorithm for calculating subset-support (require *prefix*) of an itemset Xa , we need to know the *base* contained by each transactions with respect to itemset X . The information provided by each candidate generation algorithm, with measures satisfying anti-monotone or weak anti-monotone property is presented in Table 2.7. These two tables should contain the required information to derive which candidate generation algorithms are admissible with which interestingness computation algorithms.

From the time complexities presented in Table 2.6, the most promising algorithm for calculating partial supports is Sup-RR if all subsets are known, and Incremental or Trans-RR when the prefixes are known. For calculating subset-support, QIE is the most efficient when all subsets are known, and Incremental is the most efficient when only the prefixes are known. The empirical analysis of these algorithms are presented in Section 2.6.

2.5 Applications

In previous sections, we assume the existence of an interestingness function $\mu(X)$ which can be formulated as detailed supports. Here, we demonstrate the applicability

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
1				
2				
3				
4				

Figure 2.2: Sample transactional database as a running example (repeated).

of our framework by formulating several well-known interestingness functions, such as: entropy, density, and bond, in terms of their required detailed supports (partial supports or subset supports) and their monotonicity properties.

These formulations are explained with the help of a running example, using the sample transactional database of Figure 1.1, repeated here in Figure 2.2 for convenience.

2.5.1 Entropy

The entropy of an itemset X , denoted as $H(X)$, measures the *complexity*, or the *predictability* of an itemset X . The lower the entropy, the more predictable the distribution is. Mathematically, the entropy of itemset X is formulated as:

$$H(X) = - \sum_{Y \subseteq X} \left(\frac{\text{sup}_Y(X)}{n} \right) \log \left(\frac{\text{sup}_Y(X)}{n} \right)$$

which is a function of subset-supports of X .

Example 2.9. Using our running example, we can calculate

$$\begin{aligned}
H(ab) &= - \sum_{Y \subseteq ab} \left(\frac{\text{sup}_Y(ab)}{4} \right) \log \left(\frac{\text{sup}_Y(ab)}{4} \right) \\
&= - \left(\frac{\text{sup}_\emptyset(ab)}{4} \right) \log \left(\frac{\text{sup}_\emptyset(ab)}{4} \right) - \left(\frac{\text{sup}_a(ab)}{4} \right) \log \left(\frac{\text{sup}_a(ab)}{4} \right) \\
&\quad - \left(\frac{\text{sup}_b(ab)}{4} \right) \log \left(\frac{\text{sup}_b(ab)}{4} \right) - \left(\frac{\text{sup}_{ab}(ab)}{4} \right) \log \left(\frac{\text{sup}_{ab}(ab)}{4} \right) \\
&= -\frac{1}{4} \lg \left| \frac{1}{4} \right| - \frac{1}{4} \lg \left| \frac{1}{4} \right| - \frac{1}{4} \lg \left| \frac{1}{4} \right| - \frac{1}{4} \lg \left| \frac{1}{4} \right|
\end{aligned}$$

2.5. APPLICATIONS

47

It is proven in [39] that entropy satisfies the monotone property. Therefore, if we want to mine *low entropy itemsets*, we can set $\mu(X) = -H(X)$ which is anti-monotone. Similarly, if we want to mine *large entropy itemsets*, we can set $\mu(X) = H(X)$ which is monotone.

Because of the monotone property, the entropy measure is biased toward large itemsets. This can be normalized into *scaled entropy* formulated as:

$$H_s(X) = H(X)/|X|$$

and *normalized entropy* formulated as:

$$H_n(X) = H(X) / \sum_{a \in X} H(a)$$

The purpose of dividing by the number of attributes in $H_s(X)$ is simply to punish larger itemsets. If all items are totally independent, then $H_s(X)$ will return the average entropy of individual items. For the normalized entropy, the normalization factor represents the maximum entropy that can be achieved with the given items, when all items are independent. Consequently, the value of $H_n(X)$ is guaranteed to be between zero and one. Both $H_s(X)$ and $H_n(X)$ are weak anti-monotone [39].

Another entropy-related measure is the mutual information. Having a 2-itemset $X = ab$, the mutual information $MI(X)$ is calculated as:

$$MI(ab) = H(a) + H(b) - H(ab).$$

$MI(X)$ measures the correlation between the two variables. If the two variables are totally independent, then $MI(X) = 0$. On the other hand, if one variable can be perfectly derived from the other, we have $MI(X) = \max(H(a), H(b))$. Therefore, to avoid bias towards variables with large entropy, this measure can be normalized into *normalized mutual information* (NMI) [42], as:

$$NMI(ab) = MI(ab) / \max(H(a), H(b)).$$

To generalize this measure into itemsets with more than two items, [42] proposed to calculate the minimum mutual information for every two variables, given as:

$$NMI(X) = \min_{a,b \in X} NMI(ab).$$

Because of the minimum function, $NMI(X)$ is anti-monotone.

As $H_s(X)$, $H_n(X)$, $MI(X)$, and $NMI(Z)$ are all functions of the entropy of X , these functions can be calculated using the subset-supports of X .

2.5.2 Density

The *density* of an itemset X [85], with respect to a transaction-set Y , measures the proportion of cells with value ‘1’ in the matrix $Y \times X$, which can be calculated as

$$dens_Y(X) = \frac{\sum_{t \in Y} |I(t) \cap X|}{|Y| \times |X|}. \quad (2.6)$$

Example 2.10. Using the database in our running example (Figure 2.2), we have $dens_{\{3,4\}}(ab) = (1 + 1)/(2 \times 2) = 2/4$, and $dens_{\{1,2,4\}}(ac) = (1 + 1 + 2)/(3 \times 2) = 4/6$.

The density of an itemset X with respect to a support s is defined as the maximum density $dens_Y(X)$ for all transaction-sets of size s , which can be calculated as:

$$dens_s(X) = \max_{Y \subseteq T \wedge |Y|=s} dens_Y(X).$$

Example 2.11. From our running example, we have $dens_2(ab) = 3/4$ from transaction-set $\{1, 3\}$ or $\{1, 4\}$. Similarly, $dens_2(cd) = 4/4$ from transaction-set $\{2, 4\}$.

Given a density threshold σ and the support s , an itemset X is considered dense if $dens_s(X) \geq \sigma$.

As the numerator in Eq. (2.6) is a simple summation of transactions, the transaction-set Y that produces the maximum density can be obtained by taking s transactions t with the largest $|I(t) \cap X|$.

Example 2.12. From our running example, the number of items in abc contained by transactions 1 to 4 is: 2, 1, 1, and 2 respectively. Therefore, $dens_2(abc) = (2 + 2)/6$ and $dens_3(abc) = (2 + 2 + 1)/9$.

Instead of iterating the transactions, we can calculate $dens_s(X)$ by iterating the partial supports of X . That is, for each value of k (from $|X|$ to 0), we calculate the number of transactions containing k items which need to be taken (belong to the top- s transactions). This algorithm is illustrated in Algorithm 2.13. In short, variable b maintains the number of transactions which are already taken, and variable a maintains the total number of items (cells) which are contained in those transactions. As

Algorithm 2.13: Calculating the density of itemset X .

Input: Itemset X , support value s **Output:** $dens_s(X)$

```

1  $a \leftarrow 0$ ;
2  $b \leftarrow 0$ ;
3 for  $k \leftarrow |X|$  down to 0 do
4   if  $b + sup_{=k}(X) \geq s$  then
5      $a \leftarrow a + (s - b) \times k$ ;
6      $b \leftarrow s$ ;
7     break;
8   else
9      $a \leftarrow a + sup_{=k}(X) \times k$ ;
10     $b \leftarrow b + sup_{=k}(X)$ ;
11 return  $a / (s \times |X|)$ 

```

this algorithm performs one iteration of the partial supports of X , its time complexity is $O(|X|)$.

Example 2.13. Continuing the previous example, we have $sup_{=3}(abc) = 0$, $sup_{=2}(abc) = 2$, and $sup_{=1}(abc) = 2$. When $s = 2$, we take 0 transaction from $\mathcal{S}_{=3}(abc)$ and 2 transactions from $\mathcal{S}_{=2}(abc)$, and hence the numerator equals to 4. When $s = 3$, we take 0 transaction from $\mathcal{S}_{=3}(abc)$, 2 transactions from $\mathcal{S}_{=2}(abc)$, and 1 from $\mathcal{S}_{=1}(abc)$, and hence the numerator equals to 5.

The density of an itemset is weak anti-monotone, but not anti-monotone, and hence is difficult to mine [84]. The proof of this property can be found in [85]. Instead of density, the proposing paper [85] uses *strong density*, denoted as $sdens_s(X)$, which requires all subsets of an itemset to also be dense. Mathematically, strong-density is defined as:

$$sdens_s(X) = \min_{Y \subseteq X} dens_s(Y)$$

This additional constraint is used to filter-out itemsets which are considered dense only because of few subsets, i.e., it is possible that an itemset Xa has a high density even though only very few transactions contain item a . Because of this constraint, the strong-density of an itemset is clearly anti-monotone.

2.5.3 Bond

Another interestingness measure is called *bond* [59] which is defined as the ratio between the number of transactions containing *all items*, and those that contain *any item*. This measure is analogous to Jaccard similarity. We can formulate bond with partial supports, as the numerator is $\text{sup}(X)$, and the denominator is $\text{sup}_{\geq 1}(X)$, such as:

$$\text{bond}(X) = \frac{\text{sup}(X)}{\text{sup}_{\geq 1}(X)}$$

The bond of itemsets satisfies the anti-monotone property.

Example 2.14. From our running example, we can calculate $\text{bond}(ab) = \frac{\text{sup}(ab)}{\text{sup}_{\geq 1}(ab)} = 1/3$.

2.5.4 Derivability

The support of an itemset can be bounded by those of its subsets. For example, because of the anti-monotone property, we know that $\text{sup}(X) \leq \text{sup}(X \setminus a)$. This bound is generalized by using inclusion-exclusion principle in [22] as follows. Recalling the Equation (2.1),

$$\text{sup}_Y(X) = \sum_{Z|Y \subseteq Z \subseteq X} (-1)^{|X \setminus Z|} \text{sup}(Z).$$

Since $\text{sup}_Y(X)$ is non-negative, bringing both $\text{sup}(X)$ and $\text{sup}_Y(X)$ to the LHS of the above equation, we have:

$$\text{sup}(X) \leq \text{sup}(X) + \text{sup}_Y(X) = \text{ub}_Y(X) = - \sum_{Z|Y \subseteq Z \subset X} \text{sup}(Z)(-1)^{|X \setminus Z|}$$

when $|X \setminus Y|$ odd, and

$$\text{sup}(X) \geq \text{sup}(X) - \text{sup}_Y(X) = \text{lb}_Y(X) = - \sum_{Z|Y \subseteq Z \subset X} \text{sup}(Z)(-1)^{|X \setminus Z|}$$

otherwise.

Example 2.15. Let $X = abc$, we have $\text{sup}(abc) + \text{sup}(\bar{a}bc) = \text{sup}(bc)$. Here, $\text{sup}(bc)$ never underestimates $\text{sup}(abc)$. Similarly, we have $\text{sup}(abc) - \text{sup}(\bar{a}\bar{b}c) = \text{sup}(ac) + \text{sup}(bc) - \text{sup}(c)$, which means, $\text{sup}(abc)$ is lower bounded by $\text{sup}(ac) + \text{sup}(bc) - \text{sup}(c)$.

2.5. APPLICATIONS

51

Example 2.16. From our running example, let $X = ab$, we have $ub_a(X) = 2$, $ub_b(X) = 2$, and $lb_\emptyset(X) = 2 + 2 - 4 = 0$. Hence, $lb(X) = 0 < sup(X) = 1 < ub(X) = 2$.

Knowing all subsets of X , we can define the upper bound of $sup(X)$, $ub(X)$, as the tightest upper bound with respect to each subset. That is, $ub(X)$ is formulated as $ub(X) = \min_{Y \subset X \wedge |X \setminus Y| \text{ odd}} ub_Y(X)$. We can also define the lower bound of $sup(X)$, $lb(X)$, in the similar way. As $sup(X)$ is always between $lb(X)$ and $ub(X)$, an itemset is considered derivable if it has low $ub(X) - lb(X)$. Let the *non-derivability* of $sup(X)$ be $ub(X) - lb(X)$, it is proven in [22] that non-derivability of itemsets is anti-monotone. That is, the non-derivability of itemset $Y \supset X$ is never greater than that of X . In [22], the notion of *derivable itemset* relates to itemsets whose support can be perfectly derivable, i.e., $lb(X) = ub(X)$, and analysts are interested in itemsets which are not perfectly derivable, i.e., $lb(X) < ub(X)$.

Having the subset supports of X , the bounds $ub(X)$ and $lb(X)$ can be calculated by simply iterating the subset supports. Note that since $ub(X)$ and $lb(X)$ are both in the form of $sup(X) \pm sup_Y(X)$, we can ignore $sup(X)$ while calculating $ub(X) - lb(X)$. That is, we only need to check the smallest $sup_Y(X)$ for odd and even $|X \setminus Y|$.

Example 2.17. Continuing the previous example with $X = ab$, we have $sup_{a\bar{b}}(X) = 1$, $sup_{\bar{a}b}(X) = 1$, and $sup_{a\bar{b}}(X) = 1$. Hence, $ub(X) = sup(X) + 1$ and $lb(X) = sup(X) - 1$, leads to the non-derivability of X as $1 - (-1) = 2$.

2.5.5 Expected Support in Bernoulli Noise

In many real world situations, we are faced with noisy databases, where the noise might be deliberately added to preserve privacy. The simplest way to simulate the noise is by assuming that the noise affects each cell in the whole database independently, which is also called *Bernoulli noise*. In a binary matrix (transactional database), Bernoulli noise is usually defined with two parameters, p which represents the probability that a '1' is correctly represented as '1', and q which represents the probability that a '0' is correctly represented as '0'. For example, the *pure* database has $p = q = 1$. Having a noisy database, the estimated support of an itemset (in the true database) can be calculated as:

$$\hat{sup}(X) = \sum_{k=0}^{|X|} sup_{=k}(X) \left(\frac{q}{p+q-1} \right)^k \left(\frac{q-1}{p+q-1} \right)^{|X|-k} \quad (2.7)$$

Proof. The expected support of an itemset after the database is affected with Bernoulli noise can be calculated using matrix operations. Let $sup'(x)$ be the expected support of item x after Bernoulli noise, and $sup(\bar{x})$ be the support of the negation of x (number of transactions not containing x), then the expected support can be calculated as:

$$\begin{pmatrix} sup'(x) \\ sup'(\bar{x}) \end{pmatrix} = M \begin{pmatrix} sup(x) \\ sup(\bar{x}) \end{pmatrix}$$

where

$$M = \begin{pmatrix} p & 1 - q \\ 1 - p & q \end{pmatrix}$$

To estimate the original support given the noisy database, we can inverse the matrix M , such as:

$$\begin{pmatrix} sup(x) \\ sup(\bar{x}) \end{pmatrix} = \frac{1}{p + q - 1} \begin{pmatrix} q & q - 1 \\ p - 1 & p \end{pmatrix} \begin{pmatrix} sup'(x) \\ sup'(\bar{x}) \end{pmatrix}$$

Since the noise is independently distributed, the inverse is also independent. Hence, the expected support of an itemset is simply the product of individual item's expected support. Given the noisy database, the estimated true support of an itemset, before the database was affected by Bernoulli noise can be calculated as

$$\sum_{k=0}^{|X|} sup_{=k}(X) \left(\frac{q}{p + q - 1} \right)^k \left(\frac{q - 1}{p + q - 1} \right)^{|X| - k}$$

□

Example 2.18. Using our running example (assuming the observed database is the noisy database), let $X = ab$, $u = \left(\frac{q}{p + q - 1} \right)$ and $v = \left(\frac{q - 1}{p + q - 1} \right)$. We can calculate the estimated $\hat{sup}(ab) = sup_{=0}(ab) u^2 + sup_{=1}(ab) u v + sup_{=2}(ab) v^2$.

In some cases, each item a has its own noise level p_a and q_a . For example, measuring length might be more accurate than measuring temperature. Here, the interestingness is measured as the estimated true support, given a noisy database, and the noise level per item. In this case, the estimated support of an itemset can be calculated as:

$$\hat{sup}(X) = \sum_{Y \subseteq X} sup_Y(X) \prod_{a \in Y} \frac{q_a}{p_a + q_a - 1} \prod_{a \in X \setminus Y} \frac{q_a - 1}{p_a + q_a - 1}.$$

Interestingness Measures	Monotonicity property	Detailed supports
Low entropy	Anti-monotone	Subset support
Large entropy	Monotone	Subset support
Scaled entropy	Weak anti-monotone	Subset support
Normalized entropy	Weak anti-monotone	Subset support
Normalized mutual information	Anti-monotone	Subset support
Density	Weak anti-monotone	Partial support
Strong density	Anti-monotone	Partial support
Bond	Anti-monotone	Partial support
Derivability	Anti-monotone	Subset support
Exp. Bernoulli uniform	Not anti-monotone	Partial support
Exp. Bernoulli non-uniform	Not anti-monotone	Subset support

Table 2.8: Interestingness measures along with their monotonicity properties and required detailed supports.

Example 2.19. Continuing the previous example, let $u_a = \frac{q_a}{p_a + q_a - 1}$ and $v_a = \frac{q_a - 1}{p_a + q_a - 1}$ for item a . We can calculate the estimated $\hat{sup}(ab) = sup_{\hat{a}\hat{b}}(ab) u_a u_b + sup_{a\hat{b}}(ab) v_a u_b + sup_{\hat{a}b}(ab) u_a v_b + sup_{ab}(ab) v_a v_b$.

Unfortunately, expected support is neither anti-monotone nor weak anti-monotone. The usual way to mine itemsets with large expected support is by assuming the existence of anti-monotone property, which is true if the observed database truly reflects the noisy database, i.e., the support of an itemset X in the observed database is the same as its expected support after the original database is contaminated by the noise. This condition can normally be assumed if the number of transactions is sufficiently large.

2.5.6 Summary of Interestingness Measures

The interestingness measures discussed in this Section are summarized in Table 2.8. We do not empirically evaluate all these measures because some of them are redundant, and the lack of competitors. Therefore, we only evaluate the algorithms where the competitor exists.

2.6 Experimental Results

We used three real datasets, viz., BMS-Webview-1, Mushroom, and Chess for evaluating the efficiency of the algorithms, as the representatives of large and sparse

Figure 2.3: Enumerating frequent itemsets with partial supports.

transactional database (BMS-Webview-1) and dense categorical databases (Mushroom and Chess). These datasets are chosen in order to demonstrate the scalability of the algorithms with respect to density, and the practicality in real applications. Details of these datasets are available in Appendix A. The minimum supports provided in the charts are stated as the *relative support*, which is the proportion of the absolute support and the number of transactions.

2.6.1 Computing Partial Supports and Subset Supports

Firstly, we evaluate the efficiency of the algorithms for computing PS and SS for various enumeration problems, viz., enumerating all interesting itemsets, top- K interesting itemsets, and maximal interesting itemsets for functions satisfying anti-monotone and weak anti-monotone property. To avoid the bias in the interestingness computation, we use support as the interestingness measure. For enumerating the function with weak anti-monotone property, we simply ignore the fact that support is anti-monotone, and use the fact that it is weak anti-monotone.

All Itemsets, Anti-Monotone

This section evaluates the efficiency of algorithms for mining all interesting itemsets, where the interestingness measure satisfies the anti-monotone property. First, we compare the algorithms for calculating partial-supports of frequent itemsets, and the results are depicted in Figure 2.3. We compare 4 algorithms, namely Naïve algorithm + Breadth-First (BF) enumeration, Incremental algorithm + DF, Trans-RR + DF, and Sup-RR + RDF.

From the plots, it is obvious that Sup-RR + RDF is much more efficient than the rest. Sup-RR is efficient because it is equipped with the database projection technique (hence allows very fast exact support calculation), and also very efficient calculation of

2.6. EXPERIMENTAL RESULTS

55

Figure 2.4: Enumerating frequent itemsets with subset supports.

Figure 2.5: Efficiency of calculating exact support, partial support, and subset support.

partial supports, i.e., $O(|X|^2)$. Among the memory efficient algorithms (the one with DF enumeration), Incremental algorithm is more efficient in sparse dataset (BMS-Webview-1), and Trans-RR is more efficient in dense datasets. Nevertheless, both are one to two order of magnitudes slower than Sup-RR. It is interesting that the Naïve algorithm, which is the most common algorithm, is always the slowest.

Next, we compare the efficiency of algorithms for calculating the subset-supports of frequent itemsets, and the results are depicted in Figure 2.4. We compare 3 algorithms, namely Naïve+BF, Incremental+DF, and QIE+RDF.

From the plots, we can observe that QIE is in general the most efficient, except in the very dense dataset (Chess). This is because QIE allows very fast exact support calculation. Meanwhile, the subset-support calculation takes $O(|X| 2^{|X|})$ iterations, which is very efficient for small itemsets, but becomes slow for large itemsets, which are common in dense datasets.

Next, we measure the cost of calculating the partial supports and the subset supports, by comparing them with the one that calculates only the itemset exact support. The results for this experiment are depicted in Figure 2.5. We use Sup-RR and QIE algorithms as the representation of algorithms for calculating PS and SS

Figure 2.6: Enumerating maximal frequent itemsets.

respectively, as they are in general the most efficient. We see that calculating PS is slightly less efficient than calculating support, and calculating SS is less efficient than calculating PS, where the differences are more obvious in denser datasets. This behavior is as expected, since calculating PS needs additional $O(|X|^2)$ iterations, and calculating SS needs additional $O(|X| \cdot 2^{|X|})$ iterations. In sparse (and large) dataset, the difference is negligible since such additional costs are insignificant compared to that for calculating the exact support of itemsets. This experiments also show that partial supports can be calculated faster than the subset supports.

Maximal Itemsets, Anti-Monotone

This section evaluates the efficiency of the algorithms for enumerating maximal interesting itemsets. Here, we use depth-first enumeration, and prune non-maximal itemsets as in GENMAX [31]. The details of the algorithms are presented in Section 7. For calculating the partial supports and subset supports, we use Trans-RR and Incremental algorithms for the former, and Incremental algorithm for the latter. The results are plotted in Figure 2.6.

We can see a similar trend as in the case for mining all interesting itemsets, where the incremental algorithm has similar efficiency for enumerating PS and SS, and Trans-RR is faster than the incremental algorithm for denser datasets.

Top- K Itemsets, Anti-Monotone

Instead of setting the minimum interestingness threshold, analysts might also desire to set the number of enumerated interesting itemsets, which we call the top- K interesting itemsets. In this experiment, we evaluate only the fastest algorithms for calculating each detailed support, i.e., the partial support and the subset support. When the measure satisfies the anti-monotone property, the obvious winner for the former is

2.6. EXPERIMENTAL RESULTS

57

Figure 2.7: Enumerating top- K frequent itemsets.

Figure 2.8: Enumerating frequent itemsets assuming weak anti-monotone property.

the Sup-RR algorithm and the latter is the QIE algorithm, as has been evaluated for enumerating all interesting itemsets. Therefore, we compare the two and present the results in Figure 2.7. We can observe that the result is similar with previous experiments, where Sup-RR is much faster than QIE in the *tougher* settings (higher K). One interesting thing is that the time taken by each algorithm is almost similar across datasets.

Compared to those enumerating all interesting itemsets, the algorithms for the top- K enumeration are slightly less efficient for finding the same number of itemsets. However, in practice, enumerating top- K itemsets are much more convenient, as the analysts would require lesser trial and error for finding the appropriate threshold. As depicted in the plots, the algorithms require only hundreds (and tens) of seconds for finding one million (one hundred thousand) itemsets, which should suffice most practical needs.

All Itemsets, Weak Anti-Monotone

This section evaluates the efficiency of algorithms for mining all interesting itemsets, where the interestingness measure satisfies the weak anti-monotone property. We

Figure 2.9: Enumerating maximal frequent itemsets assuming weak anti-monotone property.

evaluate the w-Apriori and w-DF algorithms for calculating partial supports (PS) and subset supports (SS). We evaluate two variants for calculating the partial support with w-DF candidate generation, which are Trans-RR (denoted as bit for brevity) and incremental (denoted as Inc). For calculating subset support (with w-DF candidate generation), the only viable algorithm is the incremental algorithm. The results are plotted in Figure 2.8.

We can observe that all three plots follow the same trends, w-DF is always much faster than w-Apriori. This is interesting as w-Apriori is far more common in the current literature. We can also observe that for the same algorithms, calculating subset supports took almost identical time as calculating partial supports. The possible explanation is that the time taken for these algorithms is dominated by their exact support counting module, and hence the calculation of partial and subset supports are insignificant. Comparing the Trans-RR (bit) and the incremental algorithm, as the case with anti-monotone property, the earlier is more efficient in dense datasets, while the former is more efficient in sparse datasets. However, the *datasets* become *sparser* with weak anti-monotone property as we cannot ignore infrequent items. Therefore, unlike the case with anti-monotone property, here the incremental algorithm is more favorable.

Maximal Itemsets, Weak Anti-Monotone

Next, we evaluate the algorithms for enumerating maximal interesting itemsets, when the measure satisfies the weak anti-monotone property. We compare the same three algorithms, as in the case with anti-monotone property. The results are depicted in Figure 2.9. The results are consistent with those for mining all interesting itemsets.

2.6. EXPERIMENTAL RESULTS

59

Figure 2.10: Enumerating top- K frequent itemsets assuming weak anti-monotone property.

Top- K Itemsets, Weak Anti-Monotone

We did the same experiment as before for the weak anti-monotone measure. The results of our experiment are depicted in Figure 2.10. We can see that the scalability is consistent with previous experiments.

2.6.2 Applications Requiring Partial Supports

This section presents the empirical results of using partial supports for deriving several interestingness measures of itemsets. The details of those measures are presented in Section 2.5.

Dense Itemset

The targets of mining in this experiment are *Dense Itemsets* [85]. In order to avoid bias in implementations, we implemented the algorithm ourselves (which is faster than the original implementation) using the same modules as others. The results for this experiment is depicted in Figures 2.11. In Figures (a) to (c), we vary minimum support while fixing the δ , and in Figures (d) to (f), we vary δ while fixing minimum support. The parameter δ is chosen in a way such that the difference is noticeable.

Our algorithms are mentioned as Sup-RR, and the original algorithm is mentioned as DENSE-SETS. DENSE-SETS is very similar to Apriori-Naïve algorithm, hence the result is also similar to that of the previous experiment. Therefore, we conclude our method of calculating partial-supports can boost the efficiency for enumerating dense itemsets.

Figure 2.11: Enumerating dense itemsets. Figures (a) to (c) plot the scalability of each algorithm with respect to s , when σ is fixed. Figures (d) to (e) plot the scalability when σ varies.

FIM with Bernoulli Noise

Here, the interestingness is measured as the estimated true support, given a noisy database [81]. For the competitor, we use the optimized algorithm described in [7], which is denoted as EMASK. The EMASK algorithm finds the count of each subset support in a naïve way (doing the inclusion-exclusion per subset) as in NIE algorithm, yielding an algorithm with time complexity $O(3^{|X|})$. As Sup-RR takes $O(|X|^2)$ iterations, we expect EMASK to be slower than Sup-RR, especially for large itemsets.

The results for this experiment are depicted in Figures 2.12. In Figure (a) to (c), we vary minimum support while fixing p and q . In Figures (d) to (f), we vary the noise parameters p and q , while fixing minimum support. Note that we do not actually contaminate the dataset with noise, but simply assume that they are noisy. Hence, the result is deterministic. In any case, the assumption does not matter since we only measure the efficiency, and not the accuracy.

For simplicity, we set p equals to q , hence, we only have one parameter to vary. As in the previous experiment, the value of p and q are set in a way such that the difference is noticeable. The impact of p and q varies between datasets. For example, *BMS-Webview-1* is very sensitive to p and q . When these parameters are set to be as

2.6. EXPERIMENTAL RESULTS

61

Figure 2.12: Enumerating frequent itemsets in presence of Bernoulli noise. Figures (a) to (c) plot the scalability of each algorithm with respect to σ , when p and q are fixed. Figures (d) to (e) plot the scalability when p and q vary.

low as 99%, there are no frequent itemsets remaining. *Mushroom* dataset is not very sensitive to these parameters. In *Chess* dataset, the effect is even reversed, i.e., lower p and q results in more itemsets.

The outcomes are as expected, that EMASK is slower than Sup-RR, especially for lower minimum supports (those with more and longer results). As before, we conclude that we have improved the algorithm for mining frequent itemsets when the database is contaminated with Bernoulli noise.

Large Bond Itemsets

Here we mine itemsets with large *bond* [59]. As mentioned previously, the bond of an itemset is calculated as the proportion of the number of transactions containing *all* items in X and the number of transactions containing *any* item in X . This formula can be easily derived from the partial support of itemsets. However, the algorithm proposed in the original paper is not efficient because of lack of optimization (database projection) and multiple redundant operations (the proposed algorithm repeatedly performs intersection and union of every item in the candidate itemsets).

Figure 2.13: Enumerating large bond itemsets.

For a fair comparison, we implement the competing algorithms using the same source-code (hence the same pruning methods and optimization techniques), except we do not store the partial supports. For calculating the union (denominator), we store the *union-list* in addition to the intersection list (the usual list) which can be maintained by performing a union of the lists. Hence, the trade-off is between storing and calculating partial supports, versus maintaining the *union-list*.

The results are presented in Figure 2.13, with our method mentioned as Sup-RR and the one storing both intersection and union list mentioned as INT-UN. The plots show that Sup-RR is more efficient than INT-UN. It is interesting that calculating partial supports is faster than maintaining the union-list, even though the latter is implemented in a quite optimized way.

2.6.3 Applications Requiring Subset Supports

This section presents the empirical results of using subset supports for deriving several interestingness measures of itemsets.

Low-Entropy Sets

The first measure requiring subset supports (SS) is the entropy of itemsets. Here, we are interested with low-entropy itemsets, or itemsets with entropy lesser than a given threshold [39]. An itemset has a low-entropy if the distribution of its subsets is biased, i.e., some subsets have a very high probability, and some others have a very low probability. With this definition, itemset (or items) with support 0 may have a very low entropy, which is meaningless. Hence, in addition to a maximum entropy threshold, we impose a minimum support requirement as in the usual frequent itemset mining problem.

2.6. EXPERIMENTAL RESULTS

63

Figure 2.14: Enumerating low entropy itemsets.

Figure 2.15: Enumerating non-derivable itemsets. Notice the absolute minimum support and linear scale in both Mushroom and Chess datasets.

The algorithm described in [39] scans the database while calculating the entropy of each itemset. Hence, it is essentially identical to the Naïve algorithm. We compare it with our QIE variants, and the results are presented in Figure 2.14. It is not surprising that the scalability is similar to our previous experiment, that QIE is always more efficient than Naïve.

Non-Derivable Itemsets

The derivability of itemsets' support, given its subsets, is also an interestingness measure. Here we compare the algorithm with QIE and the one with Naïve inclusion exclusion (NIE). The results are presented in Figure 2.15. Please note the usage of absolute support and linear scale in Mushroom and Chess datasets. The reason is that the number of NDI in those datasets are very few, hence mining NDIs with larger minimum support values takes essentially no time.

While QIE does the inclusion-exclusion in a systematic manner, yielding an $O(|X|2^{|X|})$ time complexity, NIE does individually per subset support, yielding the time complexity of $O(3^{|X|})$. Hence, it is not surprising that QIE is consistently better than

Figure 2.16: Enumerating frequent itemsets in the presence of Bernoulli noise. Figures (a) to (c) plot the scalability of each algorithm with respect to σ , when p and q are fixed. Figures (d) to (e) plot the scalability when p and q vary. The specified values of p and q are the *mean* of the noise for each item, drawn by normal distribution.

NIE, especially in dense datasets.

FIM with Item-Dependent Bernoulli Noise

Sometimes, the noise level in every item (attribute) can be different. As mentioned in Section 2.5, this measure can be derived using subset supports. The noise in each item is randomly generated using normal distribution, with the mean as specified in the figures, and the standard deviations given as 0.001, 0.05, and 0.01 for BMS-Webview-1, Mushroom, and Chess datasets respectively. Furthermore, we force the probability between 0.5 and 1, but keep the mean as it is by rejecting values beyond the limit. The probability distributions used are identical for all experiments. Note that, as before, we do not actually contaminate the dataset with noise, but simply assume that they are noisy. In any case, the assumption does not matter since we only measure the efficiency, and not the accuracy.

As this problem has never been mentioned in previous publications, we compare our QIE based technique and the Naïve algorithm, and the results are shown in Figure 2.16. In Figure (a) to (c), we vary minimum support while fixing p and q , which is

the *mean* of the noise. In Figure (d) to (f), we vary the noise parameters p and q , while fixing minimum support. In this experiment, we once again see a consistent pattern that QIE is faster than the Naïve algorithm.

2.7 Concluding Remarks

In this chapter, we presented a thorough and systematic study of the problem for enumerating interesting itemsets. In a high level, our solution framework consists of two parts, namely the candidate generation module and the interestingness computation module.

We showed that the algorithms for the candidate generation module depend on the mining target (*all* interesting itemsets, *maximal* interesting itemsets, or *top-K* interesting itemsets) and the monotonicity property of the measure. We also studied the connection between various algorithmic optimizations with the specific problem instances, which led us to improve the efficiency of the mining algorithms significantly.

For the interestingness computation module, we proved that most previously proposed interestingness measures can be formulated using *partial supports* and *subset supports* of itemsets. We also presented the algorithms for calculating those *detailed supports* efficiently in all possible problem settings, including when the measure satisfies the weak anti-monotone property, and for enumerating *top-K* and maximal interesting itemsets.

We evaluated the efficiency of the algorithms for enumerating interesting itemsets, and showed that our algorithms are much more efficient than competing techniques. In particular, the Sup-RR algorithm is the most efficient for calculating partial supports, and the QIE algorithm is the most efficient for calculating generalized supports. These two algorithms require the support of all subsets to calculate the detailed-support of an itemset, and hence are only admissible for enumerating *all* and *top-K* interesting itemsets when the measure satisfies the anti-monotone property. For other problem settings, the *incremental* algorithm is the most efficient, except for calculating the partial supports in a dense database, where the Trans-RR algorithm is the most efficient.

