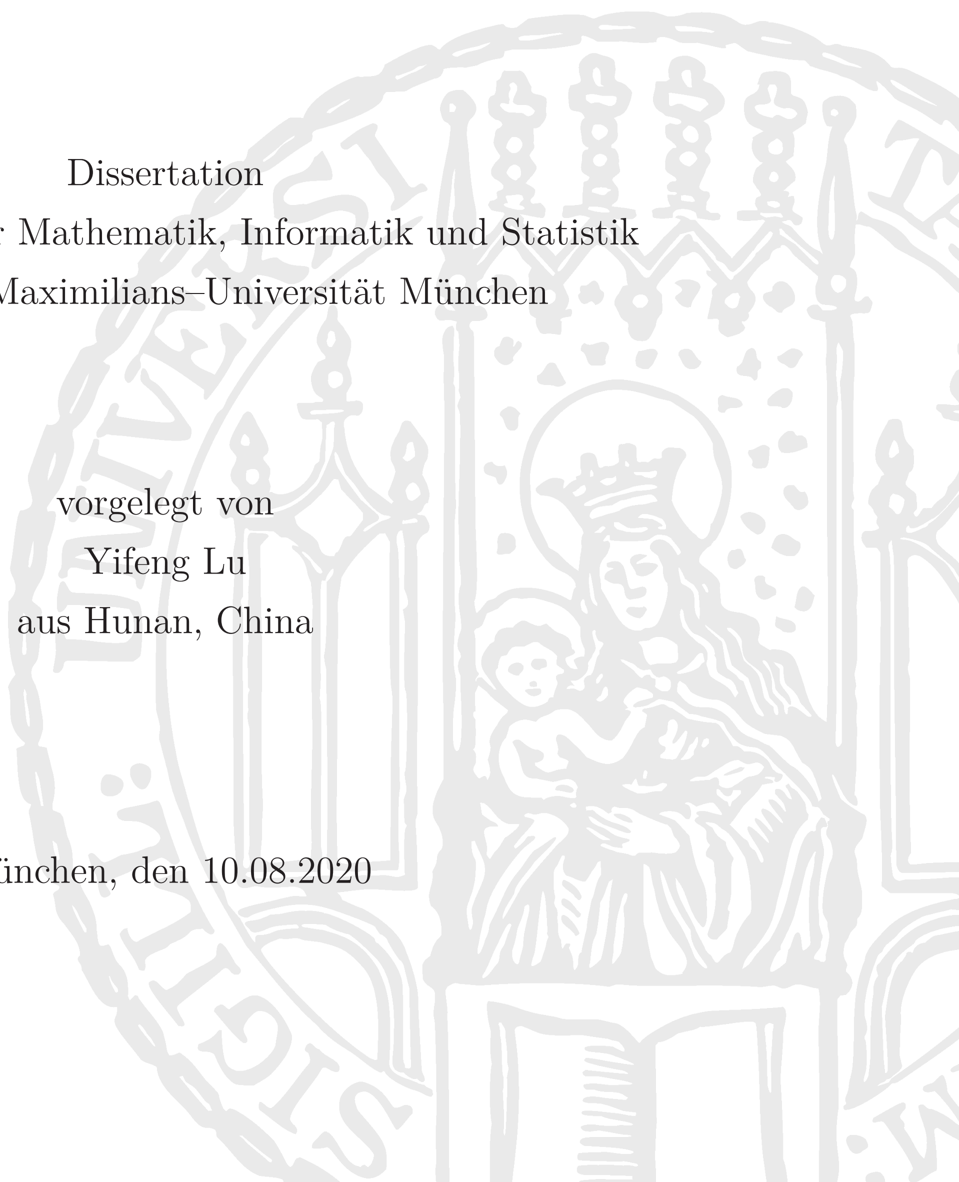
Pattern Mining under Different Conditions

Yifeng Lu

München 2020

Pattern Mining under Different Conditions

Yifeng Lu



Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig–Maximilians–Universität München

vorgelegt von
Yifeng Lu
aus Hunan, China

München, den 10.08.2020

Erstgutachter: Prof. Dr. Thomas Seidl
Zweitgutachter: Prof. Dr. Ira Assent
Datum der mündlichen Prüfung: 12.05.2021

Eidesstattliche Versicherung

Hiermit erkläre ich, Yifeng Lu, an Eides statt, dass die vorliegende Dissertation ohne unerlaubte Hilfe gemäß Promotionsordnung vom 12.07.2011, § 8, Abs. 2 Pkt. 5, angefertigt worden ist.

München, 10.08.2020

. . . .

Yifeng Lu

Contents

List of Figures vii

List of Tables xiii

Acknowledgement xiv

Abstract xvii

Zusammenfassung xix

1 Introduction 1

1.1 Knowledge Discovery in Databases 2
1.2 Pattern Mining in This Thesis 4
1.3 New Challenges in Pattern Mining 6
1.4 Outline of this thesis 8

I Rare Itemset Mining 11

2 Introduction 13

2.1 Rare Itemset Mining 13
2.2 Related Works 15
2.3 Powerset Lattice Traversing 17
2.4 Frequent Itemset Mining Algorithms 18

3 Negative Itemset Tree 25

3.1 Preliminaries 26

3.2 Negative Infrequent Itemset Tree 28
3.3 Experimental Evaluation 32
3.4 Conclusion and Future Works 34

4 Negative Itemset Tree with Residual Counts 35
4.1 Negative Itemset Tree and Support Counting 36
4.2 Infrequent Pattern Mining with Termination Nodes Pruning . 39
4.3 Infrequent Pattern Mining with 1st-layer Nodes Pruning . . . 42
4.4 Experimental Evaluation 43
4.5 Conclusion and Future Works 46

5 Bi-directional Rare Closed Itemset Mining 49
5.1 Closed Infrequent Itemset 50
5.2 Bi-directional Infrequent Itemset Mining 52
5.3 Experiments 62
5.4 Discussion and Conclusion 66

6 Mining Closed Rare Itemsets on NI-Tree 69
6.1 Basic Definition 70
6.2 Closed Itemset on Negative Itemset Tree 72
6.3 Algorithm: LSCMiner 75
6.4 Experiments 83
6.5 Conclusion 87

II Interval-based Temporal Pattern Mining 89

7 Introduction 91
7.1 Towards Interval-based Temporal Pattern Mining on Streams . 91
7.2 Related Works 93
7.3 Fundamental Sequential Pattern Mining Algorithm 96

8 Temporal Patterns in Streams 101
8.1 Temporal Event and Temporal Sequence 101
8.2 Similarity Measures and Temporal Pattern 102

9 Incremental Temporal Pattern Mining 107
9.1 Stream Clustering 108
9.2 Support Estimation 110
9.3 Fast Micro-cluster Updating 114

10 Mining Interval-based Event Streams 119
10.1 Interval-based Temporal Pattern Mining in Static Database . . 120
10.2 Interval-based Temporal Event and Sequence 121
10.3 Similarity Measures 122
10.4 Interval-based Event Stream 124
10.5 Experiment Results 125
10.6 Conclusion and Future Perspective 130

III *k*NN based Clustering 131

11 Introduction 133
11.1 Density-based Clustering 133
11.2 *k*NN-based Clustering - New Challenges 137

12 Shape Alternation Adaptable Clustering 141
12.1 Related Works 142
12.2 Preliminaries 143
12.3 Ada*k*NN Clustering Algorithm 145
12.4 Experimental Results 150
12.5 Further Properties and discussions 154
12.6 Conclusion 158

13 Extremely Noisy Datasets Clustering 161
13.1 Related Works 162
13.2 KENClus Clustering Algorithm 163
13.3 Experiments 173
13.4 Conclusion 180

14 Summary and Outlook 181

Summary and Outlook **181**

14.1 Summary 181

14.2 Outlook 182

Bibliography **185**

List of Figures

1.1 The KDD process. 2

2.1 Powerset lattice. The minimum support splits the lattice into three parts. 17

2.2 A simple powerset lattice with four distinct items. 19

3.1 Examples of (a) Negative Itemset tree and its corresponding de-tree by excluding (b) $\neg C$ and (c) $\neg C, \neg D$. (c) is also a de-tree of (b). 29

3.2 (a) The pseudo-subtraction of $\neg C$ in practice, only root is created, no merging happened. (b,c) Examples of de-tree by excluding $\neg D$ and $\neg C \neg E$. Only 1st-layer nodes are checked and removed. 31

3.3 Runtime on Connect-4 dataset 33

4.1 Negative Itemset tree built based on dataset in Figure 6.1. Termination nodes are marked in red. 36

4.2 Examples de-trees by excluding $\neg C$, $\neg C \neg D$ and $\neg C \neg D \neg B$ from the NI-tree in Figure 6.2. The node of $\neg B$ is not removed in tree (c) since the subtraction process is terminated when all *1st-layer* nodes are covered by the itemset in root. 38