While this chapter presents the generic solution, there are many additional tricks that can be employed for enumerating interesting itemsets in practice. In the next two chapters, we details the techniques to enumerate *correlated* itemsets and *fault-tolerant frequent* itemsets respectively. These are interesting, especially since enumerating itemsets with these two measures was considered intractable.

Chapter 3

Enumerating Correlated Itemsets

This chapter discusses one instance of the enumeration problem, where the interestingness μ of an itemset is defined as its *correlation*. An itemset is considered *correlated* if its support is much higher than its expected support under the independence assumption. This notion of correlation is analogous with *lift* or *interest* for measuring the strength of association rules, and has been successfully used for ranking interesting itemsets/rules, clustering, and co-clustering.

In this chapter, we prove that correlation in categorical databases satisfies the supweak monotone property, and hence correlated itemsets can be mined efficiently. Prior this discovery, mining correlated itemsets are considered intractable. We also show that, in some cases, it is desirable to integrate the closed and support constraints, on top of the correlation measure. We then propose efficient algorithms to integrate these two constraints while enumerating top- K correlated itemsets.

3.1 Preliminaries

3.1.1 Motivation

Interesting itemset is defined as a set of items which are interesting as a group. Naturally, a *group* of items is interesting if all items in the group are correlated with each other [98]. The correlation of an itemset is measured as the ratio between its relative support and its expected relative support [98], assuming all attributes are independent [2, 87, 86, 108, 100]. This measure reflects the correlation of the set, since an itemset with a high value for this measure contains items that are truly correlated with each other (beyond our expectation). This measure is analogous to *lift*, *interest*, and *surprisingness* [29], which have been successfully used for ranking

ID	Hair	Feathers	Fins	Legs	#obj	corr	Desc.
1	No	No	Yes	0	15	8.41	Fish
2	No	Yes	No	2	20	7.83	Birds
3	No	–	Yes	0	15	6.75	Fish
4	No	No	No	5	1	2.61	Starfish
5	No	–	No	–	43	0.89	?

Table 3.1: Comparison between correlation and support of sample itemsets in Zoo data.

interesting itemsets/rules [2, 87, 86], clustering [108], and co-clustering [100].

Example 3.1. Table 3.1 shows five itemsets (not the top-5 by any measure) in Zoo data, projected on four attributes, viz., hair, feathers, fins, and number of legs. Value ‘–’ in the table means that the corresponding attribute is not used (not part of the itemset). The first itemset {no hair, no feathers, fins, no legs} covers 15 animals, and has a very high correlation. This is because “fins” is highly associated with fish, which have no hair, no feathers, and no legs. On the other hand, though the fifth itemset {no hair, no fins} covers more animals, it has a very low correlation. This happens because the dataset has many animals which either have hair but no fins (mammals), or have fins but no hair (fish).

From the above example, we can see that highly correlated itemsets form an intuitive and natural grouping of objects, such as fish or birds. On the contrary, itemsets with low correlation, despite being extremely frequent, can be quite meaningless!

Having proven useful, we note that there is still no single algorithm that can enumerate correlated itemsets efficiently and completely. Previous works either combine correlation with other measures to prune the search space (use it as a post-processing) [2, 87, 86], or do a greedy/heuristic search to allow feasible enumeration, while compromising the completeness [108, 100]. The difficulty of enumerating interesting itemsets has been attributed to the lack of the anti-monotone property [108, 48].

Looking deeper into the running example, we also notice that even highly correlated itemsets may be uninteresting. For example, though the third itemset has high correlation, it is a subset of the first, and they cover exactly the same objects. Such (non-closed) itemsets are considered redundant in many applications. In another observation, we find that though the fourth itemset has high correlation, it covers only one object (starfish). These cases indicate that in order to filter-out uninteresting itemsets, we need to integrate other constraints, particularly *closed* and

support constraints. These requirements and issues lead us to another challenging research problem, i.e., to efficiently integrate closed and support constraints while enumerating correlated itemsets. Note that these constraints are optional, as some applications require non-closed itemsets [49] or infrequent [59] itemsets as well.

3.1.2 Related Work

This chapter discusses a simple and intuitive formulation of correlation, denoted as *corr*, which is the ratio between an itemset’s frequency and its expected frequency assuming all items are independent [2, 87, 86, 108]. An itemset with large *corr* appears much more often than it is expected to. This formula is analogous with *lift* and *interest* [17] for association rules.

There are several other correlation measures of itemsets, such as: χ^2 [56], *cosine similarity*, *all-confidence* [59, 48], *productivity* [98], and normalized mutual information (NMI) [42]. The χ^2 measure calculates the difference between the frequency and the expected frequency of all subsets (usually illustrated using contingency tables). Cosine similarity compares the frequency of the itemset with the *geometric mean* of the individual items’ frequency, instead of the expected frequency. All-confidence and productivity measure correlation as the *weakest* association rule, based on confidence and lift respectively, that produces the itemset. NMI calculates the minimum mutual information between all pairs of items in the itemset. There are also existing algorithms to efficiently enumerate interesting itemsets with those measures [56, 59, 101].

Rather than arguing the goodness of the measure (with respect to the usability and the efficiency to mine), our intention is mainly to present strategies to enumerate correlated itemsets with *corr* formulation. To solve this problem, most previous works prune the search space using additional constraints like frequency, and then merely scan the results to find the correlated itemsets. Conversely, we may traverse only the (potentially) interesting itemsets [100], with the risk of missing some correlated itemsets. As our proposed novel strategies enables efficient and complete mining of correlated itemsets, we conclusively advance the state-of-the-art for enumerating correlated itemsets.

3.1.3 Basic Definitions and Notations

In this chapter, we deal with a categorical database. Categorical database is a database containing n objects O and t attributes A , where the domain of each attribute is discrete values (categories). Each attribute $A_i \in A$ has domain D_i . Relating

to the frequent itemset mining paradigm, we can treat each object as a transaction, each distinct category (value of an attribute) as an item. In other words, $T = O$ and $I = \bigcup_i D_i$. A transaction (object) y contains an item $x \in D_i$, if y possesses x as the value in one of its attribute. As before, the total number of items in a database is denoted as m .

The relative support of an itemset X represents the probability of itemset X occurring in the database. When all attributes are independent, the expected relative support of an itemset X can be estimated as the product of each item's relative support, i.e.,

$$rsup(X) = \prod_{a \in X} rsup(a) \quad (3.1)$$

An itemset X is considered correlated if itemset X appears much more often than the expectation, i.e., if $rsup(X)$ is much greater than $rsup(X)$. Formally, the correlation of an itemset X , denoted as $corr(X)$, is defined as the proportion between its relative support and its expected relative support, i.e.,

$$corr(X) = \frac{rsup(X)}{rsup(X)}. \quad (3.2)$$

Our objective can then be defined as enumerating itemsets with large $corr(X)$. In this chapter, we focus on enumerating the top- K correlated itemsets. As required, additional constraints may be integrated into the basic problem, to enumerate correlated itemsets that are also closed and/or frequent.

3.1.4 Differences with the General Framework

While the general framework presented in Chapter 2 is suitable for enumerating interesting itemsets, enumerating correlated itemsets deserve special attentions.

Firstly, we benefit by scoping the problem, from the general transactional database into categorical database. In our problem, the scoping introduces the sup-weak monotone property to the measure, which did not exist beforehand. To the best of our knowledge, this scoping technique along with the induced sup-weak monotone property are novel in pattern mining area.

Secondly, and related with the first point, we demonstrate the usage of the sup-weak monotone property. It is interesting since this property is applied to a well-grounded problem, to enumerate correlated itemsets, which was previously considered intractable.

Thirdly, we present the challenges and the algorithms to integrate the support and closed constraints. One major challenge is to integrate support constraint, where the sup-weak monotone property is lost. For this issue, we propose a general solution to integrate the support constraint for mining interesting itemsets.

Lastly, we present efficient candidate generation and support-counting algorithms for enumerating interesting itemsets, where the measure satisfies the sup-weak monotone property. Even though the general idea is presented in Section 2.2.4, it is not discussed in detail as this topic not been deeply studied previously.

3.2 Algorithms

This section presents the algorithm for enumerating correlated itemsets. First, we discover that itemset correlation in categorical databases satisfies the sup-weak monotone property. Based on this property, we design an algorithm for the unconstrained problem, and then show how to integrate the closed and support constraints.

3.2.1 Monotonicity Property of Correlation

In categorical databases, each object has a value in *every* attribute. As a result of which, we have the following conditions: (1) all objects contain exactly t items, and (2) all items in one attribute form a discrete probability distribution. Because of these conditions, correlation of itemsets in categorical database satisfies supweak monotone property (c.f., Section 2.2.1), as formally presented in the following theorem.

Theorem 3.1. *In categorical databases, correlation satisfies the supweak monotone property. In other words, for any attribute A_i not used by itemset X , there exists a value $a \in D_i$, such that $\text{corr}(Xa) \geq \text{corr}(X)$.*

Proof. Consider an itemset X which does not contain any item (category) from attribute A_i (with domain D_i). From Equation (3.2), $\text{corr}(Xa) \geq \text{corr}(X)$ if and only if $p(a|X) \geq p(a)$. As $\sum_{a \in D_i} p(a|X) = \sum_{a \in D_i} p(a) = 1$, then there exists at least one item $a \in D_i$ with $p(a|X) \geq p(a)$, since otherwise, $\sum_{a \in D_i} p(a|X) < 1$. $\square$

Informally, this theorem states that we can always extend an itemset without decreasing its correlation. Another consequence of Theorem 3.1 is that the size of the most significant itemsets is always equal to t , as formally presented in following corollary.

Proposition 3.2. *All itemsets with size greater than t have relative support (and hence the correlation) equals to zero.*

This proposition follows the fact that all objects contain exactly t items.

Corollary 3.3. *There exists an itemset X of size t such that $\text{corr}(X)$ is maximum.*

Proof. Assuming the most correlated itemset contains less than t items, from Theorem 3.1, we can keep extending this itemset with other items without decreasing its correlation until its size becomes t . $\square$

3.2.2 Enumerating Correlated Itemsets

As mentioned in Section 2.2.4, candidates can be generated for functions satisfying the supweak monotone property in a similar (reversed) way as for functions satisfying the weak anti-monotone property. That is, instead of starting with the empty set and extending the itemset, we start with the full set and keep shrinking the itemset. In this case, the *maximal* itemsets are the sets of all items (of size t) contained in each object. Therefore, we do not start the enumeration from m -itemsets, but from all t -itemsets contained by each transaction.

The remaining challenge is to calculate the correlation of each itemset efficiently. Firstly, we tackle the problem to efficiently calculate the relative support of an itemset. Once we know the relative support, the correlation of an itemset can be calculated trivially. However, this task is not as easy as it may seem.

In the usual enumeration problem when we extend the itemsets, we can calculate the relative support of an itemset XY by calculating the relative support of Y in the database projected to X , i.e., we only need to consider transactions containing X . This technique is also called the *database projection technique*, which is the foundation of many efficient itemset enumeration algorithm (c.f., Section 2.3.2). However, this technique cannot be trivially adopted for the case when we shrink the itemsets. For example, let the current itemset $X = abcd$. Having calculated the relative support or the supporting transactions of $abcd$, it is not trivial to calculate $\text{sup}(abc)$ without re-scanning the database. On the contrary, if we have the supporting transactions of abc , we can filter transactions that contain item d to calculate $\text{sup}(abcd)$.

Here, we propose an algorithm to calculate the relative support of *all* direct subsets of an itemset in *one* database scan, as illustrated in Algorithm 3.1. This way is much more efficient than the naïve way, by re-scanning the database for calculating the interestingness of each subset. For *processing* an itemset X , the naïve algorithm

Algorithm 3.1: Generating direct subsets

Input: itemset X **Output:** The relative support of all direct subsets

```

1 for  $o \in T$  do
2    $Y \leftarrow I(o) \cap X$ ;
3   if  $Y = X$  then
4     for  $a \in X$  do
5        $\perp$  increment  $rsup(X \setminus a)$  by  $1/n$ ;
6   else if  $|Y| = |X| - 1$  then
7      $\perp$  increment  $rsup(Y)$  by  $1/n$ ;

```

Algorithm 3.2: Generating maximal closed subsets

Input: A closed itemset X **Output:** closed candidate children

```

1  $Result \leftarrow \emptyset$ ;
2 for  $o \in T \setminus cov(X)$  do
3    $\perp$   $Result \leftarrow Result \cup \{X \cap I(o)\}$ ;
4 return  $extractMaximal(Result)$ ;

```

requires $O(n |X|^2)$ iterations (there are $|X|$ direct subsets, and calculating the relative support of each requires $O(n |X|)$ iterations), while our proposed algorithm requires only $O(n |X|)$ iterations.

3.2.3 Integrating Closed Constraint

Next, we discuss how to integrate the closed constraint into the basic enumeration problem. The simplest way to achieve this is by first enumerating all correlated itemsets, and then filtering out those which are not closed. However, this approach is not efficient since a lot of correlated itemsets are not closed. The ideal case is to enumerate the closed itemsets directly.

Our strategy is by realizing that closed itemsets is a *formal concept* [106]. We know that the intersection of two formal concepts is also a formal concept, and hence, so is a closed itemset. Having this property, we can generate only the closed subsets by using Algorithm 3.2.

As correlation satisfies the supweak monotone property, it is guaranteed that all itemsets have at least one superset with non-lower correlation. However, as Algorithm 3.2 generates only the closed itemsets, we need a stronger property to ensure its

correctness.

Lemma 3.4. *For any closed itemset X with $|X| < m$, there exists another itemset $Y \supset X$, $\text{corr}(Y) \geq \text{corr}(X)$ and object o , such that $Y \cap I(o) = X$.*

Proof. We prove this constructively as follows. From Theorem 3.1, we know that there exists some item a such that $\text{corr}(Xa) \geq \text{corr}(X)$. We take Y' as the closure of Xa . If there exists an object o in $\text{cov}(X)$ and not in $\text{cov}(b)$ for any $b \in Y' \setminus X$, then we are done. Note that o always exists if Xa is closed. Otherwise, we take an item $b \in Y'$ such that $\text{cov}(Xa) \subset \text{cov}(Xb)$. If $\text{corr}(Xb) \geq \text{corr}(X)$, we can repeat the procedure with the closure of Xb . For the other case, we know that there is a category b' in the same attribute as b , such that $\text{corr}(Xb') \geq \text{corr}(X)$, so we can repeat the procedure with the closure of Xb' . This procedure is guaranteed to terminate in both cases as we have removed one attribute (the one containing a), and the number of attributes is finite. $\square$

Analyzing the complexity, the for-loop in Lines 2–3 can be performed in $O(n|X|)$ time using a hash table. For extracting maximal sets, the current fastest algorithm has $O(N^2 \lg \lg N / \lg^2 N)$ time complexity [79], where N is the sum of the set cardinalities. However, this approach is only of theoretical interest, since it assumes the worst case distribution. In our implementation, we use a simple “sort-and-check” algorithm, i.e., we sort and iterate the sets in decreasing order of their length, and then verify the maximality of each set by iterating the found maximal itemsets. The time complexity is $O(c \cdot c' \cdot |X|)$, where c is the number of sets, and c' is the number of maximal sets. As $c' \ll c$ and $c \leq \min(n, 2^{|X|})$, this algorithm has polynomial delay time complexity, and runs pretty fast in practice.

3.2.4 Integrating Support Constraint

In this section, we discuss the algorithm to integrate support constraint to the basic enumeration problem, by constraining the resulting itemsets to have relative support at least rminsup . For presentation purpose, we assume that the closed constraint is also integrated, even though all concepts explained in this section remain valid without this assumption.

This problem admits two extreme solutions. The first approach enumerates all closed frequent itemsets (CFI), and then filters those with high correlation. This approach is efficient when rminsup is reasonably high, thanks to the development of CFI mining algorithms. The second approach ignores the support constraint, by

enumerating all correlated itemsets, and then filters-out those which are infrequent. This approach is efficient when the correlation threshold is very high.

One intuitive way to improve the second approach is by starting the enumeration from the maximal frequent itemsets (MFI) instead of from all t -itemsets. Note that mining MFIs is usually much faster than mining CFIs as $|\text{MFI}| \ll |\text{CFI}|$. As we keep removing items in each iteration, all enumerated itemsets are also frequent.

Unfortunately, this approach cannot guarantee the completeness of the enumeration results, because there could be a non-maximal frequent itemset which does not have frequent supersets with higher correlation.

Example 3.2. Consider an itemset X and an attribute A (not used by X) with values a_1 and a_2 . From Theorem 3.1, we know that at least one among Xa_1 and Xa_2 have correlation higher than X . Suppose $\text{corr}(Xa_1) > \text{corr}(X) > \text{corr}(Xa_2)$. Being highly correlated, it is possible that itemset Xa_1 is infrequent, and itemset Xa_2 is frequent (item a_1 has much lower relative support than a_2). In this situation, X is not maximal (and hence not part of the initialization) and has no frequent superset with greater correlation.

Basic Algorithm

As the *supweak monotone* property is lost in this case, we enumerate the correlated itemset by using a bounding function, $ub(X) \geq \text{corr}(X)$, that satisfies the monotone property, i.e., $ub(X) \geq ub(Y)$ for all $Y \subseteq X$. In other words, we loosen our target, to mine the supersets of the correlated itemsets, which are those with upper bound larger than the correlation of the current K -th most correlated itemset. Such itemsets are denoted as *potential itemsets*, which can then be refined to attain the correlated itemsets.

Having the monotone upper bound function, we can use exactly the same techniques as before (Algorithms 3.1 and 3.2). The only difference is that we now use $ub(X)$ instead of $\text{corr}(X)$ for generating the itemsets. The remaining step is to derive the tightest $ub(X)$. From Equation (3.2), we have $\text{corr}(X) = \frac{\text{sup}(X)}{r\hat{\text{sup}}(X)}$. Our strategy is to keep the denominator $r\hat{\text{sup}}(X)$ as it is, and to set the numerator as the upper bound of $\text{sup}(X)$, i.e., a function whose value is no less than that of all subsets of X . One possible function is the maximum relative support of X 's 2-subsets. We do not need to consider 1-itemsets since their correlation is always 1.0 (equal to random chance) which is not interesting. However, this bound is very loose, yielding

an inefficient algorithm. Unfortunately, it is very difficult to derive a tighter upper bound.

Two-Phase Algorithm

Previously, we saw that it was difficult to derive a tight bound because we needed to consider all subsets of an itemset. To resolve this, we split the problem into two phases: (1) enumerating correlated itemsets with relative support no less than $rminsup'$, and (2) enumerating those between $rminsup$ and $rminsup'$. As all itemsets with relative support no lesser than $rminsup'$ have already been taken care of by the first phase, they do not need to be considered in the second phase.

For the first phase, assuming that the $rminsup'$ is large enough, we can mine all CFIs and filter results, as mentioned as the first extreme approach in the early part of this section. For the second phase, we can loosen the function $ub(X)$ into $ub(X, rminsup')$, which guarantees to be no less than $\mu(X)$ and satisfies monotone property, *among itemsets whose relative support are no less than $rminsup'$* . One trivial upper bound function is: $ub(X, rminsup') = \frac{rminsup'}{r\hat{sup}(X)}$. With this technique, we manage to solve the problem reasonably efficiently in real-world databases.

3.3 Experimental Results

We used three real categorical datasets, viz., Chess, Soybean, and Pendigits for the evaluating the efficiency and scalability of our algorithm, and Zoo dataset for the case study. Details of these datasets are available in Appendix A. For the MFI and the CFI mining algorithms, we used highly optimized implementations by Zhu and Grahne [112], which are available from the FIMI website.

3.3.1 Without Support Constraint

The results of this experiment are depicted in Figure 3.1, where we vary parameter K . Line “All” corresponds to the time taken for mining top- K correlated itemsets without the closed constraint, and “Closed” for those with the closed constraint.

We see that the scalability of our algorithms is similar across datasets. This is consistent with our analysis, since given the dataset, the time complexity is linear to K . We can also see that enumerating without the closed constraint takes much less time than enumerating with the closed constraint, which is again consistent with our

3.3. EXPERIMENTAL RESULTS

77

Figure 3.1: Enumerating top- K correlated itemsets.

theoretical analysis in Section 3.2.2 and 3.2.3. Regarding practicality, our algorithm takes around 10 seconds for mining 1,000 itemsets, which should satisfy most practical needs. We expect that the practical value for analysts is in the order of tens, or at most a hundred (e.g., [54, 104, 27]).

3.3.2 With Support Constraint

Next, we analyze the efficiency of our algorithm when the support constraint is integrated. Note that the closed constraint is always integrated since otherwise there is no competitor, i.e., mining all frequent itemsets is too slow. Moreover, as shown in previous experiment, mining all itemsets is much faster than mining closed itemsets. The naïve algorithm refers to mining all CFIs, and then taking the top- K . As mentioned in Section 3.2.4, our algorithm consists of two phases. The first phase gets the top- K correlated itemsets with minimum relative support $rminsup'$ (using the naïve algorithm), which are then used by the second phase to mine the top- K correlated itemsets with minimum relative support between $rminsup$ and $rminsup'$.

The experimental results are depicted in Figure 3.2, with the time taken for the first and the second phases denoted as P1 and P2 respectively. Variable $rminsup$ is denoted as rms in Figures (a)–(c) for presentation purpose. First, we analyze the scalability with respect to $rminsup$, as plotted in Figures (a) to (c). We fix the parameter K as 10, which we consider as the most practical setting, and use two possible values for $rminsup'$ as shown in each plot. We see that in all datasets, our algorithm is much faster than the naïve algorithm. We can also observe that, depending on $rminsup'$, the time taken by our algorithm could be dominated by either the first or the second phase.

Next, we investigate the impact of $rminsup'$, as plotted in Figures (d) to (f).

Figure 3.2: Enumerating frequent closed top- K correlated itemsets. The value of K is set to 10 in Figures (a) to (f).

As expected, the time taken for the first phase decreases while $rminsup'$ increases, and that for the second phase increases while $rminsup'$ decreases. This experiment shows that $rminsup'$ should be tuned properly to get the optimal efficiency. From a practical perspective, we may consider the first phase as a one-time cost, irrespective of K (assuming that we sort all CFIs by correlations) and $rminsup$. Therefore, we can simply set $rminsup'$ as low as possible, i.e., to the lowest that we can tolerate with the naïve approach. Once we have the result, we can reuse it for mining with $rminsup < rminsup'$.

Lastly, we analyze the impact of K , as depicted in Figures (g) to (i). We may observe that our algorithm is not very sensitive to K for reasonable value of K (≤ 100).

3.4. CONCLUDING REMARKS

79

Rank	Hair	Feather	Fins	Legs	#objects	corr
1	No	No	Yes	0	15	8.41
2	No	Yes	No	2	20	7.83
3	No	–	Yes	0	15	6.75
4	No	Yes	–	2	20	6.51
5	–	No	Yes	0	16	5.15
6	–	Yes	No	2	20	4.50
7	–	–	Yes	0	16	4.13
8	–	Yes	–	2	20	3.74
9	Yes	No	No	4	31	2.87
10	No	No	No	5	1	2.61

Table 3.2: Top-10 correlated itemsets in Zoo data, projected to four attributes.

When K reaches 1,000, our second phase becomes slow, especially when $rminsup'$ is large. This is because when K and $rminsup'$ are large, our upper bound function becomes loose (Section 3.2.4). Hence, our algorithm encounters many uncorrelated itemsets. Nevertheless, we do not consider such an inefficiency as a limitation, as analysts will not need K as high as 1,000 anyway.

3.3.3 Case Study on Additional Constraints

Here, we analyze the top-10 correlated itemsets of Zoo data (projected on the four presented attributes) as shown in Table 3.2. From the table, we see many non-closed itemsets are among the top, i.e., the first itemset is the closure of the third, the second itemset is the closure of the fourth, sixth, and eighth, and the fifth is the closure of the seventh. As they cover the same objects, they do not provide more information. Moreover, they can hide some truly interesting itemsets. For example, if we only select top-5, we will miss the class of mammals (ninth itemset). Therefore, we suggest to integrate closed constraints. We also see that the tenth itemset is highly correlated, despite only covering one object (starfish). This demonstrates the need for integrating the support constraint.

3.4 Concluding Remarks

We proposed an efficient algorithm for enumerating correlated itemsets, where correlation is defined as the proportion between relative support and expected relative support. Previously, this problem was considered intractable as correlation does not

satisfy the anti-monotone property. Our strategy is by scoping the problem to categorical database, and realizing that correlation satisfies the supweak monotone property in such databases.

We also proposed algorithms to efficiently integrate *closed* and *support* constraints. Integrating these constraints causes the measure to lose its supweak monotone property, and hence the mining process becomes difficult. We solved this by splitting the search space into two parts, viz., low relative support and high relative support itemsets, and solved each part individually. The notion of low and high relative support is determined by a user-defined parameter $rminsup'$. In particular, we derived a tight upper bound for correlation, hence allowing efficient enumeration, if we restrict the itemsets to have low support. The remaining itemsets can be obtained by post-processing all itemsets with high support.

From extensive empirical evaluations, we showed that our algorithms enumerate correlated itemsets efficiently. Without the support constraint, our algorithm enumerate thousands of correlated itemsets within seconds. When support and closed constraints are integrated, our algorithm beats the highly optimized implementation of the naïve algorithm. We also presented case studies that show the usefulness of integrating these two constraints.

To conclude, this chapter shows that correlated itemsets can be mined efficiently. In the next chapter, we present our algorithm to efficiently enumerate fault-tolerant frequent itemsets, which was considered intractable previously.

Chapter 4

Enumerating Fault-Tolerant Frequent Itemsets

This chapter discusses the problem to enumerate fault-tolerant frequent itemsets (FTFI). This problem is one instance of the interesting itemset enumeration problem, which considers the interestingness function μ to be the fault-tolerant (FT)-support of an itemset. In brief, FT-support calculates the frequency or support of an itemset, if the transactions are allowed to lack few items. This measure is useful for discovering the generalized knowledge of itemsets, which has been shown useful in various applications and domains.

There are many variants of FT-support. For the most general variant, there is still no efficient and complete algorithm to enumerate FT-frequent itemsets. The main difficulties are due to the lack of anti-monotone property of the FT-support, and the calculation of the FT-support of an itemset. These two challenges relate to the candidate generation and the interestingness computation modules, respectively.

In this chapter, we propose the first efficient and complete algorithm, for enumerating the general FT-frequent itemsets.

4.1 Preliminaries

4.1.1 Motivation

The support of an itemset X counts the number of transactions containing *all* items in X . Recent research initiatives in generalized knowledge discovery have questioned the strict definition of support, and led to the idea of *relaxing* it in various ways.

Figure 4.1: Application of FT-itemsets for subspace clustering.

The main purpose of introducing such flexibility is to attain generalized information, providing useful knowledge or patterns to analysts, which otherwise would remain undiscovered [105]. Such fault-tolerant (FT)-itemsets have been shown to be extremely useful in various applications and domains, e.g., for initializing the seed of EM clustering [105], discovering fault-tolerant association rules [69], discovering interesting patterns [12, 85], reconstructing noisy databases [12, 52, 23, 33], clustering, and classification [52, 23, 33]. One application of FT-itemsets for subspace clustering is illustrated as follows.

Example 4.1. Consider a transactional database as depicted in Figure 4.1, presented in matrix form. In this database, analysts are usually interested in the (6×5) matrix, which is marked by the black rectangle. In this matrix, *almost* all transactions contain *almost* all items, which then forms a *fault-tolerant pattern*. When we restrict all transactions to contain all items, as in the usual definition of frequent itemsets, we can only discover smaller matrices as marked by the dotted rectangles, where most of them are overlapping.

The fault-tolerant support of an itemset X counts the maximum number of transactions Y , such that the matrix $Y \times X$ satisfies the user-specified relaxation criteria. To differentiate the mining target X and the specifying criterion $Y \times X$, we denote the former as *itemset* and the latter as *pattern*.

4.1.2 Basic Definitions and Notations

The notion of fault-tolerance *relaxes* the requirement for a transaction (or a transaction-set) supporting an itemset, by allowing the transactions to lack few items. We consider

three relaxation criteria, namely *transaction relaxation*, *item relaxation*, and *pattern relaxation*, to define the notion of *valid* patterns.

Definition 4.1 (Transaction Relaxation). A pattern $P = Y \times X$ satisfies *transaction relaxation* if all its transactions contain most of its items. Mathematically, $\forall y \in Y, |X \setminus I(y)| \leq e_t$, where e_t is the maximum allowed number of items in X to be missed by any transaction in Y . The error tolerance e_t can be defined either as a constant value, or in proportion to the number of items $|X|$. The former is denoted as ε_t^c and the latter is denoted as ε_t .

Definition 4.2 (Item relaxation). A pattern $P = Y \times X$ satisfies *item relaxation* if all its items are contained by most of its transactions. Mathematically, $\forall x \in X, |Y \setminus T(x)| \leq e_i$, where e_i is the maximum allowed number of transactions in Y missing any item in X . As in transaction relaxation, the error tolerance e_i can be defined either as a constant value, or in proportion to the number of transactions $|Y|$. The former is denoted as ε_i^c and the latter is denoted as ε_i .

Definition 4.3 (Pattern relaxation). A pattern $P = Y \times X$ satisfies *pattern relaxation* if the total number of missing items by all transactions is small. Mathematically, $\left(\sum_{y \in Y} |X \setminus I(y)|\right) \leq e_p$, where e_p is the maximum allowed total number of missing items in P . As before, the error tolerance e_p can be defined either as a constant value, or in proportion to the size of the patterns $|Y| \cdot |X|$. The former is denoted as ε_p^c and the latter is denoted as ε_p .

We require all relaxations $\varepsilon \neq 0$, since otherwise, the problem degrades into the one without fault-tolerance. When a relaxation criteria is *imposed* or *considered*, it means the corresponding error tolerance is less than 1 for proportional relaxations, or ∞ for constant relaxations, and vice versa.

Having defined the relaxation criteria, a *valid* pattern is simply a pattern that satisfies all specified relaxation criteria. Having these definitions, we can define the FT-support of an itemset as the maximum-sized transaction-set that forms valid pattern, which is formally stated as follows.

Definition 4.4 (Fault-tolerant Support (FT-support)). A transaction-set Y FT-supports an itemset X if $Y \times X$ is valid. The FT-supporting-transactions of X , denoted as $\mathcal{S}_{FT}(X)$, is the maximum-sized transaction-set that FT-supports X . The FT-support of X , denoted as $sup_{FT}(X)$, counts the number of transactions in $\mathcal{S}_{FT}(X)$. The notion of $sup_{FT}(X)$ can also be parameterized by the error tolerance, viz., $sup_{FT}(X, \varepsilon_t, \varepsilon_i, \varepsilon_p)$ for proportional relaxations, and $sup_{FT}(X, \varepsilon_t^c, \varepsilon_i^c, \varepsilon_p^c)$ for constant relaxation.

We also generalize the notion of support by parameterizing it with the respective transaction-set. That is, the number of transactions in Y that contains X , is denoted as $sup^Y(X)$.

4.1.3 Problem Variants

We consider three variants of FTFI. In the first variant, we consider only constant transaction relaxation (ε_t^c). This variant is one of the pioneering definition [69], which is also the most well studied. In the second variant, we consider all constant relaxations ($\varepsilon_t^c, \varepsilon_i^c, \varepsilon_p^c$), and in the third variant, we also consider proportional relaxations ($\varepsilon_t, \varepsilon_i, \varepsilon_c$).

These three variants are discussed in Sections 4.2, 4.3, and 4.4 respectively.

4.1.4 Differences with the General Framework

The focus of this chapter is to enumerate fault-tolerant frequent itemsets. As mentioned above, this problem is difficult because FT-support does not satisfy the anti-monotone property, and also is difficult to calculate. Each of these sub-problems was considered intractable, and was previously solved using sub-optimal heuristic solutions.

We proposed several novel ideas to handle these difficulties. In order to handle the lack of anti-monotone property, we need to derive a bounding function ub . Here, we propose a way to derive ub by analyzing the maximal size of interesting itemsets, the individual transaction, and also the characteristic of each individual constraint. For calculating the FT-support, we propose an algorithm that make use of both the *partial supports* and *subset supports* of itemsets (c.f., Chapter 2). In particular, we formulate the fault-tolerant support as a linear function of partial supports and subset supports, yielding an integer linear programming problem. Through extensive empirical evaluations, we demonstrated that our proposed algorithms can efficiently and completely enumerate fault-tolerant frequent itemsets in real-world problem settings.

While the formulation is specific to this problem, these strategies can be applied to derive any upper bound function and to calculate the interestingness of an itemset. So in a sense, this chapter demonstrates a way to enumerate interesting itemsets with a difficult interestingness function.

4.2 Constant Transaction Relaxation ($\varepsilon_t^c \neq \infty$)

In this section, we consider the simplest and the most common variant of FTFI, by only considering constant transaction relaxation [69]. With this definition, we can process each transaction independently, as in, the FT-support of an itemset is simply the number of transactions that lack no more than ε_t^c items in it.

Lemma 4.1. *The function $\text{sup}_{FT}(X, \varepsilon_t^c)$ is anti-monotone [69].*

Proof. Adding an item to the itemset will never reduce the number of items missed by a transaction. As the maximum allowed error is defined as a constant value, a transaction that does not FT-support itemset X will never support itemset XY . Therefore, $\text{sup}_{FT}(X, \varepsilon_t^c)$ is anti-monotone. $\square$

We can observe that this notion of FT-support – $\text{sup}_{FT}(X, \varepsilon_t^c)$, is identical to a partial support $\text{sup}_{\geq |X| - \varepsilon_t^c}(X)$ (c.f., Chapter 2). To recall, $\text{sup}_{\geq k}(X)$ counts the number of transactions containing at least k items in X . Therefore, we can use partial supports to calculate the FT-support. The algorithm for calculating partial supports of an itemset has been discussed thoroughly in Section 2.3.4.

Noting that we never need partial supports representing transactions lacking more than ε items, we can improve the efficiency of the algorithms for calculating the partial supports. With this restriction, the naïve and counting algorithms still need to iterate all transactions and items, therefore these algorithms do not seem to be improved. Next, we analyze Trans-RR algorithm. From Equation (2.3), we can see that the recurrence relation of transactions lacking k items depends only on those lacking k and $k - 1$ items. Therefore, if only transactions lacking at most ε items are required, then those lacking more than ε items can be completely ignored. Based on this reasoning, the time complexity improves from $O(|T| |X|)$ to $O(|T| \varepsilon)$, and the space complexity from $O(|T| |X|^2)$ to $O(|T| |X| \varepsilon)$. A similar case applies for QIE where the time complexity improves from $O(|X| 2^{|X|})$ to $O(|X| |X|^\varepsilon)$, and also for Sup-RR where the time complexity improves from $O(|X|^2)$ to $O(|X| \varepsilon)$, and the space complexity from $O(K |X|)$ to $O(K \varepsilon)$.

4.2.1 Integrating Minimum Item Support Threshold

By only checking whether a transaction FT-supports an itemset, there is a drawback that an itemset could be considered FT-frequent even though not all items are *well-supported*, as illustrated in following example.

Example 4.2. Consider a case where 100 transactions containing an itemset abc , and none of them contains item d . When $\varepsilon_t^c = 1$, we have $\text{sup}_{FT}(abcd) = 100$. Despite being mathematically correct, the statement that $abcd$ has a FT-support of 100 is misleading.

To avoid such a case, Pei et al. [69] imposed an additional constraint which requires all items in the itemset to be *frequent* in the supporting transactions. Mathematically, if the FT-supporting transactions of X is $\mathcal{S}_{FT}(X) = \mathcal{S}_{\geq |X| - \varepsilon_t^c}(X)$, this constraint requires $\text{sup}^{\mathcal{S}_{FT}(X)}(a) \geq \sigma^{item}$.

Calculating $\text{sup}^{\mathcal{S}_{FT}(X)}(a)$ is trivial for the Naïve, incremental, QIE, and Trans-RR algorithms. However, to modify the Sup-RR algorithm is not trivial. Fortunately, the number of transactions containing exactly k items (including item a) of X , can also be calculated as a recursive function of partial supports. Recalling that $\mathcal{S}_{=k}(X)$ stores the set of transactions containing exactly k items in X , we have:

$$\text{sup}^{\mathcal{S}_{=k}(X)}(a) = \text{sup}_{\geq k}(X) - \text{sup}_{\geq k}(X \setminus \{a\})$$

Proof. The partial support $\text{sup}_{\geq k}(X \setminus \{a\})$ counts transactions that either (1) contain *at least* $k + 1$ items of X which is $\text{sup}_{\geq k+1}(X)$, or (2) do not contain a but contain *exactly* k items of X . Similarly, we can split $\text{sup}_{\geq k}(X)$, into $\text{sup}_{=k}(X) + \text{sup}_{\geq k+1}(X)$.

Therefore, subtracting $\text{sup}_{\geq k}(X \setminus \{a\})$ from $\text{sup}_{\geq k}(X)$ gives transactions that contain a , and contain *exactly* k items of X , which is our target. $\square$

4.3 Constant Relaxation ($\varepsilon_t^c, \varepsilon_i^c, \varepsilon_p^c \neq \infty$)

In the previous section, we considered a transaction to FT-support an itemset if it contains *most* items in the set. To ensure that all items are contained in significant number of transactions, we required each item to appear in at least σ^{item} transactions of the supporting transaction-set. The limitation of this additional constraint is that the FT-support of an itemset does not depend on the occurrence per item. In other words, the minimum item support threshold is either satisfied or not, without affecting the FT-support, as illustrated in the following example.

Example 4.3. Consider a database with 1000 transactions FT-supporting itemset abc , with only 10 containing item c . If we let $\sigma^{item} = 10$, $\text{sup}_{FT}(abc)$ remains at 1000, which is misleading, because 990 of those transactions do not contain item c .

4.3. CONSTANT RELAXATION ($\varepsilon_T^C, \varepsilon_I^C, \varepsilon_P^C \neq \infty$)

87

To handle this issue, the notion of pattern and item relaxations have been introduced [85, 52], which also leads to our definition of FT-support (Definition 4.4). To recall, we first define the notion of *valid pattern*, which is a matrix formed by a set of transactions and items, by using several relaxation criteria. Having the notion of *valid pattern*, the FT-support of an itemset X is the largest transaction-set $|Y|$, such that the pattern $Y \times X$ is valid.

Example 4.4. Continuing Example 4.3, with item or pattern relaxation, e.g., $\varepsilon_i^c = 5$ or $\varepsilon_p^c = 5$, the FT-support of abc is upper bounded by 15, as there are only ten transactions containing item c .

4.3.1 Candidate Generation

As mentioned in Section 2.2, the candidate generation algorithm depends on the monotonicity property of the measure. Here, we show that FT-support is anti-monotone, if the relaxation criteria is given as a constant value.

Theorem 4.2. *The function $\text{sup}_{FT}(X, \varepsilon_t^c, \varepsilon_i^c \varepsilon_p^c)$ is anti-monotone [71].*

Proof. The allowed number of misses in a pattern is constant, and addition of items will never reduce the amount of misses. $\square$

Because FT-support satisfies the anti-monotone property, we can enumerate all, maximal, or top- K interesting FTFI using our algorithms presented in Section 2.2. The remaining problem is to compute the FT-support of an itemset.

4.3.2 Interestingness Computation

First, we formulate the interestingness computation problem as an integer linear programming problem. Based on this formulation, we prove that calculating the FT-support of an itemset is an NP-complete problem. We then present a pre-processing technique that can boost the efficiency of the counting, and an efficient calculation of the upper bound of the FT-support, to improve the efficiency of the enumeration.

Integer Linear Programming (ILP) Approach

The problem to find the FT-support of an itemset can be mapped into a binary linear programming problem (BLP), where we have n binary variables, representing whether each transaction is *inside* the supporting transaction-set. Mathematically, this BLP

is denoted by an integer array $A = (a_1, a_2, \dots, a_n)$. Each variable a_i , represents one transaction t_i and is constrained to be 0 or 1. A value 1 of variable a_i means transaction i is present in the tidset. The *relaxation criteria* can then be presented in terms of the variables:

Transaction Relaxation

$$\forall i \in \{1..n\}, a_i |X \setminus I(t_i)| \leq \epsilon_t^c \quad (4.1)$$

Item Relaxation

$$\forall x \in X, \sum_{j, t_j \notin T(x)} a_j \leq \epsilon_i^c \quad (4.2)$$

Pattern Relaxation

$$\sum_{i=1}^n a_i |X \setminus I(t_i)| \leq \epsilon_p^c \quad (4.3)$$

The FT-support of an itemset, which is the maximum sized valid transaction-set, can also be mapped into an objective function of a linear program. Mathematically, it is written as:

$$\sup_{FT}(X) = \max \sum_{i=1}^n a_i \quad (4.4)$$

Analyzing this linear program, calculating the FT-support of an itemset seems intractable since the number of transactions can be potentially very large. Moreover, the binary linear programming problem is in general NP-Hard. Fortunately, many variables are *equivalent*, which means, they appear in exactly the same set of constraints/inequalities. Depending on the considered relaxation criteria, there are two types of equivalence classes.

Definition 4.5 (E_{num}^X equivalence). Two transactions are E_{num}^X equivalent if they contain the same *number of items* from itemset X . Mathematically, $t_1 \equiv t_2 \leftrightarrow |I(t_1) \cap X| = |I(t_2) \cap X|$. Hence, for an itemset X , the number of equivalence classes is $|X| + 1$. The equivalence class for transactions containing i items from X is denoted as E_i^X . In a general context, the superscript X may be omitted.

Example 4.5. Consider following transactional database:

4.3. CONSTANT RELAXATION ($\varepsilon_T^C, \varepsilon_I^C, \varepsilon_P^C \neq \infty$)

89

TID	Itemset
1	c
2	b, c
3	a, b
4	a, b, c

Let $X = bc$. Here, we have three equivalence classes, viz, $E_0^X = \{\}$, $E_1^X = \{1, 3\}$, and $E_2^X = \{2, 4\}$.

Definition 4.6 (E_{iset}^X equivalence). Two transactions are E_{iset}^X equivalent if they contain the same *set of items* from itemset X . Mathematically, $t_1 \equiv t_2 \leftrightarrow I(t_1) \cap X = I(t_2) \cap X$. Hence, for an itemset X , the number of equivalence classes is $2^{|X|}$. The equivalence class for transactions containing itemset Y but not $X \setminus Y$ is denoted as E_Y^X .

Example 4.6. Consider the database in Example 4.5. Let $X = bc$. Here, we have four equivalence classes, viz., $E_\emptyset^X = \{\}$, $E_b^X = \{3\}$, $E_c^X = \{1\}$, and $E_{bc}^X = \{2, 4\}$.

By grouping equivalent transactions, we can reduce the number of variables in the BLP from n to the number of equivalence classes, i.e., $\min(n, 2^{|X|})$ and $\min(n, |X| + 1)$ for E_{iset}^X and E_{num}^X respectively (we can safely ignore equivalence classes with zero cardinality). Note that this number is a loose upper bound, as transactions lacking more than ε_t^c items can be ignored. For example, when $|X| = 20$ and $\varepsilon_t^c = 3$, the number of variables equals 1,351, which is much smaller than 2^{20} . The linear program is still the same, except each variable now represents an equivalence class instead of a transaction. Consequently, the variable is no longer binary (representing whether the transaction is taken or not), but integer (representing the number of transactions to be taken from the equivalence class).

Relating with our general framework for enumerating interesting itemsets (Section 2.3), E_{num}^X is essentially the partial-supports of X , and E_{iset}^X is the subset-supports of X . Therefore, we can use the algorithms in Section 2.3 to extract the equivalence classes and form the linear program.

Finding the Solution to ILP

We split the problem into three cases, based on the considered relaxation criteria.

Case 1: The first case is when only the transaction relaxation is considered ($\varepsilon_i^c = \varepsilon_p^c = \infty$). This case is the same as in Section 4.2, where the FT-support of an itemset is simply the number of transactions FT-supporting the itemset.

Case 2: The second case is when the pattern relaxation is also considered ($\varepsilon_i^c = \infty$). In this case, the supporting transactions of an itemset X can be found by repeatedly taking transactions with the lowest number of *misses*, until the total number of *misses* exceeds ε_p .

Lemma 4.3. *When $\varepsilon_i = \infty$ and $\varepsilon_p \neq \infty$, the maximum set of transactions that FT-supports X can be found by repeatedly taking transactions lacking the minimum number of items in X .*

Proof. Let $miss(t)$ be the number of items in X which are missed by transaction t , and Y be the optimal (maximum sized) transaction-set. Assuming the lemma is false, there exist transactions $t_a \notin Y$ and $t_b \in Y$, such that $miss(t_a) < miss(t_b)$. Since Y satisfies the relaxation criteria, then $Y' = Y \cup \{t_a\} \setminus \{t_b\}$ also satisfies the relaxation criteria, with $|Y'| = |Y|$. Hence, Y' is also an optimal transaction-set, which contradicts our assumption. Therefore, the lemma holds. $\square$

This algorithm takes $O(n)$ time, because it iterates all the transactions. However, here two transactions are equivalent if they miss the same number of items, and hence, the transactions can be represented by E_{num}^X equivalence classes. So now, the algorithm needs to iterate only the equivalence classes of transactions missing the least number of items, and hence the complexity reduces to $O(|X|)$.

Case 3: The third case is when the item relaxation is considered ($\varepsilon_i \neq \infty$). Unlike the previous two cases, this case cannot be solved efficiently. Here, we show that finding the maximum support is an *NP-Complete* problem. In the proof, we use our *ILP* notation presented above.

Lemma 4.4. *Calculating $sup_{FT}(X) = \max \sum_{i=1}^n a_i$ in *ILP* is an *NP-Complete* problem.*

Proof. To prove *NP-Hardness* we take the well-known NP-complete problem of finding *Maximum Independent Set (MIS)* in a graph, and prove that *MIS* can be reduced into this problem. Given a graph G , the objective of *MIS* is to find a maximum set of vertices, such that there is no edge between any pair of them. The reduction is as follows. For every edge $e(u, v)$, add inequality $u + v \leq 1$ in *ILP*. This reduction is complete since the result of *ILP* will be the maximum set of variables (vertices) which are not connected. Since this case is a specialized problem of *ILP*, the *ILP* problem is at least as hard as *MIS*, implying that it is *NP-Hard*. Proving the completeness of its decision-problem ($\sum_{i=1}^n a_i \geq t$) for a threshold t is straight-forward. $\square$

4.3. CONSTANT RELAXATION ($\varepsilon_T^C, \varepsilon_I^C, \varepsilon_P^C \neq \infty$)

91

Since this problem is *NP-Complete*, we cannot expect to have a polynomial solution unless $P = NP$. While this theoretical result seems pessimistic, we show several techniques which can be useful to speed-up the calculation, to give a rather tight upper bound, or at least to prune the search space.

Lemma 4.5. *Let $\text{ext}(Y)$ be the superset of Y , such that the matrix $\text{ext}(Y) \times X$ is valid, and $|\text{ext}(Y)|$ is maximum. If there is a transaction t such that $|X \setminus I(t)| \leq 1$ and the matrix $(Y \cup \{t\}) \times X$ is valid, then $|\text{ext}(Y)| = |\text{ext}(Y \cup \{t\})|$. In other words, we can safely take any transactions that lack no more than one item in X .*

Proof. Relating with the ILP formulation, transactions lacking zero items do not appear in any inequality, and hence can safely be taken (otherwise the solution would not be maximal). Transactions lacking one item only appear in exactly one inequality, with coefficient of one. Let Y be the optimal transaction-set, that contains transaction t_1 lacking more than one item, say a and b . If there is no transaction $t_2 \notin T$ that lacks one item a or b , then our lemma holds since all transactions lacking one item are already taken. Otherwise, $Y \cup \{t_2\} \setminus \{t_1\}$ is also valid and has the same size as Y , and hence the lemma still holds. $\square$

This lemma solves the problem when $\varepsilon_t^c \leq 1$. In this case, we can simply iterate each of those transactions, and take it if the resulting pattern doesn't violate the relaxation criteria. Even when $\varepsilon_t^c > 1$, we can take all transactions missing no more than one item. This technique is very effective, as shown below. By taking a transaction containing an itemset that misses one item, say, item a , the number of allowed transactions missing this item (from the item relaxation) and the entire pattern (from the pattern relaxation) are reduced by one. Therefore, after taking ε_p transactions that miss one item, or ε_i transactions that miss each item, no more errors can be tolerated. Since ε_i and ε_p are in practice usually set to small values, there is a high probability that the error tolerance is exhausted, or almost exhausted, by considering only itemsets missing one item. This case often happens especially when the itemset is small.

With these optimizations, our algorithm becomes reasonably efficient in the real-world data-sets.

Upper Bound of FT-Support

Previously, we have shown that calculating $\text{sup}_{FT}(X)$ is an NP-Complete problem, which does not admit polynomial solutions. Here, we show several functions which

never underestimate the sup_{FT} , and can be calculated efficiently. Such a function helps to minimize the number of sup_{FT} calculation. That is, if the upper bound of an itemset is already less than the minimum support threshold, then its FT-support is definitely less than the minimum support. Therefore, we do not need to calculate the FT-support.

Our first bound is derived from following lemma.

Lemma 4.6. *Let Y be a maximal supporting transaction-set of itemset X with $\varepsilon_i^c = e$, and Y' be a maximal supporting transaction-set of X with $\varepsilon_i^c = e + 1$ and $Y \subseteq Y'$. Then, $|Y'| \leq |Y| + |X|$.*

Proof. Since Y is maximal, then adding any one transaction will cause at least one item of X to lack at least $e + 1$ transactions. Assuming $|Y'| > |Y| + |X|$, by pigeonhole principle, there is at least one item that lacks > 1 transactions in the additional transaction-set $(Y' \setminus Y)$. This contradicts our assumption, since the increase of ε_i^c is only 1. $\square$

Based on Lemma 4.6, the FT-support of itemset X is upper bounded by $sup(X) + \varepsilon_i |X|$.

As mentioned in the previous subsection, the only relaxation criteria that makes calculating FT-support difficult is the item relaxation. In this case, we can derive our second upper bound, by calculating the FT-support while ignoring the item relaxation. In other words, we assume that $\varepsilon_i^c = \infty$.

A tighter, but more expensive to calculate, upper bound can be derived by allowing the ILP system to have non-integer results (general LP). General LP can be solved in *expected polynomial* time using *simplex algorithm* [43], while ILP is NP-Hard.

The pseudo-code to calculate the support of an itemset (or any number less than the minimum support threshold if it is not frequent) is presented in Algorithm 4.1.

4.4 Proportional Relaxation ($\varepsilon_t, \varepsilon_i, \varepsilon_p < 1$)

This section presents the algorithm to enumerate FTFIs when faults are proportionally tolerated. As before, we discuss the two modules, viz., the candidate generation and the interestingness computation modules.

4.4. PROPORTIONAL RELAXATION ($\varepsilon_T, \varepsilon_I, \varepsilon_P < 1$)

93

Algorithm 4.1: Determining the FT-support of an itemset

Input: itemset X

- 1 sup = support by taking all itemsets missing at most 1 item (Lemma 4.5);
- 2 **if** *error tolerance is exhausted* **then return** sup ;
- 3 ub_1 = upper bound calculated based on Lemma 4.6;
- 4 **if** $ub_1 < \sigma$ **then return** 0;
- 5 ub_2 = FT-support assuming $\varepsilon_i = \infty$ (Lemma 4.3);
- 6 **if** $ub_2 < \sigma$ **then return** 0;
- 7 ub_3 = upper bound calculated using simplex algorithm;
- 8 **if** $ub_3 < \sigma$ **then return** 0;
- 9 sup = exact FT-support, calculated by solving the linear program;
- 10 **return** sup ;

Relaxation	Anti-monotonic	
	Const.	Prop.
Transaction	AM	Not AM
Item	AM	AM
Pattern	AM	Not AM

Table 4.1: Impact of relaxation criteria on the anti-monotone property of FT-support.

4.4.1 Candidate Generation

Monotonicity Property

We first analyze the characteristics of sup_{FT} with individual relaxation criterion. With proportional pattern relaxation, sup_{FT} is weak anti-monotone [85, 105], and with proportional transaction relaxation, sup_{FT} is not anti-monotone [105, 52, 23]. These findings are summarized in Table 4.1. We next show that sup_{FT} with only proportional item relaxation is anti-monotone.

Lemma 4.7. *When only item relaxation is considered, sup_{FT} is anti-monotone.*

Proof. Consider $P = Y \times X$ satisfying the item relaxation. In this case, all items in X are contained by at least $\varepsilon_i |Y|$ transactions, and $sup_{FT}(X) = |Y|$. Since each item is independent of the other, then $\forall Z \subseteq X, sup_{FT}(Z) \geq |Y|$. $\square$

When several relaxation criteria are combined, the anti-monotonic property of FT-support can be derived as follows.

Lemma 4.8. *When either transaction relaxation or pattern relaxation (or both) is considered, sup_{FT} is not anti-monotone.*

Proof. Suppose $\text{sup}_{FT}(X) < \xi$. That is, there is no $P = Y \times X$, with $|Y| \geq \xi$, which satisfies all relaxation criteria. Now, assume that there exists $P_1 = Y_1 \times X$, where $|Y_1| \geq \xi$, such that $Y_1 \times X$ satisfies item relaxation. In the pessimistic case, it is possible to extend X into X' , such that Y_1 contains all transactions in $X' \setminus X$, and $|X'|$ is large enough such that $Y_1 \times X'$ satisfies all relaxation criteria. Therefore, $\text{sup}_{FT}(X') \geq \xi$, implying that it is not anti-monotone. $\square$

These lemmas showed that sup_{FT} is in general not anti-monotone.

Upper Bound Function

As mentioned earlier in Section 2.2.5, one way to mine interesting itemsets when the measure does not satisfy the anti-monotone property is by deriving an anti-monotone upper bound function $ub(X)$ which never underestimates $\mu(X) = \text{sup}_{FT}(X)$. Lemma 4.8 states that we cannot construct an anti-monotone function by using either proportional transaction or pattern relaxations. So, we consider a function $ub_o(X) = \text{sup}_{FT}(X, \varepsilon_t = 1, \varepsilon_i, \varepsilon_p = 1)$. It is clear that $ub_o(X)$ never underestimates $\text{sup}_{FT}(X)$ since it uses *lesser* constraints, and it is anti-monotone. Hence, ub_o can be used as a bounding-function. This function is indeed the best (strictest) possible, assuming we only have knowledge of itemset X , i.e., we ignore all items not within X .

Lemma 4.9. *By considering only items $x \in X$, $ub_o(X) = \text{sup}_{FT}(X, \varepsilon_t = 1, \varepsilon_i, \varepsilon_p = 1)$ is the tightest upper bound for $\text{sup}_{FT}(X)$ that satisfies the anti-monotone property.*

Proof. Since we cannot use either transaction or pattern relaxation (Lemma 4.8), we can only use item relaxation. $\square$

However, the computation of $ub_o(X)$ is an NP-Complete problem, as in the case with constant relaxation. A natural approach to boost the efficiency is to design an *approximate* function that may be computed efficiently. One simple way to achieve this is by *loosening* item relaxation with pattern relaxation. It is clear that patterns satisfying item relaxation ($\varepsilon_i = e$) will also satisfy pattern relaxation with the same tolerance ($\varepsilon_p = e$). A similar technique was used in [105] to loosen transaction relaxation with pattern relaxation. Mathematically, we define $ub_a(X) = \text{sup}_{FT}(X, \varepsilon_t = 1, \varepsilon_i = 1, \varepsilon_p = \varepsilon_i)$.

Figure 4.2: Approximating and tightening the bounds for FT-support.

Example 4.7. Let $\varepsilon_t = 0.3$, $\varepsilon_i = 0.2$, and $\varepsilon_p = 0.1$. $ub_a(X)$ is the FT-support of itemset X with parameters $\varepsilon_t = 1$, $\varepsilon_i = 1$, and $\varepsilon_p = 0.2$.

Despite being not anti-monotone, the function $ub_a(X)$ is a valid bounding function since it never underestimates a valid bounding function $ub_o(X)$. In other words, we can safely *assume* $ub_a(X)$ as anti-monotone, i.e., we can use candidate generation algorithms for measures satisfying the anti-monotone property. Moreover, $ub_a(X)$ can be calculated efficiently with time complexity $O(|X|)$ having the partial supports of the itemset. Interestingly, this weaker bound is the best possible if we know only the partial supports of itemsets.

Lemma 4.10. *Having only the partial supports, $ub_a(X)$ is an optimal upper bound for $ub_o(X)$.*

Proof. Based on the proof of Lemma 4.9, we can impose only item relaxation to form a bounding function. Since we know only the number of items contained by each transaction, the worst case is to assume that the *missing* items are distributed uniformly among the transactions. In this case, all items are missed by the same number of transactions, hence item relaxation is equivalent to pattern relaxation. $\square$

Liu et al. proposed another upper bound in [52]. Analyzing the derivation, we find that their approach calculates the FT-support assuming all transactions either contain all items or contain $k + 1$ items, where $k = \lfloor \varepsilon_t |X| \rfloor$, and the missing items are uniformly distributed. This bound is clearly weaker (higher) than ub_a .

A Tighter Bound Based on the Largest Desired FTFIs

Previously, we showed that $ub_o(X)$ is optimal, having only the knowledge of itemset X . Here, we show that if we know M - the *largest desired* size of FTFIs, then we

can further tighten the bound, as illustrated in Figure 4.2. When the largest desired FTFI is of size $|X_2|$, the bound can be tightened (shown by dashed lines in the figure), and when we don't need FTFIs bigger than $|X_1|$, $X_1 \subset X_2$, the tightening can be potentially stronger. Note that our approach guarantees completeness of the results if the size of the true largest FTFI is no larger than M . In practice, even if we want to mine all FTFIs, we can safely set M as a large value (30 – 40). We will also show how to reduce the M iteratively during the enumeration process.

The idea of the further pruning is to exploit the transaction and pattern relaxations, which were omitted in $ub_o(X)$ and $ub_a(X)$. We first discuss how to handle transaction relaxation. Since the largest FTFI is of size M , then the number of errors allowed in *all* FTFIs is at most $\lfloor \varepsilon_t M \rfloor$. Therefore, by setting the transaction relaxation as a constant value $\varepsilon_t^c = \lfloor \varepsilon_t M \rfloor$, we will never underestimate the support, which constitutes a valid bounding function.

Next, we discuss how to get rid of proportional pattern relaxation. The idea is still the same as in transaction relaxation, by calculating the maximum (constant) allowed errors of all FTFIs. However, the absolute value of pattern relaxation also depends upon the size of the transaction-set. The easiest way to achieve this is by using another upper bound for the maximum transaction-set size, e.g., $|T|$. Hence, the whole matrix is allowed to have up to $\varepsilon_p^c = \varepsilon_p |T| M$ errors. Despite being valid, this bound is very loose. A tighter bound is derived as follows. For a pattern $Y \times X$, the maximum total number of errors is restricted to be at most $\varepsilon_p |X| |Y|$. Since $|X| \leq M$, the total number of allowed errors in all FTFIs is less than $(\varepsilon_p M) |Y|$. Let $\varepsilon_p(X)$ be the pattern relaxation for an itemset X , then the number of errors allowed per pattern is at most $\varepsilon_p(X) |X| |Y| = \varepsilon_p M |Y|$. Hence, we have $\varepsilon_p(X) = \varepsilon_p \frac{M}{|X|}$.

We can then define $ub'_o(X) = \sup_{FT}(X, \varepsilon_t^c = \varepsilon_t', \varepsilon_i, \varepsilon_p = \varepsilon_p(X))$, where $\varepsilon_t' = \lfloor \varepsilon_t M \rfloor$ and $\varepsilon_p(X) = \varepsilon_p \frac{M}{|X|}$. As before, we can approximate item relaxation with pattern relaxation for faster computation. In this case, ε_i becomes 1 (unused), and ε_p becomes the minimum of ε_i and ε_p' .

Example 4.8. Let $\varepsilon_t = 0.3$, $\varepsilon_i = 0.2$, $\varepsilon_p = 0.1$, $m = 8$ and $|X| = 5$. Here, we have $\varepsilon_t' = \lfloor 0.3 \cdot 8 \rfloor = 2$, and $\varepsilon_p(X) = 0.1 \cdot \frac{8}{5} = 0.16$. In this case, $ub'_o(X)$ is the FT-support of itemset X with parameters $\varepsilon_t^c = 2$, $\varepsilon_i = 0.2$, and $\varepsilon_p = 0.16$. To approximate $ub'_o(X)$, we use $\varepsilon_i = 1$ and $\varepsilon_p = \min(0.2, 0.16)$.

Further Pruning during Enumeration

Further pruning for enumerating FTFIs can be performed by relating the relaxation criteria and the database shrinking procedure (c.f., Section 2.3). Firstly, we show how to reduce the maximum size of itemsets M , which has been shown useful for tighten the bound. Let the current itemset be X and the possible extension be $Tail$, it is obvious that the largest itemset is bounded by $|X| + |Tail|$. Therefore, in each step whenever we reduce the $Tail$, we can update M into $\min(M, |X| + |Tail|)$. The more effective, but less efficient, way to tighten M is by analyzing each transaction. For an itemset to be qualified as an FTFI, there should be at least σ transactions supporting it. Let l be the length of the σ -th longest transaction (after intersection with $X \cup Tail$), we can set M as $\min(M, \lfloor \frac{l}{1-\varepsilon_t} \rfloor)$.

Example 4.9. Suppose the database contains 10 transactions, each containing 1, 2, ..., 10 items (after projection with X and $Tail$), and $\varepsilon_t = 0.2$. When $\sigma = 4$, then the 4-th longest transaction contains 7 items, and hence we can set M as $\lfloor 7/0.8 \rfloor = 8$.

We can also remove several transactions, which are impossible to support the itemset. One example is that if the length of the current itemset $|X| = 2$, with $\varepsilon_t < 0.5$, then any transactions containing less than 2 items will never support this itemset. To be exact, if a transaction t contains a items in X and k items in $Tail$, then this transaction can only support itemset of size $m = \lfloor \frac{a+k}{1-\varepsilon} \rfloor$. And hence, if $|X| > m$, then we can safely remove this transaction.

Example 4.10. Let the current itemset be $X = abc$ and the remaining items be $Tail = defg$. If a transaction t contains itemset adz , with $\varepsilon_t = 0.3$, this transaction can only support itemsets of size $\lfloor 2/0.7 \rfloor = 2$. As $|X| > 2$, then we can safely remove this transaction.

By removing transactions, we can further tighten the bound as the values of ub_o and ub_a will also reduce, and consequently several items can also be removed.

This transaction removal procedure is similar to the *global pruning* technique in [52]. However, global pruning uses Apriori candidate generation, which, as we have analyzed (c.f., Section 2.2.2), is not very effective for shrinking the database. We can design a better pruning scheme because of our thorough and systematic analysis of the database shrinking process and the candidate generation procedure.

4.4.2 Calculating the Fault-Tolerant Support

This section discusses how to handle the difficult problem of interestingness computation with item relaxation. In previous section, we show that with constant relaxation, calculating of FT-support is easy when item relaxation is not imposed, but suddenly becomes NP-Complete otherwise. When item relaxation is imposed, we propose to transform the problem into integer linear programming.

When item relaxation is defined as proportional, the problem can also be modeled as an ILP, by adding an extra factor, ε_i , in the coefficients. However, this change affects the efficiency significantly, since no pre-processing can be done. This is because we do not know how many transactions can we take for each itemset missing one item. Here, we propose to transform the constraint into a constant value, so that we can use the previous technique.

For a matrix $Y \times X$ to be valid, each transaction in Y lacks at most $\varepsilon_t |X|$ items, each item in X is lacked by at most $\varepsilon_i |Y|$ transactions, and the entire matrix must lack at most $\varepsilon_p |X| |Y|$ items. We define $h(s)$ as the FT-support of X , where the tolerance is a constant value, set as if the number of transactions in the pattern is s . Or mathematically, $h(s) = \sup_{FT}(X, \varepsilon_t^c = \varepsilon_t |X|, \varepsilon_i^c = \varepsilon_i s, \varepsilon_p^c = \varepsilon_p |X| s)$. Let s be *achievable* if there is a transaction-set Y such that the pattern $Y \times X$ valid, then we have following lemma.

Lemma 4.11. s is achievable if and only if $s \leq h(s)$.

Proof. First, observe that $h(s)$ is monotone to s , since greater s allows more errors, implying greater $h(s)$. Now let the absolute number of errors in a pattern $P = Y \times X$, when $|Y| = s$ (in any criteria) be e . If $h(s) < s$, it means that the size of the largest transaction-set that allows e errors is lesser than s . Hence, s is not *achievable*. When $h(s) \geq s$, we can use the respective transaction to form $h(s)$ as the transaction-set Y . Hence, s is *achievable*. $\square$

Lemma 4.12. $\sup_{FT}(X) = \max\{s | s = h(s)\}$.

Proof. If $h(s) > s$, there is always $s_o \geq h(s)$, such that $s_o = h(s_o)$. Otherwise, we can set s_2 as $h(s)$, and recalculate $h(s_2)$. This cannot be done indefinitely since the domain is finite (they represent support of itemset). Therefore, the largest *achievable* s is always when $s = h(s)$. $\square$

One intuitive approach to find the largest s where $s = h(s)$, is to perform binary search on s . However, even though $h(s)$ is monotone to s , $s - h(s)$ is neither monotone

4.4. PROPORTIONAL RELAXATION ($\varepsilon_T, \varepsilon_I, \varepsilon_P < 1$)

99

Algorithm 4.2: Iterative method to compute FT-support with proportional relaxation.

Input: itemset X , ε_t , ε_i , ε_p

Output: FT-support

```

1  $s \leftarrow UB$  // set  $s$  as the upper bound
2 while true do
3    $s' \leftarrow \text{sup}_{FT}(X, \varepsilon_t^c = \varepsilon_t |X|, \varepsilon_i^c = \varepsilon_i s, \varepsilon_p^c = \varepsilon_p |X| s);$ 
4   if  $s = s'$  then
5     return  $s$ ;
6   else
7      $s' \leftarrow s$ ;

```

nor anti-monotone, hence ruling out binary search. Here we propose to use an iterative method as presented in Algorithm 4.2.

First, we initialize s as the upper bound of the FT-support (line 1). One easy and reasonably good approach is by relaxing item relaxation with pattern relaxation, hence $UB = \text{sup}_{FT}(X, \varepsilon_t, \varepsilon_i = 1, \min(\varepsilon_p, \varepsilon_i))$. Having obtained the upper bound of s , we iteratively verify whether s is *achievable* (line 4). If it is, then we have obtained the result, otherwise, we lower down the estimation. The correctness of this algorithm is proved as follows.

Lemma 4.13. *The iterative method (Algorithm 4.2) calculates the FT-support of an itemset.*

Proof. Since s is the upper bound, if $s' = s$, then s is the largest value which is *achievable*, implying s is the FT-support (line 5). Otherwise, if $s' < s$, then the FT-support must be less than or equal to s' , since s' is monotone to s . $\square$

Next, we analyze the number of iterations with respect to item relaxation. Let the initial $s = a$ and $h(s) = b$. On average, (constant) item relaxation is reduced by 1 whenever s is reduced by $1/\varepsilon_i$. Also, whenever constant item relaxation is reduced by 1, $h(s)$ is reduced by at most $\Delta = \lceil (1 - \varepsilon_t)|X| \rceil$, by removing one transaction per item. Let $\Delta/(1/\varepsilon_i) = \Delta \varepsilon_i$ be denoted as α . Since the amount of first reduction of s is $a - b = d$, then the second reduction would be at most $d \alpha$, and the following reductions would geometrically progress as $d \alpha^2$, $d \alpha^3$, etc. Since the algorithm terminates when the reduction equals 0, the approximated upper bound of the number of iterations (ignoring floor and ceiling) is given as $O(-\log_\alpha(d))$, where $d \leq UB - \text{sup}_{FT}(X)$.