4.3 Statistics on real datasets. $|\mathcal{T}|$: transaction number, $|X|$: number of items, $|I|$: number of distinct items, $|L|$: average itemset size. 44

4.4 Runtime experiments on different absolute minimum support values. 46
4.5 Runtime experiments on different dataset size. 47

5.1 Ascending ordered frequency plot of items in BMS1 and mushrooms dataset. 53
5.2 Cumulative traversing time of LCM [97] on each projected databases of the BMS1 dataset. 54
5.3 Runtime performance on retail dataset, $\delta = 1\%, L_R = 10$ . . . 63
5.4 Runtime performance on BMS1 dataset, $\delta = 0.3\%, L_R = 15$ . . 64
5.5 Runtime performance on BMS2 dataset, $\delta = 0.3\%, L_R = 15$ . . 64
5.6 Runtime performance on mushrooms dataset, $delta = 30\%, L_R = 10$ 64
5.7 Runtime performance on chess dataset, $\delta = 1\%, L_R = 9, L_L = 25$ 65
5.8 Runtime performance on connect dataset,$\delta = 30\%, L_R = 9, L_L = 15$ 65

6.1 Transaction dataset and its negative dataset. 71
6.2 The initial ni-tree of dataset in Figure6.1. Red nodes are *t-nodes*. 71
6.3 Counting support on ni-tree. 72
6.4 The adapted initial ni-tree with t-node links (blue) and corresponding ni-tree by removing t-node 1, 2 or $1, 2$ together. Each link is marked with the t-node id. In each step, only child t-nodes of root are considered (e.g. node 6 can only be removed after node 4). 74
6.5 We solve the mining task by removing items in a recursive way. Such process can be represented as a tree. 76
6.6 The adapted ni-tree used in LSCMiner. R is the set of items been removed so far. 76
6.7 Recursive steps of the removing process $a \to ab \to abd \to abde$. 76
6.8 Given the current unclosed ni-tree, Trial-and-Error pruning will try all deletion sets under $R \cup c$. If no closed patterns exists, then later deletion sets can be skipped. 78

6.9 The maximum item under nodes of a is $iM'_1 = b$. Later removing process on the right ni-tree must removing b first since otherwise, t-nodes that cover a will not be removed. 79
6.10 There is a t-node that covers b. Then the maximum upper bound up to now (iM'_2), which equals to the upper bound when removing a, is used as the upper bound for further removing process on the right ni-tree. 79
6.11 Real-life datasets in our experiments. 83
6.12 Runtime on small dense dataset. 84
6.13 Runtime on large dense datasets. 85
6.14 Runtime on `BMS1` dataset (default: $N = 60k, L = 30, \beta = 20$) . 86
6.15 Memory consumption and runtime under different δ values. δ is set up to 0.4 on `connect` dataset since almost all items occurred less than 40%. 87

7.1 Streams will be merged together in order to find *intra*- and *inter*-pattern. 93
7.2 Mining models employed in point-based stream mining. Both (a) Batch and (b) Sliding Window provide a static database. . 94
7.3 Two sequences with the same event types and similar temporal information, but different events order due to some natural noise. 96

8.1 Small example of point-based events stream. 101
8.2 Similarity between temporal sequences is computed by moving sequence first. Points represent events, colors represent event labels. 104

9.1 Current sequence is covered by the sliding window. All candidate subsequences for pattern updating contain the newly evolved event ($\underline{\mathbf{A}}$). 107
9.2 Prefix tree for micro-cluster update. 114

10.1 Stream of interval-based events, each event is represented by a rectangle. 119

10.2 Two sequences with the same event types and similar temporal information, but support different patterns due to some natural noise. 121
10.3 13 Allen's relations and the 2D representation. 122
10.4 Similarity between interval events (a) and interval sequences (b). 123
10.5 Visualization of interval-based temporal pattern as an average sequence. 124
10.6 Sliding window on the interval-based event stream shown in Figure 10.1. 125
10.7 Accuracy on synthetic stream with respect to the sliding window size and the minimum support. 126
10.8 Processing time on synthetic stream with respect to the stream length, the minimum support and the sliding window size. . . 127
10.9 Accuracy on real stream with respect to the sliding window size and the minimum support. 128
10.10 Processing time on real stream with respect to the minimum support and the sliding window size. 128
10.11 Processing time on interval-based synthetic stream with respect to the stream length, the minimum support and the sliding window size.. 129
10.12 Processing time on interval-based real stream with respect to the minimum support and the sliding window size. 129