For example, when $|X| = 5$, $d = 1,000$, $\varepsilon_t = 1$ and $\varepsilon_i = 0.1$, we have $\alpha = 0.5$, implying that the number of iterations would be approximately $-\log_{0.5}(1000) \approx 10$.

Relaxation	Anti-monotonic		Interestingness Computation	
	Const.	Prop.	Const.	Prop.
Transaction	AM	Not AM	$O(X)$	$O(X)$
Item	AM	AM	NPC	NPC
Pattern	AM	Not AM	$O(X)$	$O(X)$

Table 4.2: Impact of relaxation criteria on the anti-monotone property of FT-support, and on the complexity of interestingness computation.

In our experiments (Section 4.5), we show that in practice the number of iterations is very small, and more importantly, the algorithm is efficient.

Prior to designing the algorithm, we analyze the impact of each constraint on the complexity of the interestingness computation procedure. This result is summarized in Table 4.2. When only transaction relaxation is considered, the FT-support is simply the number of all transactions in the pattern that satisfies transaction relaxation. When only pattern relaxation is considered, we can use a greedy algorithm, by repeatedly taking transactions with least number of errors, until the constraint is violated [105, 71, 85]. Having the partial supports, these two functions can be calculated in $O(|X|)$ iterations. When item relaxation is imposed, the partial supports are not sufficient since we need to know the errors contained by specific items. Moreover, calculating the FT-support when item relaxation is imposed is an NP-complete problem [71].

Comparing the algorithm with the optimal bound and that with approximate bound, we see that the optimal bound is obviously tighter, and hence the algorithm results in fewer candidates. However, the amount of time taken to verify each candidate is significantly larger. Therefore, we evaluate these two techniques empirically to find out which one is more efficient in which environment-settings.

4.5 Experimental Results

We used three real datasets, viz., BMS-Webview-1, Mushroom, and Chess for evaluating the efficiency of the algorithms, and also the sensitivity with respect to the tolerance parameters. Details of these datasets are available in Appendix A. The minimum supports provided in the charts are stated as the *relative support*, which is the proportion of the absolute support and the number of transactions.

4.5. EXPERIMENTAL RESULTS

101

Figure 4.3: Enumerating fault-tolerant frequent itemsets with constant transaction relaxation. The value of σ^i is fixed to 0.5σ in BMS-Webview-1 and Mushroom datasets, and 0.8σ in Chess dataset.

4.5.1 Constant Transaction Relaxation

Firstly, we evaluate the efficiency of the algorithms for enumerating FTFI with constant transaction relaxation [69]. As mentioned earlier, partial support can be used for deriving the fault-tolerant support of itemsets [69]. Here we evaluate the Sup-RR, QIE, BIAS algorithm (Incremental) [71] and VB-FT-Mine (Trans-RR) [44], as shown in Figure 4.3. We do not present the results of FT-Apriori [69] as it is too slow, and it would cramp the plots if included.

We can also observe that the difference between Sup-RR and QIE is not as significant as before. This behavior is as expected, because of the restriction for the number of missing items. We can also see in Figures (c) to (e) that the difference becomes more significant when δ increases, especially in dense datasets.

4.5.2 Constant Relaxation

Interestingness Computation

Firstly, we evaluate the effect of pre-processing, by greedily taking itemsets lacking one item (or none) prior to solving the ILP. Figure 4.4 compares the efficiency of the

Figure 4.4: Impact of pre-processing on the interestingness computation module.

algorithm with pre-processing (denoted as *pp* in the plots) and without pre-processing (denoted as *no pp*). For evaluation, we calculate the time for interestingness computation, for all frequent itemsets (not fault-tolerant) with the given minimum support threshold. The value of ϵ_p^c is set to twice of ϵ_i^c , and ϵ_t^c is set to infinity.

We see that pre-processing is always faster, especially in the sparse datasets. This is because there are a lot of transactions missing one item in this case, and hence, the FT-support can be determined solely by the pre-processing step. We may also notice that the relation between error tolerance and efficiency is unclear, i.e., higher tolerance is faster in BMS-Webview-1 dataset, but slower in Mushroom dataset. One possible explanation is because of the instability of the linear programming module, that involve many case-by-case optimizations. Nevertheless, we can clearly observe that pre-processing is always beneficial, therefore, we always use it for the following procedures.

Candidate Generation

The candidate generation procedure with constant relaxation is not very exciting, as the function is anti-monotone. Therefore, the time taken should be equivalent as those for calculating the subset supports (Section 2.6.1), plus the interestingness computation procedure which has just been evaluated above.

To avoid repetitions, we do not evaluate this module, and will examine the complete package for the proportional relaxation.

4.5.3 Proportional Relaxation

Lastly, we evaluate the efficiency of algorithms when the error tolerance is defined proportional to the patterns' size. As before, we split the analysis for the interestingness

4.5. EXPERIMENTAL RESULTS

103

Figure 4.5: Efficiency of ILP with proportional relaxation and Iterative + constant relaxation.

Dataset	Method	N_{cb}	N_{cf}	FTFI	T_{am} (s)	T_{sc} (s)	T_{tot} (s)
BMS-Webview-1 $\sigma = 0.001$	Optimal	165,336	84,264	8,633	395.97	16.97	415.12
	App.	173,875	94,958	8,633	0.06	5.81	7.30
Mushroom $\sigma = 0.24$	Optimal	32,799	32,135	20,412	129.97	76.89	208.44
	App.	34,902	34,247	20,412	0.02	80.77	82.57
Chess $\sigma = 0.7$	Optimal	172,822	172,095	117,062	405.05	8.58	426.93
	App.	175,889	175,151	117,062	0.06	8.99	21.90

Table 4.3: Detailed snapshot of the efficiency of the optimal and approximate bound for FT-support.

computation and the candidate generation modules independently.

Interestingness Computation

Next, we compare the efficiency of interestingness computation with proportional relaxation, between directly converting the problem to ILP and using the iterative approach which reduces to constant tolerance (Algorithm 4.2). The former is denoted as *ILP-Prop* and the latter as *Iterative*. Here, ϵ_p is fixed as half of ϵ_i , and $\epsilon_t = 1$, and the results are shown in Figure 4.5.

We again see a consistent pattern, with the numerical approach always running faster. This is perhaps due to the small number of iterations. In the Chess dataset, the maximum number of iterations is 5, with average of slightly above 1. In the BMS dataset, the maximum number of iterations reaches 30, but the average is still lesser than 2.

Figure 4.6: Efficiency of the optimal and approximate bound of FT-support.

Bound Checking

This section analyzes the efficiency and effectiveness of the bound-checking procedure. As mentioned in Section 4.4.1, we can either calculate the optimal bound or perform approximation. The results are plotted in Figure 4.6, and a detailed snapshot for one particular minimum support is presented in Table 4.3. Columns N_{cb} and N_{cf} refers to the number of candidate bound (itemsets whose upper bound is checked) and candidate frequent (itemsets whose FT-support is checked) respectively. Columns T_{am} , T_{sc} , and T_{tot} refers to the time taken to calculate the bound, the FT-support, and the total running time respectively. T_{tot} is always greater than $T_{am} + T_{sc}$ as there are other house keeping operations, for example for extending and calculating the exact support of itemsets. It is clear from the table that by approximating the bound, the number of candidates only increases a little, but the time for bound-checking decreases significantly. Hence, it is beneficial to use the approximate bound.

Regarding the effectiveness of the bound, we can see that the number of itemsets within borders is around 9.8 times, 1.6 times, and 1.5 times the numbers of FT-FIs in BMS-Webview-1, Mushroom, and Chess datasets respectively. While ideally the number of candidates frequent should be similar to the number of FTFIs, these numbers are, in our opinion, quite reasonable.

Next, we analyze the impact of M , the desired maximum size of FTFIs, as shown in Figure 4.7. Sizes of the actual largest *FTFIs* for Retail, Mushroom, and Chess datasets are 6, 13, and 14 respectively, and hence we mine complete results in all cases. We observe that the efficiency is insensitive to M .

4.5. EXPERIMENTAL RESULTS

105

Figure 4.7: Impact of the maximum desired FTFI.

Figure 4.8: Sensitivity with respect to minimum support and error tolerances.

Sensitivity w.r.t. parameters

Here we analyze the impact of each relaxation criteria, with respect to the number of FTFIs and time taken by the mining algorithm. We also evaluate the efficiency for enumerating maximal FTFIs, which we mine using depth-first traversal (incremental algorithm). To avoid repetition, we only demonstrate using the Mushroom dataset, where the difference between all and maximal itemsets is most noticeable.

Figure 4.8 shows the scalability with respect to minimum support and error tolerances. Here, we can see a consistent pattern that the number of FTFIs is quite insensitive to ε_i and ε_t . This is a good news for the analysts since we can simply pick any *natural* value of ε_i and ε_t , and need only to tune σ and ε_p . Nonetheless, the difference compared to mining without error tolerance is quite significant. This shows that FTFIs can capture interesting knowledge which remain undiscovered using strict frequent itemsets.

4.6 Concluding Remarks

In this chapter, we presented algorithms for enumerating fault-tolerant frequent itemsets (FTFI). Fault-tolerant support is a relatively new interestingness measure of itemsets, which is proven useful in many applications. However, enumerating FTFIs, is difficult as FT-support lacks the anti-monotone property, and it is also difficult to calculate.

In this work, we considered three variants of FT-support. In the first variant, we consider only the constant transaction relaxation. We showed that in this case, FT-support is identical to the partial supports of itemsets, for which efficient mining algorithms exist. In the second variant, we considered the case with any constant relaxation. This problem is difficult, as calculating the FT-support of an itemset is not trivial. In fact, we proved that calculating FT-support (with any constant relaxation) is an NP-complete problem. We solved this problem by modeling it as an integer linear programming problem, which can be solved reasonably efficiently. In the third and the most general variant, we also considered proportional relaxation. In this case, calculating the FT-support becomes even more difficult, and the FT-support becomes non-anti-monotone. We tackled the former problem by mapping it into a sequence of FT-support calculations with constant relaxation, and the latter problem by analyzing the upper bound of FT-support, and aggressively shrinking the database during the enumeration process.

Through extensive empirical evaluations in real datasets, we showed that our algorithms are efficient and practical.

Chapter 5

Duality between Mining Frequent Itemsets and Fault-Tolerant Frequent Itemsets

Instead of developing an enumeration framework as in previous chapters, we can alternatively enumerate interesting itemsets by reducing the problem to the famous and well-studied frequent itemset mining problem. Formally, given a database D , our goal is to create another database D' such that enumerating frequent itemsets in D' produces equivalent results as enumerating interesting itemsets in D .

This chapter presents a solution of this duality problem for enumerating FT-frequent itemsets (FTFI), with constant transaction relaxation (Section 4.2). FTFI is one instance of interesting itemsets, which allows a transaction that contains part of an itemset, to be considered as its supporting transaction. The objective of FTFI is to obtain the generalized knowledge of the itemset. Detailed discussions about FTFI are presented in Chapter 4. We tackle this variant because, similar to support, FT-support is also anti-monotone, and each transaction is independent. These properties allow us to transform each transaction individually.

For this problem, we manage to design the optimal solution for *transaction independent transformation*, i.e., the resulting transformed database is the smallest possible, if each transaction is transformed independently.

5.1 Preliminaries

5.1.1 Motivation

In previous chapters, we have discussed the framework to enumerate interesting itemsets. However, having an algorithm, producing an efficient implementation is still a challenging task, which include many sub-tasks including the memory management, choosing the appropriate data structure, and parallelism. One concrete example is the existence of plenty algorithmic innovations in the past decade for enumerating frequent itemsets [37, 92, 16], including a workshop devoted to finding the best implementations [11]. These facts indicate that sustained research efforts are required to generate implementations that are practically useful under various settings. As an addition, there are also other bunch of issues including enumeration in streaming/dynamic environment [30] and to ensure privacy of the database [6], which are still active research topics.

With such tremendous efforts in optimizing the algorithms and implementations for enumerating frequent itemsets, it would ideal if we can simply *reuse* those results for enumerating other notions of interesting itemsets. This chapter realizes such a need by formulating the duality problem between enumerating frequent itemsets and interesting itemsets. Informally, given a transactional database D , find a transformed database D' such that enumerating frequent itemsets (FI) in D' produces results equivalent to enumerating interesting itemsets in D .

This chapter focuses on reducing the variant of mining fault-tolerant frequent itemsets with constant transaction relaxation, to the frequent itemset mining problem. For brevity, we denote the problem to enumerate frequent itemsets as FIM and the one to enumerate FTFI as FTFIM. Therefore, for the rest of this chapter, the term *duality problem* always refers to the duality between FIM and FTFIM, and not other instances of interesting itemset mining problem. Similarly, the discussions on the FTFIM problem, including the motivations and algorithms, are out-of-scope of this chapter, and are already presented in Chapter 4.

5.1.2 Contributions

Informally, an itemset is considered as an FTFI if its FT-support is no less than the minimum support threshold σ . The FT-support of an itemset X is calculated as the number of transactions that lacks at most ε items in X . An intuitive solution for the duality problem would transform each item by embedding the faults (ε). However,

this approach turns out to be infeasible because the number of transformed items explodes. As each item is transformed, we need to post-process the result to get the desired itemsets. It also turns out that the number of *raw* outputs are abundant, which explains the infeasibility.

We then propose our solution, by transforming each transaction into several *weighted* transactions, and keeping the items untouched. Our transformation algorithm produces a transformed database of size $O(m^\varepsilon)$ the original with constant time complexity per output. We also prove that the transformed database is the smallest possible, if we transform each transaction independently. Mining FTFIs in the transformed database is feasible, since the number of items in the transformed database is the same as that in the original database, and most frequent itemset mining algorithms are output linear [92].

5.1.3 Basic Definitions and Notations

As this section deals with two databases (original and transformed), the transactions and items of a database D is denoted by T^D and I^D respectively. Similarly, the support of an itemset X in database D is denoted as $\text{sup}^D(X)$. When the context is clear, the superscript D can be neglected. Multiple transactions that contain the same itemset can be treated as one *weighted transaction*, which is a transaction with an associated weight. A transaction that contains itemset X with weight k is denoted as a pair (k, X) .

To recall, FTFIM relaxes the notion of transaction supporting an itemset, by allowing each transaction to lack ε items. Following example illustrates the difference between the FIM and the FTFIM problem, which will be used throughout this chapter. For easier presentation,

Example 5.1. Consider the sample transactional database presented in Figure 5.1. Without fault-tolerance ($\varepsilon = 0$), the support of item a is 2, since item a is contained by two transactions, viz., transactions 1 and 4. Similarly, the support of itemset cd is 2 since both items are contained in 2 transactions, viz., transactions 2 and 4. Hence, with minimum support of 2, all frequent itemsets are: $\{a, b, c, d, cd\}$.

With $\varepsilon = 1$, the support of itemset bc is 4, since all transactions contain at least 1 item of bc . Similarly, the support of itemset abc is 2 from transactions 1 and 4. Hence, with minimum support of 2, all frequent itemsets are: $ab, ac, ad, bc, bd, cd, abc, abd, acd, bcd$. Note that itemsets with size no larger than ε are not interesting.

We denote the set of all k -subsets of X as $\|X\|_k$. For example, when $X = abcd$,

Tid	Itemset
1	a, b
2	c, d
3	b
4	a, c, d

(a) Table form

	a	b	c	d
1	■	■	□	□
2	□	□	■	■
3	□	■	□	□
4	■	□	■	■

(b) Matrix form

Figure 5.1: Sample transactional database as a running example (repeated).

$\|X\|_2 = \{ab, ac, ad, bc, bd, cd\}$. Let $\mathcal{FI}(D)$ be the set of FIs in database D , and $\mathcal{FTFI}(D)$ be the set of FTFIs in D , now the duality problem can be defined as follows.

Definition 5.1. [DUALITY]. Given a database D and the error tolerance ε , construct another database D' such that $\mathcal{FTFI}(D) = \mathcal{FI}(D')$.

5.1.4 Organization

This chapter is organized as follows. Firstly, we describe the intuitive but infeasible algorithm in Section 5.2. Next, we present our proposed algorithm in Section 7.4.2. Finally, Section 5.4 concludes with future research directions.

5.2 Intuitive (Infeasible) Approach

The natural approach to solve the duality problem is to analyze the definition of *support*. A transaction y supports an itemset X if it contains *all* items in X , and it FT-supports X if it lacks at most ε items of X . As y lacks at most ε items, if we take any $\varepsilon + 1$ items of X , then at least one of them must be present in y . This property is established by following lemma.

Lemma 5.1. *A transaction y FT-supports an itemset X with fault-tolerance ε , if and only if y contains at least one item for every $(\varepsilon + 1)$ -subset of X .*

Proof.

“if” part: Transaction y FT-supports itemset X means y lacks at most ε items of X . Hence, in every $(\varepsilon + 1)$ -subset of X , at least one item is contained in y .

5.2. INTUITIVE (INFEASIBLE) APPROACH

111

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>		<i>ab</i>	<i>ac</i>	<i>ad</i>	<i>bc</i>	<i>bd</i>	<i>cd</i>
1	■	■	□	□	→	■	■	■	■	■	□
2	□	□	■	■		□	■	■	■	■	■
3	□	■	□	□		■	□	□	■	■	□
4	■	□	■	■		■	■	■	■	■	■

Figure 5.2: Item transformation with $\varepsilon = 1$

“only if” part: By contraposition, transaction y does not FT-support itemset X means y lacks at least $\varepsilon + 1$ items of X , i.e., $|X \setminus I(y)| \geq \varepsilon + 1$. Hence, there exists some $(\varepsilon + 1)$ -subset of X , no item of which is contained in transaction y . $\square$

With this lemma, we can reduce the FTFIM problem to FIM by transforming the original database D into another database D' whose items are the $(\varepsilon + 1)$ -subsets of the set of items in D . So, a transaction y contains an item x in D' , if in D it contains at least one item of x . As we transform each item of the original database, we term this transformation algorithm as *item transformation*.

Example 5.2. Figure 5.2 shows the transformation of our running example with $\varepsilon = 1$. The set of items in database D' is $\|I\|_2$. Each transaction contains an item in D' , which is a 2-itemset in D , if it contains at least one of them in D . For example, transaction 1 contains item $\{bc\}$ in D' since it contains item b in D , but it doesn't contain item $\{cd\}$ in D' since it contains neither item c nor d in D .

Based on Lemma 5.1, the FT-support of itemset X in D is equal to the support of itemset $\|X\|_{\varepsilon+1}$ in D' . For example, the FT-support of itemset abc in D is 2 (from transaction-set $\{1,4\}$), which is equal to the support of itemset $\{ab, ac, bc\}$ in D' .

5.2.1 Properties of the Transformed Database

Let the number of total items in D be m and the number of items contained by a transaction y be k . Then the number of items contained by transaction y in D' is given as:

$$\binom{m}{\varepsilon + 1} - \binom{m - k}{\varepsilon + 1} \quad (5.1)$$

Proof. First, the total number of items in D' is $\binom{m}{\varepsilon+1}$. Next, transaction y does not contain item x in D' if it does not contain any item in x in D (item x in D' represents

an itemset in D). Hence, the total number of items not contained in D' equals to $\binom{m-k}{\varepsilon+1}$. $\square$

Therefore, the size of the transformed database grows polynomially with respect to ε from the original database.

Next, we analyze the density of the transformed database. Since each transaction is transformed independently, we can calculate the density independently as well. In normal FIM, the density of a transaction is given as $\frac{k}{m} = 1 - \binom{m-k}{m}$. For the transformed database, based on Equation (5.1), the density of a transaction is given as $1 - \binom{m-k}{\varepsilon+1} / \binom{m}{\varepsilon+1} \approx 1 - \left(\frac{m-k}{m}\right)^{\varepsilon+1}$.

5.2.2 Issues of the Item Transformation

We have analyzed the correctness and the properties of the transformation algorithm. However, some issues restrict the applicability of item transformation.

The first issue is regarding the number of the frequent itemsets. An itemset X in the original database is represented by itemsets $\|X\|_{\varepsilon+1}$ in the transformed database. Therefore, if the size of original itemset is k , the number of the transformed itemsets becomes $\binom{k}{\varepsilon+1}$, which is large. For example, when the original itemset is of size 10 and $\varepsilon = 1$, the corresponding number of itemsets in the transformed database will be $\binom{10}{2} = 45$. Consequently, there are at least $2^{45} \approx 3.5 \times 10^{13}$ frequent itemsets in the transformed database.

The second and related issue appears with regards to meaningless itemsets. Based on the transformation procedure, itemset X is translated into itemset $\|X\|_{\varepsilon+1}$ in the transformed database. In other words, only itemsets of the form $\|X\|_{\varepsilon+1}$ represent an itemset in the original itemset. For example, when $\varepsilon = 1$, itemset $\{ab, ac, bc\}$ is meaningful since it is in the form of $\|abc\|_2$, but itemset $\{ab, ac\}$ is not. This issue explains the abundant number of frequent itemsets in the transformed database.

5.3 Proposed Approach

As the intuitive approach is not feasible, we use another strategy for solving the duality problem. That is, we do not analyze the support, but instead analyze the itemsets. This section presents our proposed approach, including the thinking process for solving the simpler problem, i.e., the inverse frequent itemset mining problem [19].

Algorithm 5.1: Naïve Algorithm for Inverse Frequent Itemset Mining

Input: $\mathcal{FTFI}(D)$
Output: Transformed database D'

```

1  $D' \leftarrow \emptyset;$ 
2 for  $(X, s) \in \mathcal{FTFI}(D)$ , in size descending order do
3    $w \leftarrow s - \text{sup}^{D'}(X);$ 
4   if  $w \neq 0$  then
5      $D' \leftarrow D' \cup \{(w, X)\};$ 
6 return  $D'$ 

```

5.3.1 Inverse Frequent Itemset Mining

As mentioned in the problem definition, our goal is to *transform* the database D into another database D' , such that $\mathcal{FTFI}(D) = \mathcal{FI}(D')$.

First, we relate this problem to the inverse frequent itemset mining (IFM) problem [19], which has been studied quite extensively. The problem of IFM is to reverse the mining process, i.e., given the set of frequent itemsets $\mathcal{F}$, we need to find the database D' such that $\mathcal{FI}(D') = \mathcal{F}$. Relating with the duality problem, our transformed database D' can be attained using IFM algorithm, if we set $\mathcal{F} = \mathcal{FTFI}(D)$. Note that this is only a preliminary step for the algorithm design, since in practice we cannot have $\mathcal{FTFI}(D)$ as input.

The IFM problem is in general NP-Hard. However, as our variant allows negative-weighted transactions, this problem can be solved efficiently as in Algorithm 5.1. The term (X, s) in Line 2 represents an itemset X with (FT) support s (it is the FT-support since the input is $\mathcal{FTFI}(D)$). Please notice the difference between (X, s) (Line 2) and (w, X) (Line 5) as the former represents an itemset and support, and the latter represents a weighted transaction. This algorithm iterates the itemset from the largest size, and set the weight accordingly. For easier understanding, we illustrate Algorithm 5.1 with our running example as follows.

Example 5.3. Continuing Example 5.1, the set of FTFIs and their supports (ordered from the largest sized itemsets) are $\{abc(2), abd(2), acd(2), bcd(2), ab(3), ac(3), ad(3), bc(4), bd(4), cd(2)\}$.

Initially D' is empty. In the first iteration when $(X, s) = (abc, 2)$, we have $\text{sup}^{D'}(X) = 0$, and therefore we set $w = 2 - 0 = 2$. As w is nonzero, we insert a weighted transaction $(2, abc)$ to D' . Similarly, we add transactions $(2, abd)$, $(2, acd)$, and $(2, bcd)$ to D' for the next three itemsets. Hence, at this state, we have database

D' illustrated as follows.

Weight	Itemset
2	abc
2	abd
2	acd
2	bcd

In the next iteration when $(X, s) = (ab, 3)$, we have $\sup^{D'}(X) = 4$, and therefore we set $w = 3 - 4 = -1$. Note that we allow negative weights. As w is nonzero, we insert a weighted transaction $(-1, ab)$ to D' . When $(X, s) = (bc, 4)$, we have $w = 0$, and hence we do not need to put the transaction to the new database. In the end, we have following database as the final result:

Weight	Itemset
2	abc
2	abd
2	acd
2	bcd
-1	ab
-1	ac
-1	ad
-2	cd

This database means, we have a transaction containing itemset abc with weight 2, a transaction containing itemset abd with weight 2, and so on. The support of an itemset is calculated as the sum of the weights for all transactions containing it. For example, the support of itemset ab is calculated as $2 + 2 + (-1) = 3$ from the first, second, and fifth transactions respectively.

The correctness of Algorithm 5.1 is trivial. As the itemsets are processed in decreasing order of size, those processed later would not affect the support of those processed earlier. The support of all itemsets in the transformed database equals to the intended support, which means the supports of itemsets are non-negative and follows anti-monotone property. Therefore, despite there are transactions with negative weight, the admissability of existing FIM algorithms are not affected. There are plenty of algorithms for weighted FIM. In particular, any algorithms using *horizontal data format*, i.e., data structure that stores the set of items per transaction, are admissible. Algorithms in this class include: Apriori [5], FP-Growth [37], and LCM [92].

Algorithm 5.2: Naïve Algorithm for Duality Problem

Input: Transactional Database D
Output: Transformed database D'

```

1  $D' \leftarrow \emptyset;$ 
2 for  $y \in T^D$  do
3    $\mathcal{F} \leftarrow \mathcal{FTFI}(y);$ 
4    $D' \leftarrow D' \cup IFM(\mathcal{F});$ 
5 return  $D'$ 

```

Note that Algorithm 5.1 does not solve the duality problem. The optimality and the complexity of Algorithm 5.1 will be discussed in the later part of this section, with regards to the duality problem.

5.3.2 Porting Naïve Algorithm for Duality Problem

What we have achieved from the naïve algorithm is:

Corollary 5.2. *Given a set of itemset (and support) $\mathcal{F}$, we can always form a transactional database D' such that $\mathcal{FI}(D') = \mathcal{F}$ using Algorithm 5.1.*

Another proposition we use to solve the duality problem is:

Proposition 5.3. *For any two transactional databases D_1 and D_2 , if $\sigma = 1$, then $\mathcal{FI}(D_1) \cup \mathcal{FI}(D_2) = \mathcal{FI}(D_1 \cup D_2)$.*

The condition $\sigma = 1$ simply means *all itemsets*, since otherwise, there might be itemsets which are infrequent in D_1 and D_2 , but frequent in $D_1 \cup D_2$. Note that the operator “ $\cup$ ” adds the support of two identical itemsets. For example, if D_1 contains itemset $(3, abc)$ and D_2 contains itemset $(2, abc)$, then $D_1 \cup D_2$ contains itemset $(5, abc)$.

Based on Corollary 5.2 and Proposition 5.3, we can solve the duality problem using Algorithm 5.2. For each transaction y , the algorithm finds all itemsets FT-supported by y (Line 3), constructs the *transformed database* (set of weight transactions) that supports those itemsets using IFM, and puts those transactions to the database D' (Line 4). The function IFM in Line 4 relates to the IFM problem (Algorithm 5.1).

This algorithm is correct because we transform each transaction individually. Which means, each iteration (Line 2–5) adds the support of itemset X in D' , for all X FT-supported by y , by one. Therefore, at the end, the support of any itemset

in D' is simply the number of transactions that FT-support it in D , which is essentially its FT-support. In other words, the condition $\sigma = 1$ in Proposition 5.3 does not restrict the correctness of Algorithm 5.2, i.e., Algorithm 5.2 is correct regardless the value of σ .

The remaining problem is to derive the itemsets FT-supported by a single transaction y (Line 3). By definition, an itemset X is FT-supported by y if y lacks no more than ε items in X . Rephrasing the statement, we can say that transaction y FT-supports all itemsets that contain no more than ε items not in $I(y)$. Mathematically, the set of *maximal* itemsets FT-supported by y (the positive border) can be written as $I(y) \cup \|I \setminus I(y)\|_\varepsilon$.

Example 5.4. Using our running example, let y be transaction 1, so we have $I(y) = ab$. With $\varepsilon = 1$, y supports all subsets of ab union with any 1-subset of cd , which means all subsets of abc and abd .

The number of itemsets FT-supported by y is given as $k = 2^{|I(y)|} \sum_{i=0}^{\varepsilon} \binom{|I \setminus I(y)|}{i}$, which also refers to the upper bound for number of itemsets in the transformed database, for one transaction. Analyzing the time complexity, as we need to calculate the support of each itemset in the new database (Line 3 of Algorithm 5.1), then the worst case complexity for transforming one transaction can be given as $O(k^2)$.

The number of transformed transactions and the algorithm time complexity are quite high and unacceptable. For example, with $m = 50$ and $|I(y)| = 5$, even with $\varepsilon = 1$, we have $32 \times 51 = 1,632$ transformed transactions. However, this number might be a very loose upper bound, since many transformed transactions might have zero weight.

We then work out the transformed database produced by the naïve algorithm. Using our running example, transaction 1 produces itemset $\{abc, abd, ab, ac, ad, bc, bd\}$. The algorithm first inserts itemset abc to D' with weight 1, followed by $(1, abd)$ and $(-1, ab)$, as presented in the following database:

Weight	Itemset
1	abc
1	abd
-1	ab

Interestingly, the support of itemsets ac , ad , bc , bd in D' are all 1. Which means, all those itemsets are weighted zero, and hence no more transactions are added to D' ! This indicates that our worst case analysis is indeed very loose.

5.3.3 Deriving the Weight of Itemsets

The number of transactions in the transformed database equals the number of nonzero-weighted itemsets. We then analyze the weights of individual itemsets. As we transform the transactions individually, the intended support of all itemsets FT-supported by y are one. Hence, we can formulate the weight of an itemset X , $w(X)$, as a recursive function:

$$w(X) = 1 - \sum_{A \supset X} w(A) \quad (5.2)$$

Our first result is presented in the following lemma:

Lemma 5.4. *When transforming transaction y , the weight of itemsets which are not supersets of $I(y)$ is always zero.*

We prove the lemma by induction as follows. Let the transaction y contain itemset X .

Base: The weights of all itemsets in the form of $(X \setminus a) \cup A$, for any $a \in X$, $A \cap X = \emptyset$ and $|A| = \varepsilon$ are 0.

Proof: Their only superset supported by y is $X \cup A$ with weight 1. Therefore, their weights are $1 - 1 = 0$.

Induction: Let Y not be a superset of X . Assuming all Y 's supersets, which are not supersets of X , have 0 weight, then Y has 0 weight as well.

Proof: Itemset Y can be formulated as $P \cup R$, with $P = X \cap Y$ and $P \cap R = \emptyset$. From Equation (5.2), we have:

$$\begin{aligned} w(Y) &= 1 - \sum_{Z \supset Y} w(Z) \\ &= 1 - \sum_{Z \supseteq (X \cup R)} w(Z) \\ &= 1 - w(X \cup R) - \sum_{Z \supset (X \cup R)} w(Z) \end{aligned}$$

because based on the assumption, the weights of all Y 's supersets, which are not in the form of $X \cup R$, are 0. As $w(X \cup R) = 1 - \sum_{Z \supset (X \cup R)} w(Z)$, substituting it to the above formula gives $w(Y) = 0$.

From Lemma 5.4, we can ignore itemsets which are not supersets of $I(y)$ since their weight is always zero. This is consistent with our previous example, i.e., none of the itemsets ac , ad , bc , and bd are supersets of ab , and their weights are zero.

The remaining step is to assign the weight of each transformed transaction. If the original transaction contains itemset X , any itemset transformed from itemset X can be written as $X \cup A, A \cap X = \emptyset, |A| \leq \varepsilon$. We can assume that $|X| < m - \varepsilon$, since otherwise, this transaction FT-supports all itemsets, which can be trivially translated into one transaction containing all items.

We can observe that the weight of an itemset depends only on its size. Specifically, the weight of itemset $X \cup A$ depends only on $m - |X| - |A|$ and $\varepsilon - |A|$, representing the number of items not in the itemset, and the number of items that can be added to the set. For simplicity, we use a to replace $m - |X| - |A|$ and b to replace $\varepsilon - |A|$, and $w(X)$ can be reformulated as $w(a, b)$.

Example 5.5. Let us consider transaction 1 of our running example, with $\varepsilon = 1$. The $w(ab)$ is represented by $w(2, 1)$ since there are two remaining items (c and d), and only one can be added. Similarly, $w(abc) = w(3, 0)$. Similarly, for transaction 2 with $\varepsilon = 2$, $w(b) = w(3, 2)$, $w(ab) = w(2, 1)$, and $w(abc) = w(1, 0)$.

Representing Equation (5.2) using a and b , we have:

$$w(a, b) = \begin{cases} 1 & \text{if } b = 0; \\ 1 - \sum_{i=1}^b \binom{a}{i} w(a-i, b-i) & \text{otherwise.} \end{cases} \quad (5.3)$$

Proof. The base case when $b = 0$ is trivial. For the recursion phase, the i -th iteration refers to the weight of the superset that contains i more items. Clearly, there are $\binom{a}{i}$ of them. As that superset contains i more items, both the number of items not in the set (the “ a ” part) and the items that can be added (the “ b ” part) are reduced by i , yielding $w(a-i, b-i)$. $\square$

Example 5.6. Using transaction 1 of our running example with $\varepsilon = 1$, we have $w(abc) = w(abd) = w(1, 0) = 1$, and $w(ab) = w(2, 1) = 1 - 2 w(1, 0) = -1$, which is again consistent to our previous example.

This weight formula is totally independent to the itemsets, even to the database, and hence can be calculated in advance and stored in a look-up table. Regarding the efficiency, since $a \leq m$ and $b \leq \varepsilon$, the space complexity is $O(m \varepsilon)$. From Equation 5.3, we need $b - 1$ iterations for calculating $w(a, b)$. Hence, the time complexity is $O(m \varepsilon^2)$, which is very small.

Using the identity of binomial coefficients, Equation (5.3) can be further simplified into:

$$w(a, b) = (-1)^b \binom{a-1}{b} \quad (5.4)$$

Proof. We prove this equation using mathematical induction and the identity of binomial coefficient. As the proof is quite lengthy and technical, we present it in Appendix B. $\square$

Example 5.7. Using Equation (5.4), we get $w(1, 0) = 1$ and $w(2, 1) = -1$, which are consistent with previous examples.

From Equation (5.4), it is obvious that $w(a, b)$ will never be zero. Hence, all itemsets in the form of $X \cup A$ with $|A| \leq \varepsilon$ are presented in the transformed database.

5.3.4 Optimality of the Size of the Transformed Database

Next, we analyze the optimality of our algorithm regarding the size of the transformed database. We show our algorithm is optimal for transaction-independent transformation.

When we transform each transaction independently, we need to transform each transaction y in D into several weighted transactions in D' . Therefore, the support of all itemsets FT-supported by y must be 1, and the rests must be 0. For brevity, let the itemsets FT-supported by a transaction y be denoted as $\mathcal{F}$.

It is easy to see that all itemsets not in $\mathcal{F}$ must have weight 0, since otherwise, their support in D' will not be 0, and all maximal itemsets in $\mathcal{F}$ (positive border) must have weight 1. As our algorithms set the weights of those maximal itemsets as 1 (Algorithm 5.1), while the rests are derived from them, hence our transformation algorithm is optimal, i.e., there is no other way to transform the database.

5.3.5 Properties of the Transformed Database

As discussed previously, a transaction containing itemset X is transformed into several transactions with form $X \cup A$, where A is all k -subsets of $I \setminus X$, with $k \leq \varepsilon$. Having this, the number of transformed transactions is given as $\sum_{i=0}^{\varepsilon} \binom{|I \setminus X|}{i} = O(|I|^{\varepsilon})$ per input transaction.

Given that each the number of items contained in each transaction is at most ε , then the density of the transformed database is increased by at most ε/m . However, noting that most FIM algorithms are output linear, and our transformation do not produce any spurious outputs, we can conclude that this increase of density does not slow down the mining algorithm. On the other hand, this increment can be seen as an artifact of FTFIM, which is natural since FTFIM allows longer itemsets to be mined compared to the usual FIM.

Algorithm 5.3: Algorithm for Duality Problem**Input:** ε , Database D **Output:** Transformed database D'

```

1 for  $y \in T$  do
2   if  $|I(y)| \geq m - \varepsilon$  then
3      $D' \leftarrow D' \cup \{(1, I)\};$ 
4   else
5     for  $i \leftarrow 0$  to  $\varepsilon$  do
6       for  $A \in \|I \setminus I(y)\|_i$  do
7          $a \leftarrow m - |I(y)| - i;$ 
8          $b \leftarrow \varepsilon - i;$ 
9          $w \leftarrow (-1)^b \binom{a-1}{b}$ 
10         $D' \leftarrow D' \cup \{(w, I(y) \cup A)\}$ 
11 return  $D'$ 

```

5.3.6 Transformation Algorithm

To conclude the transformation process, the optimal transformation algorithm is presented in Algorithm 5.3. For each transaction, we first check whether its size is no less than $m - \varepsilon$ (Line 2). If it is, then this transaction FT-supports all itemsets, hence we can translate it into one transaction that contains all items (Line 3). Otherwise, we iterate all itemsets FT-supported by this transaction, calculate their weights, and then put them into the transformed database (Line 5 – 12). As this algorithm takes constant number of iterations per output, this algorithm is optimal in terms of time complexity (trivial lower bound).

5.4 Concluding Remarks

The objective of the duality problem is to enumerate interesting itemsets using frequent itemset mining implementations. In this chapter, we solved this duality problem for reducing the FTFIM problem to the FIM problem. Our proposed algorithm is based on the inverse frequent-set mining (IFM) problem, which is to produce the database given the frequent itemsets.

We proved that we can transform (solve IFM on) each transaction independently to produce the desired result. In our transformation, each transaction is transformed into at most m^ε weighted transactions, and hence the total number of transactions is

5.4. CONCLUDING REMARKS

121

bounded by $n m^\varepsilon$. This number is optimal if we transform each transaction independently. We also proposed the transformation algorithm with complexity linear to the size of output, which is the trivial lower bound.

Chapter 6

Summarizing Frequent Itemsets

Previously, in Chapters 2 to 5, we discussed the enumeration problem, to enumerate all interesting patterns. While this problem is important and has received considerable interest, one common limitation of enumeration approaches is that the resulting itemsets are often abundant, and hence are difficult to interpret. In most real world scenarios, the interpretation is done manually, by scanning each pattern one by one. This limitation motivates researchers to summarize those interesting itemsets, or to present them in an interpretable form.

This chapter discusses one instance of the summarization problem, i.e., to summarize frequent itemsets where the knowledge are the itemsets and their support. In other words, by reading the summary, which is presented as a list of patterns, the analyst should be able to accurately infer all the frequent itemsets and their support. Knowing all frequent itemsets is useful, as these itemsets capture most important features in the database. As frequent itemsets are often abundant, it is important to summarize these itemsets, to improve their interpretability. This problem is the most popular variant of interesting itemsets summarization problem. However, even though this problem has been studied quite extensively, very few solutions satisfy our interpretability requirements.

Here, we propose the first summarization scheme which is interpretable, accurate, and concise. Our summary contains a set of *conditional profile*, which extends the itemset profile [104] by appending a *base* to the profile. Having the summary form, we also propose an efficient and effective algorithm to build the desired summary.

6.1 Preliminaries

One utility of frequent itemsets is to characterize the database. Having all frequent itemsets, we can see the important features in the database without losing much information. It is because what we lose are only infrequent features, which are usually insignificant for understanding the database. However, even the frequent itemsets are usually abundant, and hence are very difficult to analyze. Therefore, it is important to create a summary of frequent itemsets, to boost its interpretability without losing much information.

6.1.1 Related Works

Several patterns have been proposed to summarize frequent itemsets. First, we name *Maximal Frequent Itemset* (MFI) [10], which is the set of frequent itemsets which have no frequent supersets, and *Spanning Set* [1] which is the set of K itemsets, that can *cover* as many frequent itemsets as possible. These patterns do not encode the support of itemsets, and hence do not comprehensively represent the desired knowledge.

A similar case exists with *representative patterns* (RP) [103], which is a set of itemsets such that every frequent itemset in the database is *similar* to at least one itemset in the summary. The problem is, since we do not know which itemset in the summary represents each frequent itemset, we cannot derive the support of unmentioned frequent itemsets.

Example 6.1. Let the summary (RP) consisted of two itemsets, a (with absolute support 200) and $abc(100)$. Having this summary, it is impossible to derive the support of itemset ab , since it can be either similar to 100 (represented by abc) or 200 (represented by a).

Several other techniques create an archive instead of a summary. In other words, as long as the original information can be accurately derived, then the summary is perfect. Examples of representations within this class are Non-Derivable Itemsets (NDI) [22], generalized disjunction-free patterns [46], code tables (Krimp algorithm) [93], and Markov Random Fields (MRF) [94].

An itemset is considered non-derivable if its support cannot be derived by only knowing all its subsets. An itemset is disjunction-free if it cannot be derived by only knowing all its supersets. The code-tables approach associates each itemset with a *code*, which can be used to encode all transactions. Similarly, the MRF approach

builds a Bayesian network, such that the frequency of all frequent itemsets can be derived by a probabilistic inference. These variants require complicated logical (and probabilistic in case of MRF) inference, to derive the support of an itemset, by means of the inclusion-exclusion principle and solving a system of linear equations.

Example 6.2. Consider the following summary (NDI), and note the difficulty to see the support of itemset abc :

Itemset	support	Itemset	support
$\emptyset$	11	ab	4
a	8	ac	3
b	7	bc	3
c	5		

Since these summarization approaches require accessing the entire summary and complex operations, their resultant summaries are not interpretable.

Among the more interpretable summaries, we have *Closed Frequent Itemsets* (CFI) [61], δ -CFI [24], and condensed frequent pattern-base (FP-base) [64, 65]. A frequent itemset is considered a CFI (or δ -CFI) if none of its supersets has the same (or respectively, *similar*) support as itself. Similarly, the FP-base approach requires the supports of any frequent itemsets to be similar with any subsets presented in the FP-base (summary). On the one hand, these representations are only partially interpretable, as the information of an itemset is somehow dependent on the entire result-set. For example, with an incomplete result-set, we can never be sure of the support of non-closed itemsets, we can only have the lower bound of the support. On the other hand, these summaries suffer from the conciseness issue, as the number of δ -CFI is still lower bounded by that of MFI, which itself may be in the order of thousands.

Recently proposed summarization techniques, based on profile patterns [104, 41], are more promising in this aspect. In these approaches, the summary contains a list of *profiles*, which contains the *weight* of the profile, and a Bernoulli distribution vector of items. Each profile explains a snapshot of the database in which the items are independent.

Example 6.3. Consider a profile with weight 0.8, and consisting of 3 items, viz., a (with probability 0.6), b (0.7), and c (0.5). This profile represents a snapshot of a database with size 0.8, in which these three items are independently distributed with the given probability. This profile can be used to estimate the relative support of

itemset abc as $0.8 \times 0.6 \times 0.7 \times 0.5 = 0.17$, and ac as $0.8 \times 0.6 \times 0.5 = 0.24$. However, it cannot be used to estimate the relative support of itemsets containing items other than a , b , and c .

To build the profile, Yan et al. [104] cluster the frequent itemsets with respect to KL-divergence, and build the profile for itemsets in each cluster using the MLE principle. This algorithm is then enhanced by Jin et al. in [41] by modeling the problem to find the *best* profile as a regression problem.

We observe that each profile is interpretable, as the estimated frequency of an itemset can be calculated by simple multiplications of each item's frequency. Moreover, only small number of profiles are needed in order to build reasonably accurate summary [104, 41]. However, even though the individual profiles are interpretable, the whole summary is not interpretable. This is because using these approaches, an itemset may appear in many profiles. Yan et al. [104] propose to estimate the support as the maximum estimation from all profiles, while Jin et al. [41] propose to calculate the average of them. In any case, we need the entire summary before making any deductions. The second and the more lethal issue is regarding the accuracy. In some cases, it is impossible to accurately estimate the support of all frequent itemsets, as illustrated in the following example.

Example 6.4. Consider following set of itemsets:

Itemset	relative support	Itemset	relative support
a	0.7	c	0.8
b	0.6	abc	0.4
ab	0.6		

In order to accurately estimate the relative support of itemset ab , the weight of the profile must be small ($0.6 \approx w \times 0.7/w \times 0.6/w \Leftrightarrow w \approx 0.7$). Whereas, in order to accurately estimate the relative support itemset abc , the weight of the profile must be large ($w \approx 1.2$ using the same formula). As a profile that covers abc must cover ab as well, it is impossible to construct a summary that accurately estimates the supports of both abc and ab .

These claims about the inaccuracy of existing profile-based approaches are empirically validated in Section 6.5.2. These findings show that none of the previous proposals summarize frequent itemsets as required, as their forms do not guarantee interpretable, concise, and accurate summaries.

6.2 Summary Form: CP-Summary

Prior to designing the summarization algorithm, we first need to decide an appropriately interpretable form of the summary. As discussed above, pattern-profile has many desirable properties. However, as we have demonstrated, it is impossible for a profile-based summary to accurately estimate all itemsets. This issue is because a large profile covers too many itemsets, which are all of its subsets. To resolve this issue, we *extend* the definition of profile, by appending a condition (base) B to the profile. This profile only covers itemsets containing B . We denote this extended profile as *conditional profile*, or *C-Profile*. With this extension, it is possible for a profile to cover itemset abc without covering itemset ab , for example by making the former profile based on item c .

Example 6.5. Continuing Example 6.4, we can estimate both itemsets abc and ab accurately by using two C-Profiles. The first C-Profile has itemset c as a base and itemset ab in the profile, and the second C-Profile has empty base and itemset ab in the profile.

Another way to interpret conditional profile is by relating it to decision-tree. Continuing the previous example, in that database, the relative distribution between item a and b depends on the existence of item c . Therefore, it is natural to split the analysis into two cases, one where item c occurs and the other where item c does not occur. C-Profile is as interpretable as the traditional (unconditional) profile, because adding a condition does not make the inference of itemsets' relative support any harder.

Regarding accuracy, it is easy to construct a summary consisting of C-Profiles such that the relative support of all itemsets are accurately estimated. The simplest way is to let each profile cover exactly one frequent itemset, by setting the base as the itemset and the profile as empty.

6.3 Notations and Problem Definition

Our proposed form of summary consists of a set of *conditional profiles*, which is a modification of *itemset profile* [104]. An *itemset profile* is formed as Bernoulli distribution vector, which consists of a *weight* and a set of items along with the probability distribution.

Definition 6.1. [ITEMSET PROFILE] [104]. An itemset profile $P = (w : \{ \langle i, p \rangle, \dots \})$, consists of a weight w and a set of item-probability pairs (i, p) . The set of all

items included in a profile P is denoted as I_P , and the probability distribution of item $a \in I_P$ in P is denoted as $p_P(a)$. This profile covers itemsets $\{X | X \subseteq I_P\}$.

The estimated relative support of itemset X from a profile P , denoted as $r\hat{sup}_P(X)$, is calculated as:

$$r\hat{sup}_P(X) = w_P \prod_{a \in X} p_P(a). \quad (6.1)$$

Example 6.6. Profile $P = (0.5 : < a, 0.9 >, < b, 0.8 >)$. P covers itemset a, b, ab , and estimates their supports as $r\hat{sup}_P(a) = 0.5 \times 0.9 = 0.45$, $r\hat{sup}_P(b) = 0.5 \times 0.8 = 0.4$, and $r\hat{sup}_P(ab) = 0.5 \times 0.9 \times 0.8 = 0.36$ respectively.

Conditional profile, or C-Profile, is simply an extension of an itemset profile by a base B , and its closures. Formally, it is defined as follows.

Definition 6.2. [CONDITIONAL PROFILE]. A conditional profile $\mathcal{CP}$ is of the form $B[C]s|P$, where B is the itemset base, C is the closures of B , s is the support of B , and P is the itemset profile. $\mathcal{CP}$ covers all itemsets $\{X | B \subseteq X \subseteq B \cup C \cup I_P\}$. The “ s ” part of $\mathcal{CP}$ represents the relative support of the base, and is denoted as $s_{\mathcal{CP}}$.

The relative support of itemset X observed from a C -Profile $\mathcal{CP}$, denoted as $r\hat{sup}_{\mathcal{CP}}(X)$, is calculated as:

$$r\hat{sup}_{\mathcal{CP}}(X) = \begin{cases} s_{\mathcal{CP}} & \text{if } X \cap I_P = \emptyset \\ s_{\mathcal{CP}} w_P \prod_{a \in X \setminus (B_P \cup C_P)} p_P(a) & \text{otherwise.} \end{cases} \quad (6.2)$$

Example 6.7. Let the conditional profile $\mathcal{CP} = B[C]s|P = (pq[x, y] < 0.6 > | 0.5 : < a, 0.9 >, < b, 0.8 >)$. This profile covers all itemsets $\{X | pq \subseteq X \subseteq pqabxy\}$, for example, pqa , $pqxab$, and $pqxy$. The estimated relative support of itemset pqa , $r\hat{sup}_{\mathcal{CP}}(pqa) = 0.6 \times r\hat{sup}_P(a) = 0.6 \times 0.45 = 0.27$. Similarly, $r\hat{sup}_{\mathcal{CP}}(pqxab) = 0.6 \times r\hat{sup}_P(ab) = 0.216$, and $r\hat{sup}_{\mathcal{CP}}(pqxy) = 0.6$.

Ideally, the estimated support of an itemset should be similar to its actual support. Here, we formulate the support restoration error as:

$$err(X) = \begin{cases} 1 - \frac{r\hat{sup}(X)}{rsup(X)} & \text{if } r\hat{sup}(X) < rsup(X) \\ 1 - \frac{rsup(X)}{r\hat{sup}(X)} & \text{otherwise.} \end{cases} \quad (6.3)$$

This formula represents the relative error, ranging from 0 to 1. An itemset X is considered to be *accurately estimated* if it is not mis-represented (neither false negative nor false positive), and in case it is frequent, its support restoration error is

within a specified tolerance parameter ε . Mathematically, X is accurately estimated if:

$$\begin{cases} r\hat{sup}(X) < \sigma & \text{if } rsup(X) < \sigma \\ r\hat{sup}(X) \geq \sigma \wedge err(X) \leq \varepsilon & \text{otherwise.} \end{cases} \quad (6.4)$$

Definition 6.3. [VALID PROFILE]. A profile is *valid* if all itemsets it covers are *accurately estimated*.

Note that each valid profile can be analyzed by itself, as its contained information is all accurate. Our objective is to summarize itemsets into valid *C-Profiles*. For the sake of presentation, we define another term called *B-compressible* itemset, as follows.

Definition 6.4. [*B-COMPRESSIBLE ITEMSET*]. An itemset is *B-compressible* if it can be compressed into one valid profile with base B . Formally, Y is *B-compressible*, $B \cap Y = \emptyset$, if there is a valid C-Profile $\mathcal{CP} = B[C]s|P$ where $I_P = Y$.

Next, we define the notion of *valid summary* as:

Definition 6.5. [VALID SUMMARY]. A valid summary is a set of valid profiles such that each frequent itemset in the database is covered by at least one valid profile.

Based on the definition, a valid summary produces no false negatives or false positives. Once we have a valid summary, we can simply scan or read each C-Profile one by one, and determine the support of the covered itemsets. Finally, we define itemset summarization problem as:

Definition 6.6. [ITEMSET SUMMARIZATION PROBLEM]. Given a transactional database, the minimum support σ , and the error tolerance ε , return a summary R that maximizes the summary quality $\nu(R)$, where $\nu(R) = -\infty$ if R is not valid, and $\nu(R) = -|R|$ otherwise. In other words, we should return a valid summary with as few profiles as possible.

6.4 Algorithm

To simplify the algorithm design procedure hence allowing an efficient solution, and to improve the interpretability, we require every *C-Profile* to have a distinct *base*, and the *bases* should form a prefix-tree. This constrained problem will produce a summary with a tree-structure, as illustrated in Figure 6.1. Having these two restrictions, we can use an algorithm similar to *Tree-Regression* [41] to build the summary.

Figure 6.1: Tree-structure of a *CP-Summary*, with five *C-Profiles*, with bases $\emptyset$, a , b , c , and ab .

This algorithm is presented in Algorithm 6.1. Function *Build*($B, Tail$) (line 8) produces a summary (list of C-Profiles) for all frequent itemsets $\{X | B \subseteq X \subseteq B \cup Tail\}$. Function *Split*($B, Tail$) (line 1) finds the best *splitting item* a , and then recursively splits the itemsets to be summarized, depending on whether or not they contain item a . Function *BuildCProfile*($B, Tail$) (line 12) returns a valid C-Profile with base B and itemset $Tail$. This is always possible since itemset $Tail$ is B -compressible (Definition 6.4). Starting with the entire set of items and an empty base (corresponding to all frequent itemsets), this algorithm repeatedly splits the frequent itemsets until they are compressible. Regardless of the choice of the splitting item a , this algorithm always produces a *valid summary*.

Lemma 6.1. *The CP-summary algorithm produces a valid summary, regardless of the choice of the splitting item a .*

Proof. By definition, if a profile is valid, then all frequent subsets of the profiles are correctly generated. Since the first recursion call uses all the items as the parameter, then all frequent itemsets are covered. By induction, P_1 covers all itemsets containing item a , and P_2 covers all itemsets not containing item a . Hence, P_1 and P_2 are disjoint and complete, implying that the algorithm produces a valid summary. $\square$

This algorithm consists of two sub-problems, which are: (1) to find the best splitting item a (lines 1–7), and (2) to verify whether $Tail$ is B -compressible and if so, to build the profiles, i.e., to find the probability distributions (lines 11 and 12).

6.4.1 Finding the Best Splitting Items

The subproblem to find the best splitting items can be tackled by a brute force approach, which tries all possible items in each recursion call, and picks the one that

Algorithm 6.1: CP-Summary Algorithm

```

1 C-Profiles Split (itemset  $B$ , itemset  $Tail$ )
2 begin
3   item  $a \in Tail \leftarrow$  the best splitting item;
4   C-Profiles  $P_1 \leftarrow$  Build ( $B \cup \{a\}$ ,  $Tail \setminus \{a\}$ );
5   C-Profiles  $P_2 \leftarrow$  Build ( $B$ ,  $Tail \setminus \{a\}$ );
6   Return  $P_1 \cup P_2$ ;

7 C-Profiles Build (itemset  $B$ , itemset  $Tail$ )
8 begin
9    $Tail' \leftarrow \{x | x \in Tail \wedge B \cup \{x\} \text{ is frequent}\}$ ;
10  if isBCompressible( $B$ ,  $Tail'$ ) then
11    | Return  $\{BuildCProfile(B, Tail')\}$ ;
12  else
13    | Return Split ( $B$ ,  $Tail'$ );

14 C-Profiles Summarize
15 begin
16  | Return Build ( $\emptyset$ ,  $I$ );

```

produces the least number of profiles. This algorithm, though optimal, is not feasible, as shown by Lemma 6.2.

Lemma 6.2. *The brute force algorithm for finding the best splitting items has complexity $\Omega(|FI| + 2^{|I_F|})$, where I_F is the set of all frequent items.*

Proof. Observe that any frequent itemset can be a base of a profile. With respect to one base, let $I_B = \{x | B \cup \{x\} \text{ is frequent}\}$, and $Tail'$ be any subset of I_B . Therefore, the overall state space is $\sum_B 2^{|I_B|}$. When $B = \emptyset$, $I_B = I_F$, hence the state space for that particular base is $2^{|I_F|}$. $\square$

Jin et al. [41] solved a similar problem by using a greedy strategy, which chooses an item that results in the least restoration errors, if we summarize each partition with one profile. However, this approach is not promising for our problem. Consider a case where more than two profiles are required to accurately summarize the frequent itemsets. This greedy strategy will have no clue to split the items, since all splits produce very bad restoration errors. While this algorithm is not promising, there is nothing else to optimize since we have limited information.

We then try to view this problem from another perspective. Analyzing the procedure, we can see that the first recursion call (Line 4 of Algorithm 6.1) projects the

Figure 6.2: Splitting problem for constructing the prefix-tree.

Algorithm 6.2: Another perspective for splitting function

```

1 C-Profiles Split (itemset  $B$ , itemset  $Tail$ )
2 begin
3    $(B, P) \leftarrow bestCompressibleItemset(B, Tail);$ 
4    $Result \leftarrow \{BuildCProfile(B, P)\};$ 
5   sort  $Tail \setminus P$  in ascending or descending order;
6   for item  $a \in Tail \setminus P$  do
7      $Tail \leftarrow Tail \setminus \{a\};$ 
8      $Result \leftarrow Result \cup \mathbf{Build}(B \cup \{a\}, Tail);$ 
9   Return  $Result;$ 

```

database with respect to item a , and the second (Line 5) simply removes item a . Focusing on the second recursive call, the procedure of splitting items is equivalent to removing (and subsequently projecting) items, until the remaining itemset is B -compressible. Hence, instead of finding the splitting items, alternatively, we transform the problem into that of seeking the best compressible itemset.

This alternative perspective is presented in Algorithm 6.2, and illustrated in Figure 6.2 (b). First we find the *best* B -compressible itemset P (line 3). The profile formed by B and P is equivalent to that in line 12 of Algorithm 6.1. Having obtained this itemset, we throw each item not in P one by one as the splitting item of this branch, which is equivalent to item a in line 3 of Algorithm 1. Next, we formally prove that the *Split* function in Algorithm 6.2 is logically equivalent with that in Algorithm 6.1.

Lemma 6.3. *A summary can be produced by the splitting function in Algorithm 6.1 if and only if it can be produced by the splitting function in Algorithm 6.2.*

Proof. Let $Tail = Q \cup P$, where $Q = (q_1, q_2, \dots)$ is an ordered set, and P is B -compressible. We can simulate this procedure with the conventional perspective, by

Algorithm 6.3: Greedy algorithm for finding the best B -compressible itemset

```

1 C-Profiles Greedy (itemset  $B$ , itemset  $Tail$ )
2 begin
3    $P \leftarrow \emptyset$ ;
4   sort  $Tail$  in ascending or descending order;
5   for item  $a \in Tail$  do
6     if  $isCompressible(B, P \cup \{a\})$  then
7        $P \leftarrow P \cup \{a\}$ ;
8   Return  $P$ ;

```

taking items $q_1, q_2, \dots$, in order. □

From Lemma 6.3, we see that Algorithm 6.2 simply translates the problem of choosing the best item a into two sub-problems, viz., to choose the best compressible itemset P , and to find the best item ordering in $Tail \setminus P$.

Finding the best B -compressible itemset P

Larger P covers more frequent itemsets, hence resulting in lesser frequent itemsets to be summarized. So intuitively, we should choose P to be as large as possible. This is achieved by enumerating all compressible itemsets exclusively, which can be made efficient since compressibility satisfies the anti-monotone property.

Lemma 6.4. *If itemset X is B -compressible, then all itemsets $\{Y | Y \subset X\}$ are also B -compressible.*

Proof. For an itemset X to be compressible, there must be a probability distribution such that all subsets of X can be accurately estimated. Hence, if itemset X is compressible, we can use the same probability distribution for building the profiles of X 's subsets. □

However, the number of compressible itemsets can be very large. In one of our experiments, the length of a compressible itemset P reaches 100, implying that number of compressible itemsets is at least 2^{100} . This large number forbids an exact algorithm that finds the longest itemset.

Therefore, instead of focusing on the length of the itemset, we focus on individual items: whether we want P to contain more frequent items or less frequent items. If we keep high support items in P , then all branches will contain these frequent items. Otherwise, such an item will produce a large conditional database, or equivalently a

large branch. Both extremes are good in this case. If an item is closed with respect to the base (the most frequent), then it will also be closed in subsequent projected database, and hence can always be part of the subsequent P . On the other hand, if an item is barely frequent, it is very likely that its combination with other items becomes infrequent, implying that it will not appear in any subsequent $Tail'$. Since we have no clear winner, we tried both approaches and compared them empirically.

Finding either *lexicographically* smallest or largest itemsets with support order can be done efficiently with a greedy algorithm as in Algorithm 6.3. Starting with $P = \emptyset$, this algorithm iteratively checks each item a with the specified order, and puts a inside P if $P \cup \{a\}$ is B -compressible. Moreover, because of the anti-monotone property, this algorithm will always produce *maximal* itemsets, hence the resulting itemset should be reasonably large. Analyzing the complexity, we find that this algorithm has complexity linear with respect to the size of $Tail$, multiplied with the time taken to verify whether it is B -compressible (details in Section 6.4.2).

Ordering the items in $Tail \setminus P$

In each iteration (Line 6–9 in Algorithm 6.2), the size of $Tail$ is reduced by one, as also illustrated in Figure 6.2(b). Since the order of items in $Tail$ affects the shape of the prefix-tree, we must choose a good ordering. Let us consider two possible orderings, viz., ascending and descending order of support. If we order the items in increasing order of support, we will have many moderate sized branches. Otherwise, we will end up with one large branch and a lot of very small branches. Again, we have no clear winner, and hence we compare both orderings empirically.

6.4.2 Verifying and Forming Valid Profiles

For an itemset $Q \supset B$ to be B -compressible, there must be a conditional profile with $I_P = Q$, such that all itemsets $\{X | B \subseteq X \subseteq B \cup Q\}$ are *accurately estimated*. Based on Equation (6.4), the estimated support of an itemset X must satisfy several following conditions. First, if X is frequent, then we require $\max(\sigma, (1-\varepsilon) rsup(X)) \leq rsup_{CP}(X) \leq \frac{rsup(X)}{1-\varepsilon}$ where $rsup_{CP}(X) = s_{CP} w_P \prod_{a \in X} p_P(a)$. Similarly, when X is not frequent, we require $rsup_{CP}(X) < \sigma$. Note that all terms in the formula invoke only multiplication. Hence, by applying logarithmic transformation, we obtain linear inequalities. This problem has become identical to the problem of finding a *basic feasible solution* of linear program, which can be solved in an expected polynomial time, with respect to the number of variables and the number of constraints [43].

Optionally, we can minimize the restoration error per profile, by treating $(1 - \varepsilon)$ as another variable, and setting the objective function to maximize that value.

If we build the inequalities for all subsets of I_P , the number of constraints becomes $O(2^{|I_P|})$, which is infeasible even for moderately large I_P . We can reduce the number of variables by only using frequent itemsets and the *negative borders* of frequent itemsets (minimal itemsets which are infrequent) with respect to I_P . The correctness of this reduction is presented as follows.

Lemma 6.5. *If a negative border is accurately estimated, then all its supersets are also accurately estimated.*

Proof. Let the negative border be itemset X , which is covered by a C-Profile $\mathcal{CP}$. From Equation (6.2), we have $rsup_{\mathcal{CP}}(X \cup \{a\}) = rsup_{\mathcal{CP}} p_P(a) \leq rsup_P(X)$ for any item a , since $p_P(a) \leq 1$. If X is not frequent and it is accurately estimated, then $rsup_{\mathcal{CP}} < \sigma$. Similarly, $X \cup \{a\}$ will also be infrequent, and $rsup_{\mathcal{CP}}(X \cup \{a\}) \leq rsup_{\mathcal{CP}}(X) < \sigma$. Therefore, $X \cup \{a\}$ is also accurately estimated. $\square$

Using this observation, the number of constraints are bounded by $O(|FI(X)| + |NB(X)|)$, where $FI(X)$ is the set of all frequent subsets of X , and $NB(X)$ is the set of all negative borders subsets of X . $|NB(X)|$ itself is upper bounded by $|FI(X)| |X|$ since each frequent itemset has at most $|X|$ neighboring negative borders.

6.4.3 Time Complexity Analysis

For each recursive call (each node in the prefix-tree), our technique produces one profile. In each state, ignoring the verification time, the complexity is $O(n \log n)$ for sorting the items, where n is the number of items. Therefore, the total worst case complexity is given as $O(K (n \log n + V))$, where V is the verification time, and K is the size of the summary.

Analyzing the verification time is difficult since we need to solve a linear programming problem. For solving LP, the complexity is only known as being *polynomial to the number of variables and constraints*. Here, the worst verification time is when we consider all the frequent itemsets to be covered by the itemset. Let the number of frequent itemsets be $|FI|$, in the worst case scenario, we will have n variables and $n |FI|$ constraints. Therefore, the worst case complexity can be given as $O(K (n \log n + Poly(n |FI|)))$.

In practice, our algorithm usually finds a large first profile (with empty base), hence the time is dominated by the first recursive call. Therefore, we can assume

that the complexity is independent to K , i.e. $O(\text{Poly}(n \mid FI))$.

6.4.4 Quality Analysis

Next, we analyze the *tree-structure* of our *CP-Summary* (Figure 6.1).

Lemma 6.6. *The CP-Summary algorithm has an approximation ratio of $n/4$, where n is the number of items.*

Proof. Suppose the optimal solution consists of k C-Profiles $\{P_1, P_2, \dots, P_k\}$, and the distribution vector of P_i consists of n_i items. This condition can only happen when I_P is disjoint for all P , and no itemset formed by items in different profiles is frequent. Consider that we choose one profile P_x as the *C-Profile*. In this case, the current node (base) will have $n - n_x$ children, all of which are compressible. For example, if the child has item a as the base, it will contain only items which are originally in the same profile as a , so it is summarizable. Hence, the total number of profiles equal $n - n_x + 1$. Therefore, to minimize the number of profiles, we have to take the largest profile, and the worst case appears when all profiles are of the same size $n_i = n/k$. Since the optimal solution contains k profiles, and our solution produces $n - n/k + 1$ profiles, by simple algebra, we can derive that the worst approximation ratio is achieved when $k = 2$, yielding $n/4$ approximation ratio. $\square$

Analyzing the worst possible choice of P is difficult. It is possible to construct a test case where the optimal algorithm produces two profiles, while the poor algorithm produces $O(|FI|)$ profiles. Therefore, we analyze the quality using empirical evaluation.

6.5 Experimental Results

6.5.1 Result Quality

We used three real datasets, viz., BMS-Webview-1, Mushroom, and Chess. We also used a bunch of synthetic datasets for measuring the scalability and sensitivity of the algorithms. Details of these datasets are available in Appendix A.

Evaluation on Real Datasets

Our results are presented in Figure 6.3. For each dataset, we measure the number of profiles and the time taken, by varying the error tolerance and the minimum support.

6.5. EXPERIMENTAL RESULTS

137

Figure 6.3: Evaluation of CP-Summary in real datasets.

As a benchmark, we compare our summary size with the number of MFIs. We chose MFI because it is generally treated as the most compact representation [10], and constitutes the lower bound of several summarization techniques [61, 24]. In the figures, our techniques are denoted as a pair *asc/des*, corresponding to the heuristic used for each sub-problem. To recall, we have two heuristics and two subproblems. The first heuristic is to order the items not contained in P (Line 5 of Algorithm 6.2), and the second heuristic is to order the item in the Greedy algorithm, for choosing the best P (Algorithm 6.3). In each heuristic, we consider to sort the items in both ascending and descending order of support.

The summary size using our best heuristic is always smaller than the number of MFIs, except in the Mushroom dataset with 2% tolerance. Regarding scalability, the size of our produced summary grows when minimum support decreases. This is expected since we have more inputs, while the constraint is retained. Interestingly, except in the Mushroom dataset, the size of our summary is not sensitive to ε . This implies that, even with very low error tolerance, our technique can produce a very concise summary.

Regarding efficiency, overall, our summarization algorithm is quite efficient, especially in the Mushroom dataset. However, in some cases, viz., *asc-des* and *des-des* in the Chess dataset, our algorithm is not efficient. By deeper analysis, we see that those heuristics (RHS-des) produce a very large profile for that particular case with empty base. Hence, the verification time is very long. The time inconsistency is shown in the BMS-Webview-1 dataset when the time taken is unreasonably long when $\sigma = 0.2\%$ and $\varepsilon = 10\%$ (Figures 6.3(b) and (d)). As we see nothing unusual in the result (note the summary size in Figures 6.3 (a) and (c)), we suspect that this anomaly appears due to the inconsistent behavior of the LP-solver.