11.1 RNNDBSCAN correctly detects one cluster in (a) but failed to find two clusters properly in (b) and (c), when parameters are fixed. Even if we set parameters to find two clusters, the small cluster in (d) is incorrectly labeled as noise. 138
11.2 A real extremely noisy dataset example. 140

12.1 Clustering results of our approach. The parameter k is set to work properly on (d). No further parameter tuning is applied to the rest of datasets. 142
12.2 Clustering results of AP under default settings. 145

12.3 Similarity and Preference on toy MST example. 147
12.4 Visualization of kNN based clustering approaches on synthetic datasets and the ARI score under different k values. 155
12.5 Visualization of kNN based clustering approaches on synthetic datasets and the ARI score under different k values. 156
12.6 Visualization of kNN based clustering approaches on synthetic datasets and the ARI score under different k values. 156
12.7 Visualization of kNN based clustering approaches on synthetic datasets and the ARI score under different k values. 157
12.8 Cluster centers (red stars) and their preference values in the dataset. 157
12.9 ARI Performance vs k on *blobs* and *strip* dataset with different sizes. 158
12.10 Histogram of cluster number identified by our approach, $k \in [1, 100]$. The best (red) and the worst (blue) ARI score are illustrated for each case. 159

13.1 The clustering result of our KENClus algorithm a real-world example. 161
13.2 Clustering results on extremely noisy datasets. Here, the results of DBSCAN and HDBSCAN are promising because we put huge efforts to obtain the correct parameters. 162
13.3 Datasets under different noise settings and their k-dist histogram. 164
13.4 Comparison of histograms k-dist and our local-density with respect to the dataset in Figure 13.2a. 166
13.5 Each splitting step divides the local-density histogram into two parts. The right part (blue) is used for the next splitting. 167
13.6 Splitting local-density histogram using $k$means and EM algorithms. 167

13.7 Noise identification results on high and extremely high noise datasets. Some noises that are close to clusters or locate in small but dense areas due to randomness are identified as in-cluster points. Further refinement is necessary. 169
13.8 Synthetic dataset examples. The noise density and the noise quantity are controlled independently. 176
13.9 Experimental results on synthatic datasets. 177
13.10 Experimental results on real datasets with varies background noise levels 179

List of Tables

2.1 Powerset lattice traversing strategies and example algorithms. 17
2.2 An example of transaction database 19

3.1 Example transaction database (a) and its corresponding neg-rep dataset (b). 26

5.1 Example dataset 50
5.2 Projected databases extracted (a) from the database in Table 5.1 and (b) from the projected database of $\{a, b, c\}$ (the first row in (a)). Each row is a projected database. 57
5.3 Statistics of real-life datasets used in our experiments. 62

7.1 An example of sequence database 97
7.2 An example of temporal sequence database, each event is aligned with a timestamp. 97

12.1 Dataset Statistics 151
12.2 ARI & NMI Score on Synthetic Datasets 153
12.3 ARI & NMI Score on Real-World Datasets 154

Acknowledgement

A lot of people supported and encouraged me during my Ph.D. study. I want to take this opportunity to express my sincere appreciation for all the help I got. Although I can only mention some of them here, my thanks go to all of them.

First and foremost, I would like to express my sincere gratitude to my supervisor Prof. Dr. Thomas Seidl, for his expert supervision, invaluable advice, and endless support. His great experiences and the organizational background allowed me to work on this challenging domain. Furthermore, I warmly thank Prof. Dr. Ira Assent for her interest in my work and her willingness to act as the second referee on this thesis.

Many discussions and cooperations inspired this work with my colleagues. I am very grateful for all the support I got during the past years. They have made the past few years a great experience and an enjoyable part of my life. I want to thank Prof. Marwan Hassani, Florian Richter, Evgeniy Faerman, Daniyal Kazempour, Julian Busch, Sebastian Schmoll, and Max Berrendorf for constructive and productive discussions.

Furthermore, I especially thank Susanne Grienberger for her background help. She managed much of the administrative burden to make working in this group comfortable. I also want to express thanks to Franz Krojer, who helped to master all technical issues.

Last but not least, I would like to express my sincerest thanks to my wife Zirong. She is always my sturdy backing with her endless love, understanding, and support.