Evaluation on Synthetic Datasets

Next, we conducted experiments with synthetic data. Here, we ran three experiments where we fix the average length of transaction, the number of items, and the density. The minimum support is chosen in such a way that the number of frequent itemsets is roughly 10,000. The results are depicted in Figure 6.4.

First, we observe that our technique is always more concise than MFI. We can also see a trend – the denser datasets are easier to summarize. However, the impact of each heuristic is again not clear. In these datasets, *RHS-asc* is always more concise than *RHS-des*, with *asc-asc* being significantly more concise. This result is inconsistent with our previous study on Mushroom and Chess, where *asc-des* is more concise.

6.5. EXPERIMENTAL RESULTS

139

Figure 6.4: Evaluation of CP-Summary on synthetic datasets.

Figure 6.5: Quality of Hierarchical and K-Regression summarization algorithms. Note that CP-Summary always produces 100% precision, recall, and F_1 .

Moreover, all approaches perform reasonably well (compared to MFI), whereas *des-asc* had performed badly in Mushroom and Chess. We suspect that *asc-asc* suits the IBM-generated dataset, while *asc-des* suits the machine learning database. We leave deeper analysis of the heuristics for future work.

6.5.2 Comparison with Other Techniques

Next, we compare our approach with previous profile-based approaches [104, 41], and present the results in Figure 6.5. We use the *Hierarchical* technique from [104] (denoted as *H* in the figure), and the *K-Regression* technique from [41] (denoted as *KR*), which are the best techniques for their respective problem variants. For our technique, we use *asc-des* variant, that performs reasonably well in all datasets. We compare the summary quality using the following method. First, we set ϵ for our algorithm, which produces a summary of size k . We then produce summaries of size k using the competing algorithms, and extract the FIs from them. We consider

Dataset	Algorithm	TP	FP	FN	Precision	Recall	F_1
BMS-Webview-1 $\sigma = 0.2\%$, $K = 111$	K-Regression	789	7,381	9	0.097	0.989	0.176
	Hierarchical	294	1	504	0.997	0.368	0.538
Mushroom $\sigma = 25\%$, $K = 41$	K-Regression	5,529	26,347	16	0.173	0.997	0.296
	Hierarchical	3,971	246	1,574	0.942	0.716	0.814
Chess $\sigma = 80\%$, $K = 28$	K-Regression	5,070	11,638	13	0.303	0.997	0.465
	Hierarchical	4,734	2,904	349	0.620	0.931	0.744

Table 6.1: Detailed snapshot of the results of K-Regression and Hierarchical summarization algorithms.

an itemset to be frequent if its estimated support is greater than σ . Such itemsets can be simply enumerated using the depth-first enumeration of each profile. Having obtained the itemsets, we then measure the precision, recall, and F_1 . We consider an itemset as *true positive* if its support restoration error is no more than ε . We use the same real dataset as in previous experiments, with $\sigma = 0.2\%$, 25% , and 80% for BMS-Webview-1, Mushroom, and Chess dataset respectively.

Our algorithm is not plotted since it always achieves 100% precision and recall. From the figures, in general, we see that *Hierarchical* produces better precision, and *K-Regression* produces better recall when ε increases. Considering the F_1 measure, the best technique in the best case achieves around 75% performance.

A detailed snapshot, ignoring the support restoration error, is presented in Table 6.1. The summary size (K) is determined by the one produced by our algorithm with $\varepsilon = 10\%$. From the table, we can see that previous techniques are not comprehensive even for restoring the frequent itemsets.

Since we always achieve 100% quality in the results, we conclude that our technique is better for recovering the original frequent itemsets. To add, our technique also offers the feature to independently analyze each profile, which are not provided in those two previous works.

6.6 Concluding Remarks

In this chapter, we tackled the classical problem of summarizing frequent itemsets, where the knowledge is formed as the itemsets and their support. We demonstrated that previous techniques are not sufficient, as they are either non-interpretable, not accurate, or not concise. We proposed a summarization based on C-Profile, which is an extension of a pattern profile with an itemset base.

We designed an algorithm to build the summary. In short, our algorithm produces a prefix-tree based summary, where each summary pattern has a distinct base, and all the bases form a prefix tree. The choice of the branching of the tree is determined using two heuristic algorithms. To summarize a bunch of itemsets into a C-Profile, we use an optimal linear programming approach. Given the estimation error tolerance ε , the produced summary concisely represents all frequent itemsets, and accurately estimates all their support within ε (zero false negative). Our produced summary also never estimates infrequent itemsets as frequent (zero false positive), and each frequent itemset is *inferred* by exactly one summary pattern (zero redundancy). Through extensive empirical evaluations, we show that our algorithm is efficient and effective for producing concise summaries.

Chapter 7

Summarizing Subspace Clusters

This chapter discusses the problem to summarize interesting itemsets, where the knowledge is formed as the itemsets and their associated transactions. This knowledge is normally used to find a set of objects which are similar in some attributes. This set of objects is also commonly referred as subspace cluster [45]. However, the subspace clusters in a database (objects similar in several attributes) are usually abundant, and a lot of them are redundant.

This issue attracts researchers to summarize subspace clusters. An ideal summary should cover/infer most interesting patterns (high recall) and should not contain misleading information (high precision). However, all previous summarization algorithms either compromise recall (coverage) by maintaining perfect precision, or vice versa. That is, no summarization scheme tolerates both precision and recall. We argue that, in practice, analysts are more willing to compromise a little of both rather than retain the perfection of one property. The simplest way to do this is by compromising each property consecutively. However, this approach is ineffective. Here, we propose a novel algorithm that compromises both precision and recall concurrently. Through extensive empirical evaluations, we show that our proposed algorithm is much more effective than the consecutive approaches.

7.1 Preliminaries

In many recent applications such as mining in sensor networks, text mining, or micro-array analysis, we often deal with very large and high dimensional databases. Analyzing a high dimensional database is difficult, since very few objects are similar with respect to the entire attributes. To overcome this, researchers have proposed to find

objects which are similar, not necessarily in all attributes, but only in some subset of attributes (which is also called subspace) [3, 45]. Such a set of objects along with the corresponding subspace is denoted as a subspace cluster.

Though subspace clusters have been shown to be useful [3], the number of subspace clusters in a database are typically abundant, and most of them are redundant [57]. This limitation has attracted researchers to summarize the resulting clusters, usually by finding some representatives of the clusters. This problem is an instance of the interesting itemset summarization problems, where each subspace cluster is treated as an interesting itemset (or attribute) along with its associated transactions (or objects).

7.1.1 Related Works

The subspace clustering problem has been studied quite extensively in the past couple of decades. A subspace cluster $(Y \times X)$ means that all objects Y are related in attributes X . There is another similar concept called bi-cluster [53], where the data is formed as two sets of objects $\mathcal{X}$ and $\mathcal{Y}$, along with some connections between them. In this case, a bi-cluster $Y \times X$, $Y \subseteq \mathcal{Y}$ and $X \subseteq \mathcal{X}$ is a pair of object-sets, where *most* of the objects in X are related to most of the objects in Y .

Previous works [3, 45, 53] mainly discuss the type or the definition of a subspace-cluster, and the algorithms to find the clusters. On the contrary, the focus of this chapter is to draw an analogy between the summarization problem and the subspace clustering problem. That is, assuming we know the desired knowledge (which is the set of all subspace clusters in this case), what kind of summary is useful to represent that knowledge?

Our objective can be seen as formulating the evaluation measure of subspace clusters. There are several proposals to evaluate the subspace-clusters [62, 58]. However, these previous works are mostly based on the information-theory paradigm, and do not consider the interpretability aspect of the summary.

The problem to summarize subspace clusters has become quite popular in the past several years. To the best of our knowledge, all previous works use the same summary form, which is a matrix of transactions and items. However, despite using the same form, the ways in which they interpret the summary vary. Several works [57, 8, 103] treat the knowledge of a pattern $Y \times X$ as: all objects in Y are *sufficiently* similar. With this definition, two patterns are considered similar if their object-sets

are similar. Clearly, some information is lost as they ignore the specifying attribute-set X . Another related variant is tackled by Xiang et al. [102]. While [102] also considers the attributes, they require concurrent access of all summary patterns to infer the knowledge. This issue is illustrated in following example.

Example 7.1. Consider the following categorical database with 20 objects and 2 attributes:

	A_1	A_2
$O_1 - O_{10}$	a	x
$O_{11} - O_{20}$	a	y

In this database, [102] would return two patterns $\{(O_{1..10} \times \{A_1 = a, A_2 = x\}), (O_{11..20} \times \{A_1 = a, A_2 = y\})\}$ as the summary. The fact that objects $O_{1..20}$ are the same in attribute A_1 is not obvious, especially if these two patterns are presented separately. In other words, in order to obtain the complete knowledge of the subspace clusters, we need to see the whole representation *together*, which is clearly impractical.

Regarding the accuracy measure of the summary, most previous works define their own notion of *precision* and *recall* [100, 82, 57, 103]. Müller et al. [57] measure coverage of an input pattern $Y \times X$ as the number of objects in Y that appear in any summary-pattern. Xin et al. [103] measure the coverage of an input pattern $Y \times X$ as the *most similar* patterns in the result-set, which is defined using the Jaccard similarity of the object-set. Wong et al. [100] measure the spuriousness as the *distance* between two patterns p and q if they are *fused* into one pattern, by combining the objects and attributes of both p and q . Robardet et al. [82] measure the spuriousness of a pattern $Y \times X$ as the number of *holes* in the matrix, which can be generalized into the number of holes in each object (row) and each attribute (column).

We also notice that many subspace clustering algorithms have their inherent *summarization measure*, in the form of the pruning mechanism and the stopping criterion [3, 8]. However, these works mainly focus on the efficiency of the algorithm and the empirical quality of the result (e.g., the classification accuracy). Furthermore, the criterion they use is somehow subsumed by the specific summarization algorithms mentioned above.

Interestingly, all previous works either compromise precision while maintaining perfect recall [100, 82], or vice versa [57, 103]. It is surprising that no previous work compromises *both* precision and recall. In many real world applications, it is hard

Figure 7.1: One-cluster summaries of (a) the input clusters, provided by (b) and (c) the consecutive approaches, and (d) the proposed approaches, as shown by green-thatched regions.

to imagine why one would need the output to be *perfectly* non-spurious or the input to be *perfectly* covered at an expense of the other property. In fact, we argue that most applications would indeed not mind compromising a bit of both, rather than sacrificing a lot of one property for the sake of achieving perfection in the other. This argument is illustrated in the following example.

Example 7.2. Consider the database in Figure 7.1 (a) that consists of 4 clusters, A , B , C , and D , and assume that we want to summarize them by using only one cluster. If we require the summary to be perfectly non-spurious, the best we can achieve is as presented in Figure (b), by taking pattern B (or C) as the summary. This summary is not ideal, as many subspace clusters, such as C and D , are hardly covered. Similarly, if we require all subspace clusters to be perfectly covered, we need to fuse all clusters into one big cluster, as presented in Figure (c). This big cluster is not desirable, as it is too spurious. Without requiring perfect precision or recall, we can produce a reasonably good summary, by fusing clusters B and C , as in Figure (d). We can see that all input clusters are reasonably well-covered, and the resulting cluster is not very spurious.

7.2 Summary Form

In this work, our summary-pattern is in the form of objects and attributes $Y \times X$, which can be interpreted as: all objects in Y are similar to all attributes in X . For measuring the accuracy, we relate the notion of recall and precision with coverage and spuriousness [33]. That is, an input pattern is considered *covered* if its *similar* to at least one summary pattern, and a summary pattern is considered non-spurious

Figure 7.2: Spuriousness and coverage for representing pattern $p = (Y_p \times X_p)$ by pattern $r = (Y_r \times X_r)$. The green-hatched region represents uncovered patterns, and the red-dotted region represents spurious patterns.

if it is *similar* to at least one input pattern. Unlike previous works which force either perfect precision or recall, our goal is to slightly compromise both precision and recall, to boost the conciseness of the summary.

7.3 Notations and Problem Definition

This chapter deals with subspace clusters. Each cluster is formulated as a matrix $(Y \times X)$, which can be interpreted as: all objects in Y are similar with respect to all attributes in X . For a pattern $p = (Y \times X)$, $|p|$ denotes the size of p , which is $|Y| \times |X|$. This pattern can also be treated as an itemset and its associated transactions. The *intersection* of two patterns $p = Y_p \times X_p$ and $q = Y_q \times X_q$ is defined as the intersection of the matrices, and is mathematically given as: $p \cap q = (Y_p \cap Y_q) \times (X_p \cap X_q)$. Similarly, the *union* (or also called as the *fusion*) of two patterns p and q is defined as the union of both the objects and attributes of the matrices, and is mathematically given as: $p \cup q = (Y_p \cup Y_q) \times (X_p \cup X_q)$.

The objective of this work is to produce a summary/representation R , which is a set of patterns, given a set of subspace clusters P . An ideal representation should be both concise (interpretable) and accurate. Here, we measure the conciseness using the number of patterns, and measure the accuracy using *spuriousness* and *coverage* [33]. These measures are analogous with precision and recall respectively in information retrieval.

The *coverage* of an input pattern $p \in P$ by a summary pattern $r \in R$ measures how much of p is covered by r , and is formulated as:

$$cov_r(p) = \frac{|p \cap r|}{|p|}. \quad (7.1)$$

The rationale behind this metric is as follows. The pattern $r = (Y_r \times X_r)$ means that all objects in Y_r are similar in attributes X_r , which implies that all objects in $Y_q \subseteq Y_r$ are similar in attributes $X_q \subseteq X_r$. Therefore, the information of pattern $p = (Y_p \times X_p)$ covered by r is given as $|p \cap r|$, as illustrated in Figure 7.2.

Following the above rationale, when there are many summary patterns, we only need to consider the one that best covers the input pattern. Hence, the coverage of a pattern $p \in P$ by a representation R is formulated as:

$$cov_R(p) = \max_{r \in R} cov_r(p). \quad (7.2)$$

A pattern p is considered well-covered if $cov_R(p) \geq 1 - \varepsilon_r$, with degree of compromise ε_r . Previous works (e.g., [104]) compute the coverage of P as the average coverage of its individual patterns. However, this may result in representations that completely miss some input patterns merely because the average coverage of the other patterns is high. Since we wish to guarantee that all patterns are well-covered, we calculate the coverage of P as the *minimum* coverage of individual patterns, as:

$$cov_R(P) = \min_{p \in P} cov_R(p). \quad (7.3)$$

Therefore, for P to be considered covered by a representation R , the necessary condition is $cov_R(P) \geq 1 - \varepsilon_r$. Next, we discuss about *spuriousness*. Following previous notations, a summary pattern r is considered spurious if it does not represent any input pattern. Therefore, the spuriousness of a summary pattern $r \in R$ with respect to P is formulated as:

$$spur_P(r) = 1 - \max_{p \in P} \frac{|p \cap r|}{|r|}. \quad (7.4)$$

The rationale is the same as that for coverage, and is illustrated in Figure 7.2. A summary pattern r is considered non-spurious if $spur(r) \leq \varepsilon_s$ with degree of compromise ε_s . The spuriousness of the representation R is calculated as the *maximum* spuriousness of individual patterns, because of the same reason as in the case of

coverage.

$$spur_P(R) = \max_{r \in R} spur(r). \quad (7.5)$$

Therefore, for R to be considered non-spurious, the necessary condition is $spur_P(R) \leq \varepsilon_s$. Having formalized the ingredients, we can now define the summarization problem as follows.

Definition 7.1. [SUMMARIZATION PROBLEM]. Given a set of subspace clusters P (interesting itemsets) and the parameters ε_r and ε_s specifying the degree of compromise for coverage and non-spuriousness respectively, return a summary R that maximizes the summary quality $\nu(R)$, where $\nu(R) = \infty$ if $cov_R(P) < 1 - \varepsilon_r$ or $spur_P(R) > \varepsilon_s$, and $\nu(R) = -|R|$ otherwise. In other words, we should return a summary with as few profiles as possible, such that R is not spurious and P is well-covered.

7.4 Approach

This section discusses the approaches to solve the summarization problem. We first analyze how to compromise coverage and non-spuriousness. This analysis leads us to design two algorithms, the first considers each property consecutively, and the second considers both concurrently in each iteration.

7.4.1 Analysis of Compromising Recall and Precision

Compromising Recall – Pattern Selection Approach

This section discusses how to compromise recall, while retaining precision. To do this, every summary pattern must be a part of some input pattern. Therefore, the only way is to select some input patterns as the representatives [103]. We denote this approach as the *pattern selection approach*.

From Equation (7.1), a pattern p covers another pattern q if $cov_p(q) \geq 1 - \varepsilon_r$. We can model the coverage between patterns as a *coverage graph* by considering each input pattern as a vertex, and creating an edge (u, v) if pattern u covers pattern v . As the set of summary patterns is a subset of the input patterns which can cover all input patterns, it is essentially equivalent to a *dominating set* in the coverage graph. In other words, if $\Gamma(r)$ is the set of patterns that can be covered by pattern r , R becomes a *valid summary* if $\bigcup_{r \in R} \Gamma(r) = P$.

Unfortunately, finding the dominating set with minimum size is an NP-complete problem, and hence cannot be solved efficiently and optimally unless $P = NP$. Previous work [103] solves a similar problem (for summarizing frequent itemsets) in a greedy manner, by iteratively selecting the pattern (as the summary pattern) that covers the most number of uncovered input patterns.

If N is the number of input patterns, this algorithm has time complexity $O(N^2)$. However, prior to running this algorithm, we need to construct the coverage graph, with time complexity $O(N^2C)$, where C is the number of iterations to calculate the coverage, which will be discussed in more detail in Section 7.4.2. This greedy algorithm is proven to have $\ln |N|$ approximation ratio.

Compromising Precision – Pattern Fusion Approach

This section discusses how to compromise precision, while retaining recall. To do this, every input pattern must be a part of some summary pattern. Therefore, the only possible approach to reduce the number of patterns is by repeatedly *fusing* input patterns, as has been done in [100, 82]. This *fusing* operation is illustrated in Example 7.2. We term the algorithm that repeatedly fuses two patterns as the *pattern fusion approach*. The soundness of this approach is explained by the following lemmas.

Lemma 7.1. *The only way to reduce the size of a representation without reducing coverage is by fusing patterns.*

Proof. Since every input pattern must be a subset of at least one representation, then by the pigeon-hole principle, at least one representation is a superset of at least two patterns, say p and q . In other words, that representation is a superset of $p \cup q$, which can be built by fusing p and q . $\square$

Lemma 7.2. *Any valid representation can be constructed by repeatedly fusing two patterns.*

Proof. Consider three patterns a , b , and c . By contradiction, this lemma is false if we can merge those patterns into $a \cup b \cup c$, but there is no sequence of fusion procedures that can achieve that. Without loss of generality, we can assume that $|a| \leq |b| \leq |c|$. Since $a \cup b \cup c$ is valid, we have $\text{spur}(a \cup b \cup c) \leq \varepsilon_s$, or $\frac{|c|}{|a \cup b \cup c|} \geq 1 - \varepsilon_s$. As $|b \cup c| \leq |a \cup b \cup c|$, then we have $\text{spur}(b \cup c) \leq \text{spur}(a \cup b \cup c) \leq \varepsilon_s$. Hence, pattern $b \cup c$ is valid. In other words, we can fuse b and c first, and then fuse the resulting pattern with a . $\square$

The remaining problem is to find the optimal fusion sequence, which leads to representations with the least number of patterns. Previous works [100, 82] relate this problem with clustering, and then propose to use a hierarchical method, by repeatedly fusing two patterns with the lowest spuriousness (the most similar patterns), until no more patterns can be fused.

If N is the number of input patterns, this algorithm can be implemented with $O(N^2 C \lg N)$ time and $O(N^2)$ space, by storing all pairs of patterns in the priority queue. As above, the term C refers to the number of iterations to calculate the spuriousness of a pattern.

Compromising Precision and Recall

Summarizing previous discussions, we compromise recall by selecting/removing patterns, and compromise precision by fusing patterns. Therefore, to compromise both properties, we need to consider both operations. We now analyze the impact of fusing patterns to the summarization power, as formally stated in following lemmas.

Lemma 7.3. *Let $\Gamma(p)$ be the set of patterns covered by p (including p itself), then $\Gamma(p \cup q) \supseteq \Gamma(p) \cup \Gamma(q)$.*

Proof. From Equation (7.1), we have $\text{cov}_p(x) \leq \text{cov}_{p \cup q}(x)$. Therefore, if x is covered by p , then it is also covered by $p \cup q$. $\square$

This lemma implies that the set of patterns covered by a pattern p will not be reduced if it were fused with other patterns.

Lemma 7.4. *Let $\Upsilon(p)$ be the set of patterns (excluding p) that can be fused with p . If $|p| \geq |q|$, then $\Upsilon(p \cup q) \subseteq \Upsilon(p)$.*

Proof. From Equation (7.4), for any pattern x , we have $\text{spur}(p \cup x) \leq \text{spur}(p \cup q \cup x)$, since the largest pattern covered by $p \cup x$ is the same as that covered by $p \cup q \cup x$, while the pattern's size is either smaller or equal. Therefore if x can be fused with $p \cup q$, then it can also be fused with p . $\square$

This lemma implies that fusing two patterns reduces their potential to be fused with other patterns, which means, we should not do any unnecessary fusions.

7.4.2 Algorithms

Having analyzed how to compromise each property, we can now design the algorithms to build the summary.

Consecutive Algorithm

One way to compromise both precision in recall is by using a two-phase algorithm that considers each property separately and consecutively. Our first variant, *Select-Fuse*, first *selects* some patterns that covers all the patterns, and then *fuses* them to further compress the summary. We can safely fuse the patterns during the second phase, because the fusion operation never reduces the covered patterns, as stated in Lemma 7.3. Conversely, our second variant, *Fuse-Select*, first *fuses* the patterns until no more patterns can be fused, and then *selects* some patterns (from the result of the first phase) that can cover the entire input patterns.

However, these consecutive algorithms are not very effective, as only few patterns can be summarized during the second phase. This is because the results of the first phase are already quite distinctive. For example, in the *Select-Fuse* variant, it is unlikely that two *similar* patterns belong to the result of the first phase, as the patterns they cover highly overlap. In other words, having taken a pattern, the benefit for taking similar patterns is small, which is then avoided by the algorithm. A similar argument can be drawn for the other variant, where the algorithm fuses *neighboring* patterns in the first phase.

The complexity of these consecutive algorithms is dominated by the first phase, as the input size of the second phase is generally much smaller. Therefore, we can deduce that *Select-Fuse* has $O(N^2 C \lg N)$ time complexity, and *Fuse-Select* has $O(N^2 C)$ time complexity.

Concurrent Algorithm

The limitation of the consecutive approaches is that, when compromising one property (e.g., precision), the algorithm optimizes the conciseness without considering the other property (e.g., recall). Therefore, the second phase becomes merely a *refining* phase and is not effective. To resolve the limitation, we propose the CONCURRENT algorithm that concurrently considers both properties in each iteration. This algorithm is illustrated in Algorithm 7.1. As before, $\Gamma(p)$ returns all patterns that can be covered by p , and $\Upsilon(p)$ returns all patterns that can be fused with p .

The CONCURRENT algorithm is described as follows. First, we find a pattern p that covers most uncovered patterns (Line 4). Next, we fuse it with the *best* pattern q^* that satisfies $\text{spur}(p \cup q^*) \leq \varepsilon_r$ and q^* covers some uncovered patterns (Line 11). We consider two possible notions of *best*, viz., with respect to coverage (Line 7) and spuriousness (Line 9). To optimize the coverage, we choose q^* such that $p \cup q^*$ covers

Algorithm 7.1: The CONCURRENT algorithm

Input: Subspace clusters P , degree of compromise ε_r and ε_s
Output: Representation R

```

1  $left \leftarrow P$ ;
2  $result \leftarrow \emptyset$ ;
3 while  $left \neq \emptyset$  do
4    $p \leftarrow \arg \max_{x \in P} |\Gamma(x) \cap left|$ ;
5    $left \leftarrow left \setminus \Gamma(p)$ ;
6   while  $(Q = \{q \in \Upsilon(p) | \Gamma(q) \cap left \neq \emptyset\}) \neq \emptyset$  do
7     if  $best = \text{maximum coverage}$  then
8        $q^* \leftarrow \arg \max_{q \in Q} |\Gamma(q) \cap left|$ ;
9     else if  $best = \text{minimum spuriousness}$  then
10       $q^* \leftarrow \arg \min_{q \in Q} spur(p \cup q)$ ;
11      $left \leftarrow left \setminus \Gamma(q^*)$ ;
12      $p \leftarrow p \cup q^*$ ;
13    $result \leftarrow result \cup \{p\}$ ;
14 return  $result$ ;
```

most uncovered patterns, and to optimize the spuriousness, we choose q^* such that $p \cup q^*$ has the lowest spuriousness, i.e., q^* is the most similar to p . This procedure is repeated until all patterns are covered.

This algorithm is effective, because the fusion operations (which compromise precision) are guided to optimize the coverage. Unlike the *Select-Fuse* approach, CONCURRENT still considers the neighboring patterns for fusions, even though their covered patterns highly overlap. Compared to the *Fuse-Select* approach, CONCURRENT never fuses any pattern that does not cover any uncovered patterns. Note that such a fusion is not only useless, but also detrimental. This is because, as a consequence of Lemma 7.4, fusing two patterns reduces their potential to be fused with other patterns.

Example 7.3. Relating to our example in Figure 7.1, assume $\Gamma(A) = \{A\}$, $\Gamma(B) = \{A, B\}$, $\Gamma(C) = \{C, D\}$, $\Gamma(D) = \{D\}$, and $\Upsilon(B) = \{A, C\}$ (the rest are not important). CONCURRENT first takes pattern B since it covers patterns A and B . Next, pattern B is fused with pattern C that covers patterns C and D . Note that this algorithm will not fuse B with A since pattern A does not cover any uncovered patterns. As all patterns have been covered, the algorithm is terminated, with the same result as in Figure 7.1 (d).

Next, we analyze the complexity of CONCURRENT. Let the number of inputs be N . Before running the algorithm, we need to build the *coverage graph* (c.f.,

section 7.4.1). To do this, we need to check all pairs of patterns, and hence the time complexity is $O(N^2 C)$, where C is the number of iterations to iterate a pattern, i.e., to fuse two patterns, or to calculate the coverage and the spuriousness of a pattern.

At a glance, CONCURRENT seems inefficient, since the while loop in Line 3–16 may take $O(N)$ iterations, and the inner-loop in Lines 6–14 may take $O(N^2 C)$ iterations, resulting in $O(N^3 C)$ time complexity.

However, this number is highly overestimated, and is further reduced using amortized analysis. We observe that whenever Line 13 is executed, the number of uncovered patterns reduces by one. Therefore, all operations in the Line 6–14 loop are executed at most N times. Each iteration in Lines 4, 8, and 10 takes $O(N)$ time each for finding the *best* pattern, yielding $O(N^2)$ overall. Line 6 takes $O(NC)$ time for iterating the patterns and calculating the spuriousness, yielding $O(N^2 C)$ overall, and Line 13 takes $O(C)$. The remaining are Lines 5 and 12, which might be expensive since we need to update the coverage graph. However, since at least one edge is removed whenever Lines 5 and 12 are called, the total time complexity for these two lines are $O(N^2)$. Therefore, the total time complexity is given as $O(N^2 C)$.

In the worst case, the term C can be $O(|A| + |O|)$, where $|A|$ is the number of attributes and $|O|$ is the number of objects. However, for calculating the spuriousness and coverage, we often only need to iterate the attributes to filter-out too spurious or non-covered patterns, i.e., the intersection of the attributes is already too small. Moreover, many subspace clusters have few attributes, and hence this algorithm is efficient in practice, as shown in our empirical study.

7.5 Experimental Results

In this section, we provide the empirical evaluation for the efficiency and effectiveness of our summarization algorithm. Our problem accepts subspace clusters (interesting itemsets) as inputs. For the sake of simplicity and repeatability, we use closed frequent itemsets and the supporting transaction sets in a categorical database as the input clusters. By using closed frequent itemsets (CFI), a matrix $(Y \times X)$ is a subspace cluster if all objects in Y share exactly the same value in each attribute in X , and the number of objects $|Y|$ is no less than a given parameter σ . Other requirements are that, there is no other object-set $Y' \supset Y$ which has the same value in each attribute in X , and there is no other attribute-set $X' \supset X$, such that Y are the same w.r.t. X' . In our experiment, we use the implementation of *FPclose* by Zhu and Grahne [112].

We used three real categorical datasets, viz., Mushroom, Chess, and Soybean.

7.5. EXPERIMENTAL RESULTS

155

Algorithm	Mushroom ($\varepsilon = 0.15$)			Soybean ($\varepsilon = 0.15$)			Chess ($\varepsilon = 0.15$)		
	$\sigma = .05$	.1	.15	.4	.43	.46	.7	.75	.8
Input (CFI)	12,843	4,886	2,262	19,794	14,513	10,763	23,893	11,526	5,084
<i>Select-Fuse</i>	1,344	650	400	480	410	364	343	197	92
<i>Fuse-Select</i>	–	1,346	653	–	–	671	–	–	128
CONCURRENT _{cov} (% of Input)	1,210 (9.4%)	560 (11%)	339 (15%)	347 (1.8%)	303 (2.1%)	267 (2.5%)	227 (.95%)	146 (1.3%)	78 (1.5%)
(% of Consecutive)	(90%)	(86%)	(85%)	(72%)	(74%)	(73%)	(66%)	(74%)	(85%)
CONCURRENT _{spur} (% of Input)	1,253 (9.8%)	580 (12%)	343 (15%)	388 (2.0%)	329 (2.3%)	307 (2.9%)	274 (1.1%)	165 (1.4%)	91 (1.8%)
(% of Consecutive)	(93%)	(89%)	(86%)	(81%)	(80%)	(84%)	(80%)	(84%)	(99%)

Table 7.1: Number of patterns produced by the Concurrent and Consecutive algorithms, varying σ .

Algorithm	Mushroom ($\sigma = 0.15$)			Soybean ($\sigma = 0.46$)			Chess ($\sigma = 0.8$)		
	$\varepsilon = .1$	.15	.2	.1	.15	.2	.1	.15	.2
Input (CFI)	2,262			10,763			5,084		
<i>Select-Fuse</i>	779	400	178	826	364	149	226	92	53
<i>Fuse-Select</i>	924	653	489	1,136	671	413	241	128	77
CONCURRENT _{cov} (% of Input)	702 (31%)	339 (15%)	152 (6.7%)	779 (7.2%)	267 (2.5%)	92 (.85%)	226 (4.4%)	78 (1.5%)	34 (.67%)
(% of Consecutive)	(90%)	(85%)	(85%)	(94%)	(73%)	(62%)	(100%)	(85%)	(64%)
CONCURRENT _{spur} (% of Input)	697 (31%)	343 (15%)	156 (6.9%)	777 (7.2%)	307 (2.9%)	98 (.91%)	226 (4.4%)	91 (1.8%)	40 (.79%)
(% of Consecutive)	(89%)	(86%)	(88%)	(94%)	(84%)	(66%)	(100%)	(99%)	(75%)

Table 7.2: Number of patterns produced by the Concurrent and Consecutive algorithms, varying ε .

Details of these datasets are available in Appendix A. We compare two variants of CONCURRENT, viz., optimizing coverage (denoted as CONCURRENT_{cov}) and optimizing spuriousness (denoted as CONCURRENT_{spur}), and two consecutive algorithms, viz., *Select-Fuse* and *Fuse-Select*. For simplicity, we set $\varepsilon_r = \varepsilon_s$, and mention them as ε in the plots.

7.5.1 Effectiveness

First, we present the number of patterns returned by CONCURRENT and the consecutive algorithms in Tables 7.1 and 7.2. These two tables present the summary size produced by each algorithm in various settings. The rows “(% of Input)” and “(% of Consecutive)” present the percentage (ratio) of the number of the produced patterns with the input size, i.e., the number of CFIs, and with the best consecutive algorithm, respectively. Some values for *Fuse-Select* are missing because the program crashes due to allocating too much memory. The observations are shown in tables instead of plots because we want to show three numbers, i.e., the absolute value, the relative

size to the input, and the relative size to the consecutive approach.

Comparing the two consecutive approaches, we see that *Select-Fuse* always performs better than *Fuse-Select*. This is somewhat expected, as *Fuse-Select* performs too many unnecessary fusions. Therefore, we compare CONCURRENT only with the *Select-Fuse* algorithm, and observe that CONCURRENT always performs better. We can also see that the difference is quite significant in some cases, i.e., CONCURRENT_{cov} produces up to 38% less patterns than *Select-Fuse*. We can also see that the differences are not significant with small ε . This is because when ε is low, not many fusions and selections can be done, and hence, the performances of all algorithms are similar.

Between the two variants of CONCURRENT, we see that CONCURRENT_{cov} is always better than CONCURRENT_{spur} . Analyzing the algorithm, we find that CONCURRENT_{cov} tends to fuse two distinctive patterns, as far as they are still within the spuriousness threshold. This strategy allows more patterns to be covered in each iteration, and hence results in less summary patterns. Some exceptions occur when $\varepsilon = 0.1$, however, the difference is insignificant.

Analyzing the scalability, we find that all algorithms share similar characteristics. Varying σ , we can see that lower minimum support produces more patterns, but smaller ratio to the inputs. This is because with lower σ , we have more patterns as input. Moreover, high support itemsets tend to be more distinctive to each other, and hence are more difficult to summarize. Varying ε , we see an obvious trend, i.e., higher ε allows more effective summarization.

7.5.2 Efficiency

Next, we analyze the efficiency of the algorithms. The results of this experiment are depicted in Figure 7.3. We can see that *Fuse-Select* is by far the slowest, while the other three, viz., *Select-Fuse* and the two variants of CONCURRENT are comparable. This is expected as these three algorithms have time complexity of $O(N^2C)$, while *Fuse-Select* has time complexity of $O(N^2C \lg N)$. Note that the second phase of *Select-Fuse* is not significant, as the input (the result of phase one) size is very small, and only few remaining patterns can be summarized, yielding few number of iterations.

Analyzing the scalability, we see that the time taken by each algorithm is inverse-proportional to the minimum support σ , and proportional to the tolerance ε . These observations are as expected, because lower σ means more inputs, and higher ε means more iterations. It is also worth noting that the running time of the algorithms is not

Figure 7.3: Efficiency of concurrent and consecutive algorithms, varying σ and ϵ .

very sensitive to ϵ .

Overall, we conclude that CONCURRENT is reasonably efficient in real-world datasets with practical settings.

7.6 Concluding Remarks

In this work, we tackled the problem to summarize interesting itemsets, where the knowledge is formed as itemsets and their associated transactions (subspace cluster). We observed that all previous approaches produce summaries with either perfect precision or perfect recall. We argued that such a perfect nature is not required in many real world applications, and then proposed a summarization problem that compromises both precision and recall, to boost its conciseness.

One way to compromise both properties is by considering each property separately, which can be done by executing existing algorithms consecutively. However, this approach is not effective. We then proposed a novel algorithm that concurrently considers both properties. We considered two algorithm variants, by greedily optimizing the coverage or the spuriousness of the summary pattern in each iteration.

Our algorithm takes the interesting itemsets as input, and runs in quadratic time to the number of input patterns.

Extensive empirical evaluations showed that the concurrent algorithm is much more effective than the consecutive algorithm. Between our algorithms, we observed that optimizing coverage always leads to a more concise summary.

Chapter 8

Summarizing Itemsets for Class Description

This chapter discusses the problem to summarize interesting itemsets, where the knowledge is formed as itemsets (feature-sets) representing objects in a class. A set of objects (transactions) can be described by a set of features (itemset). We are interested in itemsets discriminating those which are commonly satisfied by some set of objects, and not satisfied by other set of objects. The objective of this problem is to find a small set of itemsets that accurately describe all objects in a class. A description is accurate if it describes all (or most) objects in the target class, and none (or few) in the non-target class.

In natural language, a simple sentence, formed as a conjunction or disjunction of terms (features), can be used to describe a concept. In this work, we generalize this definition by allowing a sentence to describe an object, if it satisfies at least k terms in the sentence. Interestingly, this small modification allows a much more concise summary, without sacrificing its accuracy and interpretability. As a summary describes an object if any of its sentence does, we can interpret the summary by reading each sentence independently. Therefore, it satisfies our interpretability requirement (c.f., Chapter 1).

8.1 Preliminaries

The *class description* problem requires describing a class accurately and in an interpretable manner, presenting interesting general properties of the class [35]. For

example, we can describe mammals in an animal database as animals¹ that produce milk. This description is ideal since it is concise (consists of only one sentence, hence easily interpretable), and accurate (descriptive), i.e., all mammals produce milk (characterizing) and all animals that produce milk are mammals (discriminating). We can treat this description as an interesting itemset, where the item represents a property “produces milk”.

A simple, yet accurate description of a class is important and useful knowledge for data analysts. Simple descriptions can be easily understood by end-users and can also be used as intermediate results on which more advanced data analysis techniques, e.g., OLAP [25, 83] may be applied.

We may treat a description as a summary of all patterns/sentences that describe the concept. Here, our objective is to find the best set of sentences, which can accurately describe the class. Our objective in one sense is similar to classification, however, here we focus more on the conciseness of the classifier rather than the accuracy and generality.

Relating to our summarization framework (Section 1.3), we can treat the properties of objects in a concept (and those not in a concept) as the interesting itemsets, or the knowledge which we want to summarize.

8.2 Summary Form

The early solutions of this problem [18, 36] consider each attribute separately. Relating with the notion of *sentences*, these algorithms restrict each sentence to contain only one term. While these approaches are interpretable, their conciseness and accuracy are limited since they do not consider combinations of more than one attribute.

The class description problem can also be addressed using classification techniques. That is, we treat the produced classifier as the description. However, most classifier-based descriptions are not interpretable. For instance, many classification algorithms such as SVM and KNN produce non-axes-parallel, or even non-linear decision boundaries between classes. Such representations are very hard to explain using natural language [35], and hence are not semantically meaningful. Rule-based classifiers such as decision trees [80] and associative classifiers [50] produce axes-parallel decision boundaries, but they tend to produce a large number of rules to achieve the desired accuracy, and hence are also not concise. Consider the following two concepts described by simple sentences:

¹specifically, females of the species

Example 8.1. *Iris versicolor* is a species of iris flower that satisfies any two properties among: petal length between 3 and 4.9 cm, petal width between 1 and 1.6 cm, sepal length equals to 5.9 cm, and sepal width equals to 2.7 cm. This rule achieves 98% recall and 98% precision according to the *Iris* dataset in the UCI machine learning repository.

Example 8.2. Students doing well in at least three out of the four courses: “data structure”, “algorithm”, “artificial intelligence”, and “database” will receive high grades in “data mining” [69].

While these two concepts are simple (can be explained by one sentence of natural language), we need multiple rules to explain them using conventional rule-based classifiers. A conventional rule is comprised of a conjunction of terms – where each term is a constraint on an attribute. So we can see that concept represented in Example 8.1 requires six conventional rules/sentences (each comprising of two terms), while that in Example 8.2 requires four rules/sentences to describe them.

Based on the above discussion, we propose to extend the conventional sentence with a threshold parameter k , specifying the number of terms required to describe an object. In other words, we allow the object to be partially covered by the subspace. This idea is inspired from the usage of fault-tolerant itemsets (c.f., Chapter 4) and partial-supports (c.f., Section 2.3.1). This simple enhancement, surprisingly, can boost the effectiveness of a description, as illustrated in the above examples.

Having the form, the next step is to design an efficient algorithm to produce an effective and accurate summary.

8.3 Notations and Problem Definition

This chapter deals with a database that contains a set of objects O and attributes A . As our objective is to describe a class, we require all objects to also have a class label. The value of an attribute $a \in A$ of an object $o \in O$ is denoted as $a(o)$. The set of objects belonging to class l is denoted as pos , and the set of objects not belonging to class l is denoted as neg . The problem we tackle is to describe the target class l as concisely and accurately as possible. In brief, a *description* consists of several *sentences*, and each sentence contains several *terms*. For better presentation, we formally define each notation in a bottom-up manner.

Definition 8.1 (Term, describe/cover). A term t_V^a is formed as a pair of an attribute a and its respective continuous range (for numerical attribute) or set (for categorical

attribute) of values V . A term *describes* all objects whose values are in the set, or within the range, of the corresponding attribute, i.e., $t \Rightarrow o \equiv a(o) \in V$.

Example of terms are: “*petal length* = [3, 4.9]” (from Example 8.1), “*database* = {GOOD}” (from Example 8.2), or “*odor* = {ALMOND, ANISE, NONE}” (from the *mushroom* database). Relating with interesting itemsets terminology, a term can be treated as an item, which is a constraint of an attribute. The notion of *describes* can also be interpreted as *covers*, *contains*, or *is satisfied by*.

Definition 8.2 (Sentence). A sentence s_X^k is formed as a pair of a threshold k and a set of terms $X = \{t_1, t_2, \dots\}$ (read: k -of- X). Sentence s describes objects if any k of its terms do, i.e., $s \Rightarrow o \equiv (|\{t \in X | t \Rightarrow o\}| \geq k)$.

From the itemset-mining point of view, a sentence is essentially an itemset. The parameter k specifies the required number of items for a transaction to support an itemset, which is omitted when $k = |X|$. The integration of this parameter follows the concept of fault-tolerant itemsets and partial supports of itemsets (Chapters 4 and 2 respectively). This notion of sentence subsumes the usual definitions, where the sentence is defined as a conjunction of items ($k = |X|$), or disjunction of items ($k = 1$). For the sake of interpretability, we require all terms in one sentence to have a distinct attribute. The objective is to prevent a case as follows.

Example 8.3. Consider a sentence $s = 2\text{-of-}\{A = \{a, b\}, A = \{a, c\}, B = 3\}$. This sentence covers object o with $(A(o) = \{a\})$, or $((A(o) = b \text{ or } c) \text{ and } B(o) = 3)$. Despite being concise, this sentence is confusing. It is better to split this sentence into two sentences as $\{A = \{a\}\}$ and $\{A = \{b, c\}, B = 3\}$.

Finally, a description describes an object if any of its sentences do.

Definition 8.3 (Description/Summary). A description or summary R is formed as a set of sentences $R = \{s_1, s_2, \dots\}$, and it describes objects which are covered by any of the sentences, i.e., $R \Rightarrow o \equiv \{o | \exists s \in R, s \Rightarrow o\}$.

As a summary describes an object if any of its sentence does, we can interpret the summary by reading each sentence independently.

We now define the accuracy measure of a description. As in previous chapters, we measure the accuracy of a summary using F_1 . However, we add a parameter γ to weigh the negative instances for handling imbalanced classes. For example, suppose we want to describe a class that exists in 90% of the database. In this case, a description that simply covers all objects will have F_1 as high as 0.95, which is not

Algorithm 8.1: PUB algorithm

Input: K : number of sentences.

```

1  $description \leftarrow \emptyset$ ;
2 for  $i \leftarrow K$  down to 1 do
3    $s \leftarrow findBestSentence(\beta = 1/i)$ ;
4   if  $s = \emptyset$  then break;
5    $description \leftarrow description \cup s$ ;
6 return  $description$ ;

```

desirable. With $\gamma = 9$ (to balance the class), the same description has F_1 measure of 0.67.

While building the intermediate results, we also use F_β measure, where β controls the importance of recall over precision. That is, $\beta = 2$ means we weigh recall twice as much as precision (recall is more important), and $\beta = 0.5$ means we weigh precision twice as much as recall. The formula of F_β is given as follows:

$$F_\beta = \frac{(1 + \beta^2) \text{precision} \times \text{recall}}{\beta^2 \text{precision} + \text{recall}}$$

Our objective is to produce an interpretable and accurate description of a class. As our summary form is interpretable, we only need to consider the conciseness, which is determined using the number of sentences in the summary. Therefore, the class description problem can be formalized as follows.

Definition 8.4 (Class Description Problem). Given a binary labeled database and the maximum number of sentences K , return a summary or description R that maximizes the summary quality $\nu(R)$, where $\nu(R) = -\infty$ if $|R| > K$, and $\nu(R) = F_1$ otherwise. In other words, we should return a summary with the highest F_1 .

8.4 Algorithm

To solve this problem, we propose the PUB algorithm, which is an acronym for Partial coverage of SUBspace. This algorithm consists of several modules, as discussed in following subsections.

Algorithm 8.2: Function *findBestSentence*

Input: β ,
 w : maximum number of terms per sentence.
 b : number of considered sentences per level.

```

1 candidates  $\leftarrow \{\emptyset\}$ ;
2 Result  $\leftarrow \emptyset$ ;
3 for  $i \leftarrow 1$  to  $w$  do
4   newCandidates  $\leftarrow \emptyset$ ;
5   for  $s \in \textit{candidates}$  do
6     extend( $s$ ) with all possible attributes and  $k$ , and store in
       newCandidates;
7   candidates  $\leftarrow \textit{newCandidates}$ ;
8   find the best candidate, and update Result if necessary;
9   prune candidates by taking the top- $b$   $F_\beta$ ;
10 return Result;
```

8.4.1 Main Method

The main module of PUB is presented in Algorithm 8.1. This module iteratively finds the best sentence. We set $\beta = 1/i$ in each iteration, where i is the remaining number of sentences (line 3). In other words, we weigh precision i times more than recall. The rationale is that high recall is not very important when we still have sentences available, since the remaining objects can be covered by the remaining sentences.

8.4.2 Finding the Best Sentence

For finding the best sentence, PUB uses a bounded-breadth-first-search strategy, as illustrated in Algorithm 8.2. PUB searches the sentence in a level-wise order, from the one with 0 term, 1 term, 2 terms, etc. Besides β , this function needs two additional parameters, b and w . Parameter b is used to bound the search space, by keeping only b *candidates* (sentences to be extended) per level. This parameter provides the usual trade-off between accuracy and efficiency, i.e., with larger b , the descriptor becomes more accurate, but the algorithm becomes less efficient. Parameter w specifies the maximum number of terms in a sentence. This parameter trades-off accuracy for efficiency and interpretability, i.e., sentences with more terms are (naturally) less interpretable. Some pruning techniques are also integrated, i.e., PUB only keeps potential candidates. A candidate is not potential if both its true positives and negatives are subsumed by other candidates.

Figure 8.1: Plots of x (true positives) vs. y (false positives) for various values of F_β .

The resulting sentence is the one with the best F_β . Note that this sentence might contain less than w terms.

8.4.3 Finding Optimal Values

In extending an attribute (term) to a sentence, PUB finds the *best* range/set of values of the particular attribute for all possible values of k , where the notion of *best* is based on F_β . For this purpose, we able to design an algorithm with linear time complexity based on the following theoretical foundation.

Let x be the true positives, y be the true negatives², then the formula of F_β can be simplified into:

$$F_\beta = \frac{(\beta^2 + 1) x}{\beta^2 |pos| + x + \gamma y} \quad (8.1)$$

Isolating elements containing y and x , we have:

$$y = m x + c \quad (8.2)$$

where

$$m = \frac{\beta^2 + 1 - F_\beta}{\gamma F_\beta} \text{ and } c = -\frac{\beta^2 |pos|}{\gamma}.$$

Observe that Equation (8.2) is a linear equation, where the gradient is a function of F_β , i.e., larger F_β implies lower gradient, as illustrated in Figure 8.1. A negative value in the y -axis means that with a given x (true positives), the specified F_β can never be achieved. One important implication of Equation (8.2) is stated as follows.

²We use x and y for better understanding, since we relate the terms with Cartesian coordinates.

Property 8.1. F_β can be increased by adding (or removing) a point (or vector) (x_1, y_1) , with y_1/x_1 lesser (or greater in case of removing) than the current gradient.

Calculating x and y of each value

Before running the algorithm, we first need to calculate x and y contributed by each value $v \in V$. Note that these values are *not* independent to the current (to be extended) sentence s and the threshold k . Let the extending term be t_v . We can observe that if an object o is covered by s_X^k , then it will also be covered by $s_{X \cup t_v}^k$. In other words, these objects always contribute to the result-set, regardless of the chosen value. We denote the number of positive and negative instances in this set as x_0 and y_0 . On the other hand, if an object is *not* covered by s_X^{k-1} , then it will also not be covered by $s_{X \cup t_v}^k$. Hence, we can ignore those objects. Lastly, if an object o is covered by *exactly* $k-1$ terms in s , this object will be covered by $s_{X \cup t_v}^k$ if and only if $o \in t_v$. As this object is covered by $s_{X \cup t_v}^k$ but not by s_X^k , this object reflects the impact of appending term t_v to the sentence. The number of such positive and negative objects are denoted as v_x and v_y respectively. Let the set of chosen values be V , then we have $x = x_0 + \sum_{v \in V} v_x$, and similarly for y . The objective is then to find the values V , to maximize $F_\beta(x, y)$.

Accumulating these values needs one pass through database, which takes $O(|O|)$ time complexity.

Categorical Attributes

Our strategy to find the optimal (set of) values is by systematically increasing the current F_β , until we reach the global optimal. Let V be a set of distinct values for a given categorical attribute. The obvious solution to find the optimal values is by trying all possible subsets of V with time complexity $2^{|V|}$. Here, we show an algorithm to solve the problem with time complexity $O(|V| \lg(|V|))$.

From Property 8.1, we can increase F_β by taking any value v with $v_y/v_x < m$, where m is the gradient as in Equation 8.2. It is then intuitive that repeatedly taking values with minimum v_y/v_x produces the optimal result. This algorithm is presented in Algorithm 8.3, and the proof of optimality is given as follows.

Lemma 8.2. *Algorithm 8.3 produces optimal result, i.e., the set of values that produces the highest F_β .*

Proof. Let the current gradient be m . Consider two values $a, b \in V$, with $a_y/a_x < b_y/b_x < m$. From *cosine law*, if we add b to the current values, the resulting gradient

Algorithm 8.3: Selecting optimal values for categorical attributes.

```

1 Sort the values  $V$  in ascending order of  $y/x$ ;
2  $x \leftarrow 0$ ;
3  $y \leftarrow 0$ ;
4  $Result = \emptyset$ ;
5 for  $v \in V$  do
6   if  $F_\beta(x + x_v, y + y_v) > F_\beta(x, y)$  then
7      $x \leftarrow x + x_v$ ;
8      $y \leftarrow y + y_v$ ;
9      $Result \leftarrow Result \cup \{v\}$ ;
10  else break;

```

will be between m and b_y/b_x , which is always greater than a_y/a_x . Therefore, we can always add a after adding b to achieve higher F_β . On the other hand, it is possible that we first add a , but adding b afterward decreases F_β , implying that the optimal solution is only a . As there is no benefit to add b before a , algorithm 8.3 is optimal. $\square$

Analyzing the complexity, we need to sort the values (based on y/x) then iterate them. Therefore, the time complexity is $O(|V| \lg |V|)$. Since $|V| \ll |O|$, we can consider that the time taken by this procedure is not significant, compared to accumulating the values (Section 8.4.3).

Numerical Attributes

The difference between categorical and numerical attributes is that every value in a categorical attribute can be taken arbitrarily, while the values in a numerical attribute must be within a continuous range. The naïve algorithm for dealing with numerical attributes is to try all possible ranges with time complexity $O(|V|^2)$. Note that V is a set of distinct values of the specified attribute in the database, which can be as large as $|O|$ in the case of numerical attributes.

For easier presentation, we use x_i and y_i as the number of false positives and negatives for objects with value v_i , instead of v_{i_x} and v_{i_y} . The range of values bounded by values v_i (exclusive) and v_j (inclusive) is denoted as $(i, j], i < j$. The F_β of objects in the range $(i, j]$ is denoted as $F(i, j)$. We define another notation, x'_k (and y'_k) that stores the cumulative number of true positives for all values less or equal to v_k , i.e., $x'_k = \sum_{i \leq k} x_i$. Having x'_k , the number of true positives in a range $(i, j]$ can be calculated as $x'_j - x'_i$.

Algorithm 8.4: Selecting optimal values for numerical attributes.

```

1  $C \leftarrow \{(0, 0)\};$ 
2  $Result \leftarrow 0;$ 
3 for  $i \leftarrow 1$  to  $|V|$  do
4    $(x'_a, y'_a) \leftarrow findBest(C, Result);$ 
5   if  $F_\beta(x'_i - x'_a, y'_i - y'_a) > Result$  then
6      $Result \leftarrow F_\beta(x'_i - x'_a, y'_i - y'_a);$ 
7    $add(C, (x'_i, y'_i));$ 
8    $filter(C);$ 

```

Figure 8.2: Finding maximum y -intercept through a set of points, given a gradient.

Our algorithm in a high level is presented in Algorithm 8.4. The set C stores all values that *potentially* form (become the left-boundary of) the optimal F_β . For every v_i (right-boundary), we find the best left-boundary in C that produces the maximum F_β (Line 4). Next, we *add* the current value to C (Line 8), and then filter the set (Line 9). The challenge is then to efficiently implement the *findBest* and *filter* functions. The naïve algorithm for *findBest* iterates C , resulting in $O(|V|^2)$ time complexity.

Relating with Equation (8.2), we can treat $(x'_i + x_0, y'_i + y_0)$ as the *current* point, and (x'_a, y'_a) as the vector to move the point. From Property 8.1, removing x'_a, y'_a increases F_β if and only if $y'_a/x'_a > m$, or equivalently $y'_a - m x'_a > 0$. Therefore, finding v_a that can increase F_β can be done by calculating the maximum $y'_a - m x'_a$.

Let $f(A, m) = A_y - m A_x$ for a point A . This equation can be restructured into $A_y = m A_x + f(A, m)$, which forms a linear equation with gradient m and y -intercept $f(A, m)$. Since we want to maximize $f(A, m)$, the problem can be reformulated into: given a set of points and gradients, find the highest y -intercept for each gradient that passes through at least one point. This problem is illustrated in Figure 8.2 where we have 5 points A, B, C, D, E . When the input gradient is m_1 , the result is p that

8.4. ALGORITHM

169

passes through point B . Similarly, when the input gradient is m_2 , the result is q that passes through point E .

First, we analyze the points that potentially form the best left-boundary, for any given gradient.

Lemma 8.3. *Points not in the convex hull can never produce the highest y -intercept.*

For example, point D in Figure 8.2 can never produce the highest y -intercept, regardless the input gradient. Having C , we then need to find the point that produces the maximum y -intercept, given the gradient.

Lemma 8.4. *Given a gradient, the y -intercept of points in the convex hull is unimodal.*

Because of the unimodal property, the maxima is the first point (value) v_a with $f(v_a, m) > f(v_{a+1}, m)$. Note that we only consider points in the convex hull. Having found the maximum y -intercept, we might be able to increase F_β , and equivalently reduce the gradient. We then need to recalculate the optimal point v_a , with respect to the new gradient.

Lemma 8.5. *If $A_x < B_x$, and both A and B are part of convex hull, $f(A, m) - f(B, m)$ grow monotonically with m .*

For example, in Figure 8.2, we have $f(A, m_1) - f(B, m_1) < f(A, m_2) - f(B, m_2)$. Lemma 8.5 shows that, if $f(A, m) < f(B, m)$ for any value of m , then point A can never produce the highest y -intercept, since the gradient always decreases.

The proofs of Lemma 8.3 to 8.5 are straight-forward from the definition of convex hull.

Now we have found the optimal left-boundary, with respect to right-boundary v_i . The remaining problem is to efficiently recalculate the optimal left-boundary after shifting the right-boundary to v_{i+1} .

Lemma 8.6. *If $F(i, k) \leq F(j, k)$ and $i < j$, then $F(i, k') \leq F(j, k')$ for all $k' > k$ with $F(i, k') \geq F(i, k)$.*

Proof. Can be proven using *sine-law*. □

Lemma 8.6 finishes-off the problem, by stating that the optimal left-boundary never decreases. Whenever we increase the right-boundary, we can iteratively increase the previous left-boundary until we hit the maxima (because of the unimodal property). Pseudo-code for the two functions are presented in Algorithm 8.5 and Algorithm 8.6. Since every point is removed at most once, the time complexity of Algorithm 8.4 is linear.

Algorithm 8.5: Function *findBest*

Input: C, F_β

```

1 repeat
2    $m \leftarrow \text{calculateGradient}(F_\beta);$ 
3   while ( $|C| > 1$ ) and ( $\frac{y_{C_2} - y_{C_1}}{x_{C_2} - x_{C_1}} > m$ ) do
4      $\perp$   $\text{remove}(C_1);$ 
5     recalculate  $F_\beta;$ 
6 until  $F_\beta$  does not change;
7 return  $C_1;$ 

```

Algorithm 8.6: Function *filter*

Input: C

```

1 while ( $|C| > 2$ ) and ( $C_{|C|-1}$  not in convex hull) do
2    $\perp$   $\text{remove}(C_{|C|-1});$ 

```

8.4.4 Complexity Analysis

This section analyzes the efficiency of PUB. The main method (Algorithm 8.1) performs K iterations of finding the best rule. Hence, it contributes to the overall complexity by a factor of K . For finding the best sentences (Algorithm 8.2), PUB extends the sentence w times, while in each iteration, there are at most b sentences to be extended. Hence, the complexity is multiplied by $w b$.

Finding the optimal values for each attribute needs $O(|O|)$ complexity, as explained in Section 8.4.3. As we find the optimal extensions for all possible k , and we need to calculate the F_1 for each extension, for all k , the time complexity becomes $O(w^2 |A| |O|)$.

The total time complexity is the multiplication of all those complexities, giving $O(n b w^3 |A| |O|)$.

8.5 Experimental Results

We evaluate our algorithm using various datasets from the UCI machine learning repository, including *easy* and *difficult* datasets, numerical and categorical attributes, balanced and imbalanced classes, and number of attributes ranging from 4 to 166.

Overall three experiments were conducted. The first shows the benefit of adding a threshold k in a sentence. Next, we compare the effectiveness of PUB with $C4.5$ algorithm [80], which is a famous decision tree-based classifier. We do not compare

8.5. EXPERIMENTAL RESULTS

171

Dataset	Class Label (frequency)	$F_1(K = 1)$		$F_1(K = 2)$		$F_1(K = 3)$	
		Partial	All	Partial	All	Partial	All
Iris $ A = 4$, $ O = 150$	<i>I. setosa</i> (33%)	1.0	1.0	1.0	1.0	1.0	1.0
	<i>I. virginica</i> (33%)	0.9852	0.9608	0.9901	0.98	1.0	0.98
	<i>I. versicolor</i> (33%)	0.9849	0.9751	1.0	0.9899	1.0	0.9899
Mushroom $ A = 22$, $ O = 8, 124$	Edible (52%)	0.9976	0.9956	0.9995	0.9929	0.9993	0.9994
	Poisonous (48%)	0.9996	0.9844	1.0	0.998	1.0	0.9990
Glass $ A = 9$, $ O = 214$	Class 1 (33%)	0.9381	0.8884	0.9796	0.9158	1.0	0.8874
	Class 2 (36%)	0.856	0.7498	0.9721	0.8224	1.0	0.8594
	Class 7 (14%)	1.0	0.9718	1.0	0.9825	1.0	1.0
Wine $ A = 13$, $ O = 178$	Class 1 (33%)	1.0	1.0	1.0	1.0	1.0	0.9915
	Class 2 (40%)	1.0	0.435	1.0	0.971	1.0	0.9784
	Class 3 (27%)	1.0	1.0	1.0	1.0	1.0	1.0
Abalone $ A = 8$, $ O = 4, 177$	Class 8 (14%)	0.758	0.7489	0.7703	0.7493	0.771	0.7493
	Class 9 (17%)	0.729	0.7125	0.7409	0.7126	0.7491	0.713
	Class 10 (15%)	0.7289	0.7047	0.7446	0.727	0.7425	0.7246
	Class 11 (12%)	0.7389	0.7203	0.7517	0.7301	0.7545	0.7295
Adult $ A = 14$, $ O = 32, 561$	$> 50K$ (76%)	0.8245	0.7919	0.8311	0.7938	0.83	0.8163
	$\leq 50K$ (24%)	0.8102	0.7566	0.8171	0.8046	0.8201	0.7915
WDBC $ A = 30$, $ O = 569$	Benign (63%)	0.9916	0.9534	0.9976	0.968	1.0	0.9644
	Malignant (37%)	0.9911	0.947	1.0	0.9692	1.0	0.9695
Clean2 $ A = 166$, $ O = 6, 598$	Class 0 (85%)	0.888	0.802	0.9308	0.8666	0.947	0.8755
	Class 1 (15%)	0.9139	0.8706	0.952	0.8992	0.9554	0.9251
Average		0.9112	0.8799	0.9275	0.8987	0.9319	0.9021

Table 8.1: Impact of allowing partial coverage for concept description.

with rule-based classifier (CBA) [50] because of the burden in setting the appropriate parameters and discretization. Moreover, as CBA produces conventional sentences (rules), the result of CBA can also be discovered by PUB. Lastly, we evaluate the sensitivity of parameters, viz., the maximum candidates per level and the number of terms (see Algorithms 8.1 and 8.2), to the efficiency and effectiveness of PUB. We also conduct a case study for the descriptions produced by PUB. To balance the class, we set parameter $\gamma = |pos|/|neg|$ (see Section 8.3). With this setting, the total weight of positive instances equals that of negative instances.

8.5.1 Impact of Partial Coverage

This section analyzes the impact of allowing partial coverage of sentences. We describe all classes in the input database whose frequency is greater than 10%, and the results are presented in Table 8.1. The accuracy of descriptions that allow partial coverage is mentioned in column *Partial*, while that of descriptions that require total coverage is mentioned in column *All*. Both algorithms use parameters $b = 50$ and $w = 10$. These parameters are chosen to allow efficient computation, and produce descriptions with high accuracy. More analyses on sensitivity of parameters are presented in Section 8.5.3.

From the table, it is clear that allowing partial coverage can boost the effectiveness of a description. The improvement is up to 10% in *glass* and *clean2*, even though in some others it is marginal. It is interesting that on average, one sentence that allows partial coverage is more effective than three conventional sentences. Using *t-test* significance test, the *p-value* is as low as 0.003.

In most cases, the information gained by using multiple sentences is minimal. This shows that most real-world concepts are simple enough to be described with one sentence. Allowing multiple sentences also significantly reduces the interpretability, as will further be discussed in our case study in Section 8.5.4.

The fact that some descriptions with more sentences are less accurate than those with lesser sentences (e.g., *mushroom* and *adult*) is due to the greedy and heuristic nature of PUB. However, as will be explained in Section 8.5.2, despite not always producing the optimal sentences, PUB is more comprehensive than previous techniques (C4.5), hence the resulting descriptions are also more accurate.

8.5.2 Comparison with C4.5

We implemented C4.5 ourselves because we need to weigh the negative instances, as well as calculate F_1 for the specified number of rules. In our implementation, we store each leaf-node in the priority-queue, and iteratively split the *most impure* node. The impurity of a node is measured as the (absolute weighted) number of instances in the node, which does not belong to the class specified by that node. For example, the impurity of a node containing 10 instances of class *A* and 5 instances of class *B* is 5. We repeatedly split the nodes until there is no split that produces positive *relative information gain*. As our objective is to describe a class, we first relabel the database into two classes, target class and non-target class. The number of rules (sentences) is then calculated as the number of leaves that belong to the target class.

8.5. EXPERIMENTAL RESULTS

173

Figure 8.3: F_1 measure of PUB and C4.5 algorithms.

The parameters of PUB are the same as in the previous experiment.

In this experiment, we use the same datasets as in previous experiments, and the results are depicted in Figure 8.3. We can see that PUB needs less sentences to produce higher quality descriptions. Even though C4.5 seems to be better in Figures 8.3(e) (and f), it needs 27 sentences to even match the quality of PUB with 2 sentences. While our sentences are more complex than the one produced by C4.5, from the natural language point of view, both are equally easy to interpret. In Figure 8.3(e), we see that the accuracy of C4.5 sometimes decreases as the number of sentences increases (from 1–2 and 8–9). This weird behavior is because C4.5 optimizes the information gain per split, while our measure of accuracy is F_1 .

Besides the addition of parameter k , another difference between PUB and C4.5 is that the space traversed by PUB to find the optimal description is much more than

Figure 8.4: Sensitivity of PUB with respect to b and w .

that by $C4.5$. $C4.5$ greedily splits a node with the value which produces the highest relative information gain, while PUB keeps the top- b sentences to be extended later. Therefore, it is very likely that $C4.5$ is trapped in a local optima, while PUB can find the global optima.

There are also differences in the splitting procedure. $C4.5$ splits categorical attributes with *all* values in that attribute, while PUB efficiently seeks the subspace which can optimally (and efficiently) divide the space into two. Due to the binary splitting, PUB can combine several values with similar behavior, and therefore produce less sentences (more interpretable). We have tried to perform binary splitting with $C4.5$ (split per value), but the result is much worse. In numerical attributes, $C4.5$ finds the optimal splitting point, while PUB finds the optimal range (two splitting points) with identical time complexity.

From the above discussion, it is not surprising that PUB is significantly better than $C4.5$ for the class description problem.

8.5.3 Sensitivity to Parameters

Next, we analyze the sensitivity of parameters with respect to efficiency and accuracy, which are plotted in Figure 8.4. In this experiment, the number of sentences is fixed

to one.

In our analysis in Section 8.4.4, the time complexity is cubic with respect to w , and linear with respect to b , which is as depicted in the figures. Regarding accuracy, we first see that the accuracy is not sensitive to b . Therefore, we recommend using a sufficiently low value for b (5 – 10). We use 50 in our previous experiments since the efficiency is linear.

We can also see that the accuracy grows proportional to w , though the growth rate diminishes while w increases. Moreover, larger w leads to less interpretable result (more attributes). Therefore, we suggest to set w as the highest interpretable value (depending on the user), while incorporating another stopping criterion, to stop extending the sentence if the improvement is not significant.

From a practical point of view, we can see that PUB is quite efficient and effective in most cases. Even in the largest dataset (*Clean2*), we can construct a reasonably good description ($F_1 = 0.9$) in less than one minute. Nonetheless, if efficiency is the main concern, we can always perform extra pruning, for example, by only extending with some attributes, or finding the best values of some particular k .

8.5.4 Case Study

We perform our case study on the *glass* database. This database contains seven types of glasses with 9 attributes, representing the *refractive index* (RI) and chemical composition of each glass. In this study, we analyze *Class 2* glass, which is a “*building window glass with no float processed*”. This class is the most frequent class (around 35%) in the database. To balance the class, we set $\gamma \approx 35/65 = 0.55$.

First, we analyze individual attributes, and find that no attribute is descriptive. The most descriptive term is “ $Na \leq 13.98$ ” with F_1 of 0.73. Note that a description that simply covers all objects has F_1 of 0.67. The case doesn’t get better even if we allow multiple terms. The best sentence we found to describe this class is $\{RI \geq 1.52, Na \leq 13.98, Mg \leq 3.98, Al = [0.56, 2.12], K \leq 0.81, Ca \geq 7.83\}$ with F_1 of 0.75.

Interestingly, the accuracy improves significantly if we allow the objects to be partially covered by the subspace. With 2 terms, PUB finds sentence 1-of- $\{K = [0.33, 1.1], Ca \geq 10.79\}$, with F_1 of 0.77, i.e., better than the best of conventional technique. Using 5 terms, PUB finds sentence 2-of- $\{RI \geq 1.52, Mg = [2.72, 3.15], K = [0.33, 1.1], Ca = [7.08, 8.52], Fe = [0.08, 0.24]\}$, with F_1 of 0.82. Finally, with all attributes (4-of-9) we can get F_1 as high as 0.87 with single sentence. Note that the interpretability does not worsen with the addition of the threshold parameter.

For example, our sentence with 5 terms means, all glasses in this class are likely to satisfy at least two of the terms (confidence = 0.91), and any glass satisfying at least two terms are likely to be in this class (confidence = 0.75). This knowledge for this database is novel, and can be used as a simple description of this class.