Abstract

New requirements and demands on pattern mining arise in modern applications, which cannot be fulfilled using conventional methods. For example, in scientific research, scientists are more interested in unknown knowledge, which usually hides in significant but not frequent patterns. However, existing itemset mining algorithms are designed for very frequent patterns. Furthermore, scientists need to repeat an experiment many times to ensure reproducibility. A series of datasets are generated at once, waiting for clustering, which can contain an unknown number of clusters with various densities and shapes. Using existing clustering algorithms is time-consuming because parameter tuning is necessary for each dataset. Many scientific datasets are extremely noisy. They contain considerably more noises than in-cluster data points. Most existing clustering algorithms can only handle noises up to a moderate level. Temporal pattern mining is also important in scientific research. Existing temporal pattern mining algorithms only consider point-based events. However, most activities in the real-world are interval-based with a starting and an ending timestamp. This thesis developed novel pattern mining algorithms for various data mining tasks under different conditions.

The first part of this thesis investigates the problem of mining less frequent itemsets in transactional datasets. In contrast to existing frequent itemset mining algorithms, this part focus on itemsets that occurred not that frequent. Algorithms NIIMiner, RaCloMiner, and LSCMiner are proposed to identify such kind of itemsets efficiently. NIIMiner utilizes the negative itemset tree to extract all patterns that occurred less than a given support threshold in a top-down depth-first manner. RaCloMiner combines existing bottom-up frequent itemset mining algorithms with a top-down itemset

mining algorithm to achieve a better performance in mining less frequent patterns. LSCMiner investigates the problem of mining less frequent closed patterns.

The second part of this thesis studied the problem of interval-based temporal pattern mining in the stream environment. Interval-based temporal patterns are sequential patterns in which each event is aligned with a starting and ending temporal information. The ability to handle interval-based events and stream data is lacking in existing approaches. A novel interval-based temporal pattern mining algorithm for stream data is described in this part.

The last part of this thesis studies new problems in clustering on numeric datasets. The first problem tackled in this part is shape alternation adaptivity in clustering. In applications such as scientific data analysis, scientists need to deal with a series of datasets generated from one experiment. Cluster sizes and shapes are different in those datasets. A kNN density-based clustering algorithm, kadaClus, is proposed to provide the shape alternation adaptability so that users do not need to tune parameters for each dataset. The second problem studied in this part is clustering in an extremely noisy dataset. Many real-world datasets contain considerably more noises than in-cluster data points. A novel clustering algorithm, kenClus, is proposed to identify clusters in arbitrary shapes from extremely noisy datasets. Both clustering algorithms are kNN-based, which only require one parameter k.

In each part, the efficiency and effectiveness of the presented techniques are thoroughly analyzed. Intensive experiments on synthetic and real-world datasets are conducted to show the benefits of the proposed algorithms over conventional approaches.

Zusammenfassung

In modernen Anwendungen ergeben sich neue Anforderungen an das Pattern Mining, die mit herkömmlichen Methoden nicht erfüllt werden können. Beispielsweise interessieren sich Wissenschaftler in der wissenschaftlichen Forschung mehr für unbekanntes Wissen, das sich normalerweise in Patterns verbirgt, die signifikant, aber nicht sehr häufig sind. Bestehende Itemset-Mining-Algorithmen sind jedoch für sehr häufige Pattern ausgelegt. Darüber hinaus müssen Wissenschaftler ein Experiment viele Male wiederholen, um die Reproduzierbarkeit sicherzustellen. Eine Reihe von Datensätzen wird gleichzeitig generiert und wartet auf Clustering, das eine unbekannte Anzahl von Clustern mit verschiedenen Dichten und Formen enthalten kann. Die Verwendung vorhandener Clustering-Algorithmen ist zeitaufwändig, da für jeden Datensatz eine Parameteroptimierung erforderlich ist. Viele wissenschaftliche Datensätze sind extrem verrauscht, d.h. Es gibt erheblich mehr Rauschen als In-Cluster-Datenpunkte. Die meisten vorhandenen Clustering-Algorithmen können nur Rauschen bis zu einem moderaten Pegel verarbeiten. Temporal Pattern Mining ist auch in der wissenschaftlichen Forschung wichtig. Bestehende zeitliche Pattern-Mining-Algorithmen berücksichtigen nur punktbasierte Ereignisse. Die meisten Ereignisse in der realen Welt basieren jedoch auf Intervallen mit einem Start- und einem Endzeitstempel. In dieser Arbeit wurden neuartige Pattern-Mining-Algorithmen für verschiedene Data-Mining-Aufgaben unter verschiedenen Bedingungen entwickelt.