Observe that the best range for Ca is different in all those descriptions. One possible reason is that this class consists of several concepts. Setting PUB to produce two sentences with at most 5 terms each, we can find a description with $F_1 = 0.95$. The first sentence is: 2-of- $\{RI = [1.516, 1.517], Mg = [2.76, 3.25], Si \leq 72.57, Ca = [7.83, 8.27], Fe = [0.12, 0.35]\}$ with precision = 0.93 and recall = 0.75, and the second sentence is: 3-of- $\{RI = [1.518, 1.519], Na = [11.02, 14.43], Al = [0.56, 1.43], Si = 0.75, and Ca = [11.14, 14.96]\}$, with precision of 0.92 and recall of 0.33. The fact that the first sentence has much greater recall is due to the greedy nature of PUB that iteratively finds the best sentence.

Such a boost of accuracy conforms that this class is separated into several concepts, for example: glasses with Ca around eight, and glasses with Ca between eleven and fifteen. However, this description becomes much more difficult to interpret. For example, when there are overlapping subspaces, the impact of the attribute becomes unclear, since it appears in two different sentences that explain two different concepts.

8.6 Related Works

As the existing solutions to our problem has been discussed in Section 8.2, here we discuss other problems, which are related to the class description problem.

Our objective is to describe a class. From this point of view, this problem is seemingly related to the *conceptual clustering* problem [26, 55]. Unlike the traditional clustering problem, conceptual clustering also produces the description for each discovered cluster. The main difference between the class description problem and the conceptual clustering problem is that conceptual clustering deals with unlabeled data, while the class description problem deals with labeled data. Therefore, algorithms for conceptual clustering are not applicable for class description

Considering the sentence form, our proposal defines a sentence as a combination of terms with a threshold k , specifying the minimum number of terms to be satisfied by an object, to be considered as covered by a sentence. Having the form, the next step is to find the best description, which is a set of sentences. From this point of view, our problem is similar to the boolean expression mining [110]. The objective of [110] is to find all interesting (frequent) boolean expressions of items. Another related

work is the redescription mining problem [60, 109]. That is, given a set of objects (or even many sets of objects), the redescription mining problem finds all combinations of features (attributes) which describes the given objects.

There are many differences which make these works are not comparable with ours. Firstly, they restrict the database to be transactional (binary), while we allow the general categorical and numerical databases. Secondly, their outputs are all interesting sets, which are interesting by itself. Whereas, our output (summary) is a set of sentences which are interesting as a whole. Thirdly, each of their output-pattern is in the form of Boolean expressions, while our output is in the form of itemset with a parameter k , which is a combination of many Boolean expressions.

8.7 Concluding Remarks

We tackled a problem to describe a class in a database. A description consists of a set of sentences, where each sentence consists of a set of terms. Usually, a sentence is formed as a disjunction or conjunction of terms. Here we extended the definition by a parameter k , which specifies the number of terms that is required to represent the sentence. This way of generalizing a sentence (itemset) for summarizing itemsets uses the concept of fault-tolerant itemsets and partial-supports of itemsets.

We proposed an algorithm, PUB, to produce the description. This algorithm requires parameters K and w specifying the number of sentences and the maximum number of terms in each sentence respectively. The strength of PUB relies in its broad search of best sentences, and its optimality to find the values specifying the terms. For finding the best sentences, PUB uses a bounded breadth first search strategy, by keeping the top- b sentences in each *level* (number of terms). To extend each sentence with a term, PUB takes linear time w.r.t. the number of objects for finding the optimal values in an attribute.

Through extensive experimental evaluations, we concluded that PUB is both effective and efficient for describing a class in a database. Our case study in *glass* database demonstrated the interpretability of the summary.

178 *CHAPTER 8. SUMMARIZING ITEMSETS FOR CLASS DESCRIPTION*

Chapter 9

Conclusion

9.1 Summary

This thesis aims to improve the state-of-the-art in pattern mining methodology. Here, we focus on two variants of the pattern mining problem, which we term as the *enumeration problem* and the *summarization problem*. The objective of the enumeration problem is to find all interesting itemsets, given an interestingness measure. For the latter, the objective is to represent those interesting itemsets in an interpretable and concise form.

9.1.1 The Enumeration Problem

We observe that there is a huge gap between progresses in enumerating interesting itemsets and frequent itemsets. While the former problem is interesting and useful, it is difficult as general interesting itemsets do not possess some properties of frequent itemsets. In this thesis, we managed to narrow down this performance gap, by proposing efficient algorithms for enumerating interesting itemsets.

In particular, we showed that almost all interestingness measures can be formulated as a function of *detailed supports*, and proposed efficient algorithms to calculate the *detailed supports* of an itemset, with only a little more cost than that to calculate its frequency. With these results, the interestingness of an itemset can be calculated very efficiently. We also proposed a way to efficiently enumerate interesting itemsets when the interestingness measure does not possess the anti-monotone property.

When the measure satisfies the anti-monotone property, our proposed algorithm can boost the efficiency of the state-of-the-art algorithm by two to three orders of

magnitude. In other cases, our framework can produce complete results in a reasonable time, whilst the same problem has been previously considered infeasible.

As our framework is general, it has a major implication. As in, whenever an analyst designs a new interestingness measure, she can simply use our solution framework to obtain the interesting itemsets. By being modular, our framework invites implementation experts to help optimize the implementations of each module, thereby improving the efficiency of the overall mining algorithm.

We also initiated the duality problem, which reduces the problem of enumerating *interesting* itemsets to that of enumerating *frequent* itemsets. The solution of this duality problem allows interesting itemsets to be mined using frequent itemset mining algorithms, and even their implementations. This is useful as the frequent itemset mining problem has been extremely well studied. As a step towards realizing this goal, we successfully proved a duality between the fault-tolerant frequent itemset mining problem and the frequent itemset mining problem.

The results of this thesis improves the efficiency of many interesting itemset enumeration algorithms. The practical implications of our results are that many real problems can now be solved in a better way, and many real data can now be analyzed more effectively. Some classification and clustering techniques are based on interesting itemsets. Interesting itemsets have also been used for analyzing very large databases, such as social networks and web corpora. All these applications can certainly be improved by our innovations. We also believe that our findings would be instrumental in changing expectations and aspirations of research in enumerating itemsets. The common accepted practice is to provide two separate publications: one for proposing a measure with a case study, and the other for proposing an efficient algorithm, typically some years after the initial proposal. With our framework, a pioneering effort can propose a new measure *and* develop an efficient algorithm, thus enabling other researchers to immediately benefit from both the idea and the implementations.

9.1.2 The Summarization Problem

One limitation of any enumeration approach is that its results are abundant. This issue has led to the popular research topic of summarizing interesting itemsets. However, not many prior works are driven by the need to improve the interpretability of interesting itemsets.

In this thesis, we adopt the most widespread use-case – where the analyst interprets the summary by scanning the patterns individually. For such scenarios, we

postulate the requirements for a summary to be interpretable. That is, each summary pattern must be semantically meaningful, and the inference of the knowledge should not require multiple patterns. These requirements are intuitive and practical, and match those for other types of summarization problems, for example document summarization.

As the notions of interesting patterns and the desired knowledge, are domain and application dependent, we do not tackle the general summarization problem. Instead, we tackle three popular variants of the itemsets summarization problem, where we propose an innovative solution to boost the summarization quality in each problem variant. For summarizing frequent itemsets and their frequency, we propose a novel summary form called *c-profile*, which is an extension of pattern profile. For summarizing frequent itemsets and their supporting transactions, we propose to compromise both their precision and recall concurrently. For summarizing sentences describing a concept in a database, we propose to extend the notion of sentences to allow partial coverage of objects. In short, we managed to design novel and interpretable algorithms, which are better than the state-of-the-art algorithms in each of the three variants under consideration.

Nowadays, very large and complicated databases are ubiquitous. As our research improves their analyzability, those complicated databases can now be understood better, and hence better decisions can be made.

9.2 Future Work

We have proposed several algorithms for enumerating interesting itemsets and summarizing interesting itemsets, which advanced the state-of-the-art. However, there is always room for algorithmic improvements. Our algorithms for calculating the detailed supports of itemsets uses a recurrence relation of subsets (c.f., Chapter 2). This property restricts their usability to the specific problem variants, where the desired results are *downward closed*, i.e., an itemset is part of the result-set if all its subsets are. Designing equally efficient algorithms without this restriction is very useful, to boost the efficiency for other problem variants. For instance, for enumerating maximal interesting itemsets, or when the interestingness measure does not possess the anti-monotone property.

Regarding the candidate generation procedure, we still do not have a general formula to bound the search space when the measure does not satisfy the anti-monotone property. Nevertheless, we have proposed several novel techniques. For instance,

for enumerating correlated itemsets (c.f., Chapter 3), we propose to scope our problem from the transactional database into categorical database. We also propose to enumerate from large itemsets, and prune the search space using maximal frequent itemsets. And for enumerating fault-tolerant frequent itemsets (c.f., Chapter 4), we propose to formulate the interestingness of an itemset using linear program, and to aggressively remove items and transactions during enumeration. We would like to exploit these novel strategies, for enumerating interesting itemsets with other interestingness measures.

Though it is general, our framework still requires analysts to provide the properties of the measure, viz., the monotonicity property and the required detailed supports. A module to automatically find the properties would be very useful to improve the usability of our framework.

In Chapter 5, we proved the duality between the frequent itemset mining and the fault-tolerant frequent itemset mining problem. We also presented an algorithm to reduce the problem. We proved that our algorithm is optimal if we transform each transaction independently. Despite being optimal, the number of transactions increase by a factor of $O(m^\varepsilon)$, which is infeasible for reasonably large ε . We would like to study on a more efficient way to exploit the duality between these two problems. In the current state, our reduction is specific to this problem variant. We are interested in finding the properties of the measures, to extend our reduction for other notions of interesting itemsets.

Regarding the summarization problem, our algorithms are currently not very efficient. For summarizing frequent itemsets and their frequency (c.f., Chapter 6), we heavily used the linear programming solver module. Alternatively, we could slightly compensate the conciseness of the summary, by using an efficient but approximate algorithm. For summarizing interesting itemsets and their supporting transactions (c.f., Chapter 7), our algorithm accepts the interesting itemsets as input. This is not efficient since the number of interesting itemsets can be very large. Therefore, it is desirable to design a *direct* summarization algorithm, to directly produce the summary without materializing the interesting patterns. For describing a concept in a database (c.f., Chapter 8), our algorithm repeatedly scans the database to find the *optimal* set/range of values in each attribute. As in the first problem, it might be desirable to use a sub-optimal, but more efficient algorithm. As a theme in our work so far, we have chosen to optimize the summary quality. In the future, we would like to push the efficiency boundary with acceptable loss on accuracy.

Besides efficiency, our algorithms do not provide any performance guarantee regarding the number of patterns in the first two problems, and the accuracy in the third problem. This performance guarantee, or expected performance, will be useful when analysts wish to evaluate our algorithms. It will also be useful to generalize our specialized algorithms into a framework, for summarizing general interesting itemsets, as we achieved in the first part of this thesis, for enumerating interesting itemsets.

Throughout this thesis, we assume that the interestingness measures μ and ν have been defined. However, as mentioned in the introduction of this thesis (c.f., Chapter 1), defining appropriate measures is difficult, and is still an open and very active research topic. Our knowledge on the difficulty for enumerating interesting itemsets might be able to guide the analysts, for defining the *best* interestingness measure. A related challenge is to design a flexible/adaptive mining algorithm, where the notion of interestingness is dynamic. This is very useful since, as mentioned before, it is not trivial to formulate a *desired property* into mathematical equations. And hence, it is very likely that the analyst is not satisfied with the current formulation.

Regarding the patterns themselves, currently we focus only on *itemset*. This restriction limits the usage of our algorithms to transactional databases. Therefore, it is useful to extend the algorithm to admit more complicated patterns such as sequence, tree, and graph.

Currently, we limit our research on main-memory algorithm, where the database and all intermediate results fit inside the main memory. This is a reasonable assumption for most applications, as nowadays main memory is huge and reasonably cheap. However, in some applications such as analyzing social network or web corpus, the data size might reach Terabytes or even Petabytes, which require a completely different data management mechanism. In addition, the data might also be continuously updated, which requires an efficient incremental algorithm.

Throughout this thesis, we tackle the problem in a general manner, without considering the application-domain of the problems. Many domains, such as biology or social networks, have a vast amount of data, and there is knowledge yet to be discovered. We are interested to apply our research results, for discovering knowledge and for better understanding those data, in those applications.

Appendix A

Experimental Setting

All our experiments assume the use of main-memory algorithm. The experiments are performed in Pentium 4, 3 GHz machine with 1 GB main memory. All codes are implemented in C++. For the (integer) linear programming solver, we use lp_solve5.5 library¹. This library can solve a linear programming problem with as large as 30,000 variables and 50,000 constraints.

All datasets we use are taken either from FIMI website² or UCI repository³. Some statistics on these datasets are presented in Tables A.1 and A.2. We also used IBM market-basket database generator⁴ for generating synthetic datasets.

Name	n	m	max length	density (%)
BMS-Webview-1 [111]	59,602	497	267	0.51
Mushroom	8,124	119	23	19.32
Chess	3,196	75	37	49.33

Table A.1: Transactional databases from FIMI.

Name	n	t	m	density (%)
Mushroom	8,124	23	119	19.32
Chess	3,196	37	75	49.33
Soybean	307	36	151	23.84
Pendigits	10,992	17	42	40.48
Zoo	101	16	36	44.44

Table A.2: Categorical datasets from UCI repository. All numerical attributes in the Pendigits dataset are discretized into two bins with equal width.

¹<http://lpsolve.sourceforge.net/5.5/>

²<http://fimi.cs.helsinki.fi/data/>

³<http://archive.ics.uci.edu/ml/datasets.html>

⁴<http://www.almaden.ibm.com/software/quest/Resources/index.shtml>

Appendix B

Proof of Equation (5.4)

We use the identity of binomial coefficient, which is given as:

$$\sum_{i=0}^b (-1)^{b-i} \binom{a}{i} \binom{a-i}{b-i} = 0. \quad (\text{B.1})$$

Proof.

$$\begin{aligned} & \sum_{i=0}^b (-1)^{b-i} \binom{a}{i} \binom{a-i}{b-i} \\ &= \sum_{i=0}^b (-1)^{b-i} \frac{a!}{(a-i)!i!} \cdot \frac{(a-i)!}{(b-i)!(a-b)!} \cdot \left(\frac{b!}{b!}\right) \\ &= \sum_{i=0}^b (-1)^{b-i} \binom{a}{b} \cdot \binom{b}{i} \\ &= \binom{a}{b} \sum_{i=0}^b (-1)^{b-i} \binom{b}{i} = \binom{a}{b} \cdot 0 = 0. \end{aligned}$$

□

□

Similarly, we have:

$$\sum_{i=0}^b (-1)^{b-i} \binom{a}{i} \binom{a-i-1}{b-i} = 1. \quad (\text{B.2})$$

Proof. We prove Equation (B.2) by induction. Base case: when $b = 0$ and $a = b + 1$ can be proven by simple algebra. For the induction step, assuming that (B.2) holds

for all $a' < a$ and $b' < b$,

$$\sum_{i=0}^b (-1)^{b-i} \binom{a}{i} \binom{a-i-1}{b-i} = \dots$$

Here we split $\binom{a}{i}$ into $\binom{a-1}{i} + \binom{a-1}{i-1}$ when $i > 0$, and equate $\binom{a}{i} = \binom{a-1}{i}$ when $i = 0$, hence we have:

$$\begin{aligned} &= \sum_{i=0}^b (-1)^{b-i} \binom{a-1}{i} \binom{a-1-i}{b-i} \\ &\quad + \sum_{i=1}^b (-1)^{b-i} \binom{a-1}{i-1} \binom{a-i-1}{b-i}. \end{aligned}$$

Note that LHS of “+” sign is in the form of Equation (B.1) hence equals to 0. For RHS, let $j = i - 1$, $a' = a - 1$, and $b' = b - 1$, we have:

$$= \sum_{j=0}^{b'} (-1)^{b'-j} \binom{a'}{j} \binom{a'-j-1}{b'-j} = 1. \text{ (from the assumption)}$$

□

□

Our target is to prove that:

$$w(a, b) = 1 - \sum_{i=1}^b \binom{a}{i} \cdot w(a-i, b-i) = (-1)^b \binom{a-1}{b}. \quad (\text{B.3})$$

We prove (B.3) by induction. As in the proof of (B.2), we omit the proof for the base case when $b = 0$ and $a = b + 1$ as they can be proven by simple algebra. For the induction step, assuming that (B.3) holds for all $a' < a$ and $b' < b$,

$$\begin{aligned} w(a, b) &= 1 - \sum_{i=1}^b \binom{a-1}{i} (-1)^{b-i} \binom{a-i-1}{b-i} \\ &= 1 - \sum_{i=0}^b \binom{a-1}{i} (-1)^{b-i} \binom{a-i-1}{b-i} + (-1)^b \binom{a-1}{b} \end{aligned}$$

The middle part is the same as in Equation (B.2) and hence equal to 1. Therefore,

we have:

$$\begin{aligned}w(a, b) &= 1 - 1 + (-1)^b \binom{a-1}{b} \\ &= (-1)^b \binom{a-1}{b}\end{aligned}$$

□

Bibliography

- [1] F. N. Afrati, A. Gionis, and H. Mannila. Approximating a collection of frequent sets. In *KDD*, pages 12–19, 2004.
- [2] C. C. Aggarwal and P. S. Yu. A new framework for itemset generation. In *PODS*, pages 18–24. ACM Press, 1998.
- [3] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data. *Data Mining and Knowledge Discovery*, 11(1):5–33, 2005.
- [4] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD Conference*, pages 207–216, 1993.
- [5] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.
- [6] S. Agrawal and J. R. Haritsa. A framework for high-accuracy privacy-preserving mining. In *ICDE*, pages 193–204, 2005.
- [7] S. Agrawal, V. Krishnan, and J. R. Haritsa. On addressing efficiency concerns in privacy-preserving mining. In *DASFAA*, pages 113–124, 2004.
- [8] I. Assent, R. Krieger, E. Müller, and T. Seidl. INSCY: Indexing subspace clusters with in-process-removal of redundancy. In *ICDM*, pages 719–724, 2008.
- [9] D. Avis and K. Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*, 65(1-3):21–46, 1996.
- [10] R. J. Bayardo, Jr. Efficiently mining long patterns from databases. In *SIGMOD Conference*, pages 85–93, 1998.

- [11] R. J. Bayardo, Jr., B. Goethals, and M. J. Zaki, editors. *FIMI '04, Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Brighton, UK, November 1, 2004*, volume 126 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.
- [12] J. Besson, R. G. Pensa, C. Robardet, and J.-F. Boulicaut. Constraint-based mining of fault-tolerant patterns from boolean data. In *KDID*, pages 55–71, 2005.
- [13] F. Birzele and S. Kramer. A new representation for protein secondary structure prediction based on frequent patterns. *Bioinformatics*, 22(21):2628–2634, 2006.
- [14] F. Bonchi, F. Giannotti, C. Lucchese, S. Orlando, R. Perego, and R. Trasarti. A constraint-based querying system for exploratory pattern discovery. *Information Systems*, 34(1):3–27, 2009.
- [15] F. Bonchi and C. Lucchese. Pushing tougher constraints in frequent pattern mining. In *PAKDD*, pages 114–124, 2005.
- [16] C. Borgelt. Efficient implementations of apriori and eclat. In *FIMI*, 2003.
- [17] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: Generalizing association rules to correlations. In *SIGMOD Conference*, pages 265–276, 1997.
- [18] Y. Cai, N. Cercone, and J. Han. Attribute-oriented induction in relational databases. In *Knowledge Discovery in Databases*, pages 213–228. AAAI/MIT Press, 1991.
- [19] T. Calders. Computational complexity of itemset frequency satisfiability. In *PODS*, pages 143–154, 2004.
- [20] T. Calders and B. Goethals. Depth-first non-derivable itemset mining. In *SDM*, 2005.
- [21] T. Calders and B. Goethals. Quick inclusion-exclusion. In *KDID*, pages 86–103, 2005.
- [22] T. Calders and B. Goethals. Non-derivable itemset mining. *Data Mining and Knowledge Discovery*, 14(1):171–206, 2007.
- [23] H. Cheng, P. S. Yu, and J. Han. Approximate frequent itemset mining in the presence of random noise. pages 363–389, 2007.

- [24] J. Cheng, Y. Ke, and W. Ng. δ -tolerance closed frequent itemsets. In *ICDM*, pages 139–148, 2006.
- [25] E. F. Codd, S. B. Codd, and C. T. Salley. Providing OLAP to user-analysts: An IT mandate. Technical report, 1993.
- [26] D. H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Mach. Learn.*, 2(2):139–172, Sept. 1987.
- [27] A. W.-C. Fu, R. W. w. Kwong, and J. Tang. Mining N-most interesting itemsets. In *ISMIS*, pages 59–67, 2000.
- [28] B. C. M. Fung, K. Wang, and M. Ester. Hierarchical document clustering using frequent itemsets. In *SDM*, 2003.
- [29] L. Geng and H. J. Hamilton. Interestingness measures for data mining: A survey. *ACM Computing Surveys*, 38(3), 2006.
- [30] C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu. Mining frequent patterns in data streams at multiple time granularities. In *Next Generation Data Mining*, 2002.
- [31] K. Gouda and M. J. Zaki. Genmax: An efficient algorithm for mining maximal frequent itemsets. *Data Mining and Knowledge Discovery*, 11(3):223–242, 2005.
- [32] G. Grahne and J. Zhu. High performance mining of maximal frequent itemsets. In *HPDM*, 2003.
- [33] R. Gupta, G. Fang, B. Field, M. Steinbach, and V. Kumar. Quantitative evaluation of approximate frequent pattern mining algorithms. In *KDD*, pages 301–309, 2008.
- [34] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: current status and future directions. *Data Mining and Knowledge Discovery*, 15(1):55–86, 2007.
- [35] J. Han and M. Kamber. *Data Mining. Concepts and Techniques*. Morgan Kaufmann, 2nd edition, 2006.
- [36] J. Han, S. Nishio, H. Kawano, and W. Wang. Generalization-based data mining in object-oriented databases using an object cube model. *Data and Knowledge Engineering*, 25(1-2):55–97, 1998.

- [37] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD Conference*, pages 1–12, 2000.
- [38] J. Han, J. Wang, Y. Lu, and P. Tzvetkov. Mining top-k frequent closed patterns without minimum support. In *ICDM*, pages 211–218, 2002.
- [39] H. Heikinheimo, J. K. Seppänen, E. Hinkkanen, H. Mannila, and T. Mielikäinen. Finding low-entropy sets and trees from binary data. In *KDD*, pages 350–359, 2007.
- [40] H. Heikinheimo, J. Vreeken, A. Siebes, and H. Mannila. Low-entropy set selection. In *SDM*, pages 569–580, 2009.
- [41] R. Jin, M. Abu-Ata, Y. Xiang, and N. Ruan. Effective and efficient itemset pattern summarization: regression-based approaches. In *KDD*, pages 399–407, 2008.
- [42] Y. Ke, J. Cheng, and W. Ng. Mining quantitative correlated patterns using an information-theoretic approach. In *KDD*, pages 227–236, 2006.
- [43] J. A. Kelner and D. A. Spielman. A randomized polynomial-time simplex algorithm for linear programming. In *STOC*, pages 51–60, 2006.
- [44] J.-L. Koh and P.-W. Yo. An efficient approach for mining fault-tolerant frequent patterns based on bit vector representations. In *DASFAA*, pages 568–575, 2005.
- [45] H.-P. Kriegel, P. Kröger, and A. Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *ACM Transactions on Knowledge Discovery from Data*, 3(1), 2009.
- [46] M. Kryszkiewicz and M. Gajek. Concise representation of frequent patterns based on generalized disjunction-free generators. In *PAKDD*, pages 159–171, 2002.
- [47] W. Lee, S. J. Stolfo, and K. W. Mok. Mining in a data-flow environment: Experience in network intrusion detection. In *KDD*, pages 114–124, 1999.
- [48] Y.-K. Lee, W.-Y. Kim, Y. D. Cai, and J. Han. Comine: Efficient mining of correlated patterns. In *ICDM*, pages 581–584, 2003.
- [49] J. Li, H. Li, L. Wong, J. Pei, and G. Dong. Minimum description length principle: Generators are preferable to closed patterns. In *AAAI*, 2006.

- [50] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *KDD*, pages 80–86, 1998.
- [51] B. Liu, W. Hsu, L.-F. Mun, and H.-Y. Lee. Finding interesting patterns using user expectations. *IEEE Transactions on Knowledge and Data Engineering*, 11(6):817–832, 1999.
- [52] J. Liu, S. Paulsen, X. Sun, W. Wang, A. B. Nobel, and J. Prins. Mining approximate frequent itemsets in the presence of noise: Algorithm and analysis. In *SDM*, 2006.
- [53] S. C. Madeira and A. L. Oliveira. Biclustering algorithms for biological data analysis: A survey. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1:24–45, 2004.
- [54] H. Mannila and H. Toivonen. Multiple uses of frequent sets and condensed representations (extended abstract). In *KDD*, pages 189–194, 1996.
- [55] R. S. Michalski and R. E. Stepp. Learning from observation: Conceptual clustering. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, chapter 11, pages 331–364. Tioga, 1983.
- [56] S. Morishita and J. Sese. Traversing itemset lattice with statistical metric pruning. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 15-17, 2000, Dallas, Texas, USA*, pages 226–236. ACM, 2000.
- [57] E. Müller, I. Assent, S. Günnemann, R. Krieger, and T. Seidl. Relevant subspace clustering: Mining the most interesting non-redundant concepts in high dimensional data. In *ICDM*, pages 377–386, 2009.
- [58] E. Müller, S. Günnemann, I. Assent, and T. Seidl. Evaluating clustering in subspace projections of high dimensional data. *PVLDB*, 2(1):1270–1281, 2009.
- [59] E. Omiecinski. Alternative interest measures for mining associations in databases. *IEEE Transactions on Knowledge and Data Engineering*, 15(1):57–69, 2003.
- [60] L. Parida and N. Ramakrishnan. Redescription mining: Structure theory and algorithms. In *AAAI*, pages 837–844, 2005.

- [61] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT*, pages 398–416, 1999.
- [62] A. Prikainen and M. Meila. Comparing subspace clusterings. *IEEE Transactions on Knowledge and Data Engineering*, 18(7):902–916, 2006.
- [63] D. Pavlov, H. Mannila, and P. Smyth. Beyond independence: Probabilistic models for query approximation on binary transaction data. *IEEE Transactions on Knowledge and Data Engineering*, 15(6):1409–1421, 2003.
- [64] J. Pei, G. Dong, W. Zou, and J. Han. On computing condensed frequent pattern bases. In *ICDM*, pages 378–385, 2002.
- [65] J. Pei, G. Dong, W. Zou, and J. Han. Mining condensed frequent-pattern bases. *Knowledge and Information Systems*, 6(5):570–594, 2004.
- [66] J. Pei and J. Han. Constrained frequent pattern mining: a pattern-growth view. *SIGKDD Explorations*, 4(1):31–39, 2002.
- [67] J. Pei, J. Han, and L. V. S. Lakshmanan. Mining frequent item sets with convertible constraints. In *ICDE*, pages 433–442, 2001.
- [68] J. Pei, J. Han, and L. V. S. Lakshmanan. Pushing convertible constraints in frequent itemset mining. *Data Mining and Knowledge Discovery*, 8(3):227–252, 2004.
- [69] J. Pei, A. K. H. Tung, and J. Han. Fault-tolerant frequent pattern mining: Problems and challenges. In *DMKD Workshop*, 2001.
- [70] A. K. Poernomo, C. Garboni, T. Calders, V. Gopalkrishnan, B. Goethals, and C. Robardet. Representation of the support for itemsets. *Under preparation for publication*.
- [71] A. K. Poernomo and V. Gopalkrishnan. Mining statistical information of frequent fault-tolerant patterns in transactional databases. In *ICDM*, pages 272–281, 2007.
- [72] A. K. Poernomo and V. Gopalkrishnan. Duality between the fault-tolerant frequent itemsets and frequent itemset mining problems. Technical Report TR-CAIS-2008/09, Nanyang Technological University, Singapore, September 2008.

- [73] A. K. Poernomo and V. Gopalkrishnan. CP-summary: A concise representation for browsing frequent itemsets. In *KDD*, pages 687–696, 2009.
- [74] A. K. Poernomo and V. Gopalkrishnan. Efficient computation of partial-support for mining interesting itemsets. In *SDM*, pages 1014–1025, 2009.
- [75] A. K. Poernomo and V. Gopalkrishnan. PUB: A class description technique based on partial coverage of subspace. In *ICDM*, pages 920–925, 2009.
- [76] A. K. Poernomo and V. Gopalkrishnan. Towards efficient mining of proportional fault-tolerant frequent itemsets. In *KDD*, pages 697–706, 2009.
- [77] A. K. Poernomo and V. Gopalkrishnan. Efficiently mining top-K correlated itemsets in categorical databases. Technical Report TR-CAIS-2010/02, Nanyang Technological University, Singapore, February 2010.
- [78] A. K. Poernomo and V. Gopalkrishnan. Summarizing subspace clusters by compromising coverage and non-spuriousness. Technical Report TR-CAIS-2010/03, Nanyang Technological University, Singapore, March 2010.
- [79] P. Pritchard. A fast bit-parallel algorithm for computing the subset partial order. *Algorithmica*, 24(1):76–86, 1999.
- [80] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
- [81] S. Rizvi and J. R. Haritsa. Maintaining data privacy in association rule mining. In *VLDB*, pages 682–693, 2002.
- [82] C. Robardet, R. G. Pensa, J. Besson, and J.-F. Boulicaut. Using classification and visualization on pattern databases for gene expression data analysis. In *PaRMa*, 2004.
- [83] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-driven exploration of OLAP data cubes. In *EDBT*, pages 168–182, 1998.
- [84] J. K. Seppänen. Using and extending itemsets in data mining: Query approximation, dense itemsets, and tiles, May 2006.
- [85] J. K. Seppänen and H. Mannila. Dense itemsets. In *KDD*, pages 683–688, 2004.

- [86] R. Srikant and R. Agrawal. Mining generalized association rules. In *VLDB*, pages 407–419, 1995.
- [87] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *SIGMOD Conference*, pages 1–12, 1996.
- [88] N. Tatti. Maximum entropy based significance of itemsets. In *ICDM*, pages 312–321, 2007.
- [89] N. Tatti. Maximum entropy based significance of itemsets. *Knowledge and Information Systems*, 17(1):57–77, 2008.
- [90] T. Uno and H. Arimura. An efficient polynomial delay algorithm for pseudo frequent itemset mining. In *Discovery Science*, pages 219–230, 2007.
- [91] T. Uno, T. Asai, Y. Uchida, and H. Arimura. LCM: An efficient algorithm for enumerating frequent closed item sets. In *FIMI*, 2003.
- [92] T. Uno, M. Kiyomi, and H. Arimura. Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *FIMI*, 2004.
- [93] M. van Leeuwen. Patterns that matter, February 2010.
- [94] C. Wang and S. Parthasarathy. Summarizing itemset patterns using probabilistic models. In *KDD*, pages 730–735, 2006.
- [95] J. Wang, J. Han, and J. Pei. Closet+: searching for the best strategies for mining frequent closed itemsets. In *KDD*, pages 236–245, 2003.
- [96] K. Wang, S. Zhou, and J. Han. Profit mining: From patterns to actions. In *EDBT*, pages 70–87, 2002.
- [97] G. I. Webb. Discovering associations with numeric variables. In *KDD*, pages 383–388, 2001.
- [98] G. I. Webb. Self-sufficient itemsets: An approach to screening potentially interesting associations between items. *ACM Transactions on Knowledge Discovery from Data*, 4(1):1–20, 2010.
- [99] G. I. Webb and S. Zhang. K-optimal rule discovery. *Data Mining and Knowledge Discovery*, 10(1):39–79, 2005.

- [100] A. K. C. Wong and G. C. L. Li. Simultaneous pattern and data clustering for pattern cluster analysis. *IEEE Transactions on Knowledge and Data Engineering*, 20(7):911–923, 2008.
- [101] T. Wu, Y. Chen, and J. Han. Re-examination of interestingness measures in pattern mining: a unified framework. *Data Mining and Knowledge Discovery*, 21(3):371–397, 2010.
- [102] Y. Xiang, R. Jin, D. Fuhry, and F. F. Dragan. Succinct summarization of transactional databases: an overlapped hyperrectangle scheme. In *KDD*, pages 758–766, 2008.
- [103] D. Xin, J. Han, X. Yan, and H. Cheng. Mining compressed frequent-pattern sets. In *VLDB*, pages 709–720, 2005.
- [104] X. Yan, H. Cheng, J. Han, and D. Xin. Summarizing itemset patterns: a profile-based approach. In *KDD*, pages 314–323, 2005.
- [105] C. Yang, U. M. Fayyad, and P. S. Bradley. Efficient discovery of error-tolerant frequent itemsets in high dimensions. In *KDD*, pages 194–203, 2001.
- [106] M. J. Zaki and M. Ogihara. Theoretical foundations of association rules. In *DMKDW*, 1998.
- [107] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *KDD*, pages 283–286, 1997.
- [108] M. J. Zaki, M. Peters, I. Assent, and T. Seidl. CLICKS: an effective algorithm for mining subspace clusters in categorical datasets. In *KDD*, pages 736–742, 2005.
- [109] M. J. Zaki and N. Ramakrishnan. Reasoning about sets using redescription mining. In *KDD*, pages 364–373, 2005.
- [110] L. Zhao, M. J. Zaki, and N. Ramakrishnan. Blossom: a framework for mining arbitrary boolean expressions. In *KDD*, pages 827–832, 2006.
- [111] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *KDD*, pages 401–406, 2001.
- [112] J. Zhu and G. Grahne. Reducing the main memory consumptions of FPmax* and FPclose. In *FIMI*, 2004.