Der erste Teil dieser Arbeit untersucht das Problem des Mining seltener Itemsets in Transaktionsdatensätzen. Im Gegensatz zu bestehenden Algorithmen für das Mining von häufige Itemsets konzentriert sich dieser Teil auf nicht so häufig auftretende Itemsets. Wir schlagen neue Algorith-

men NIIMiner, RaCloMiner und LSCMiner in dieser Arbeit vor, um solche Objektgruppen effizient zu identifizieren. NIIMiner verwendet den negativen Itemset-Baum, um alle Pattern, die unter einem bestimmten Unterstützungsschwellenwert aufgetreten sind, von oben nach unten zu extrahieren. RaCloMiner kombiniert vorhandene Bottom-Up-Algorithmen für häufiges Itemset-Mining mit einem Top-Down-Algorithmus für das Itemset-Mining, um eine bessere Leistung beim Mining weniger häufiger Pattern zu erzielen. LSCMiner untersucht das Problem des Mining seltener sogenannter closed Patterns.

Der zweite Teil dieser Arbeit untersuchte das Problem des intervallbasierten zeitlichen Pattern Mining in der Stream-Umgebung. Intervallbasierte zeitliche Pattern sind sequentielle Pattern, bei denen jedes Ereignis mit einer zeitlichen Start- und Endinformation ausgerichtet ist. Die Fähigkeit, intervallbasierte Ereignisse zu verarbeiten und Daten zu streamen, fehlt in bestehenden Ansätzen. In diesem Teil wird ein neuartiger intervallbasierter zeitlicher Pattern-Mining-Algorithmus für Stream-Daten beschrieben.

Der letzte Teil dieser Arbeit untersucht neue Probleme beim Clustering. Das erste Problem, das in diesem Teil behandelt wird, ist die Anpassungsfähigkeit von Clustering an wechselnde Formen von Clustern. In Anwendungen wie der wissenschaftlichen Datenanalyse müssen sich Wissenschaftler mit einer Reihe von Datensätzen befassen, die aus einem Experiment generiert wurden. Clustergrößen und -formen unterscheiden sich in diesen Datensätzen. Ein auf kNN-Dichte basierender Clustering-Algorithmus, kadaClus, wird vorgeschlagen, um Clusteralgorithmen in die Lage zu versetzen, verschiedene und sich verändernde Clusterformen zu erkennen, sodass Benutzer nicht die Parameter für jeden Datensatz anpassen müssen. Das zweite in diesem Teil untersuchte Problem ist das Clustering in einem extrem verrauschten Datensatz. Viele reale Datensätze enthalten erheblich mehr Rauschen als In-Cluster-Datenpunkte. Ein neuartiger Clustering-Algorithmus, kenClus, wird vorgeschlagen, um Cluster in beliebigen Formen aus extrem verrauschten Datensätzen zu identifizieren. Beide Clustering-Algorithmen basieren auf kNN und erfordern nur einen Parameter k.

In jedem Teil werden die Effizienz und Effektivität der vorgestellten Tech-

niken gründlich analysiert. Intensive Experimente sowohl mit synthetischen als auch mit realen Datensätzen werden durchgeführt, um die Vorteile der vorgeschlagenen Algorithmen gegenüber herkömmlichen Ansätzen aufzuzeigen.

Chapter 1

Introduction

During the past decades, information technology becomes ever more prevalent in nearly every aspect of our daily lives, such as medical biology, finance, and scientific research. The amount of data generated and stored continues to grow at an astounding rate in various fields, calling for a need for efficient and effective automatic data analysis tools. The interdisciplinary field of Knowledge Discovery in Databases (KDD) has thus emerged to unveil the potential patterns contained implicitly in the data automatically. In recent years, new requirements and demands on pattern mining arise in modern applications, which cannot be handled by conventional methods. For example, existing itemset mining algorithms are designed to extract very frequent patterns, but in some applications, people are more interested in unknown knowledge hidden in less frequent patterns. The study of pattern mining under new conditions has attracted much attention.

In this chapter, the main concepts of Knowledge Discovery in Databases are first introduced in Section 1.1. Afterward, new pattern mining requirements and demands studied in this thesis are briefly described in Section 1.3. This chapter concludes with an outline of the thesis in Section 1.4.

1.1 Knowledge Discovery in Databases

> *"Knowledge Discovery in Databases is the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data"*[36]

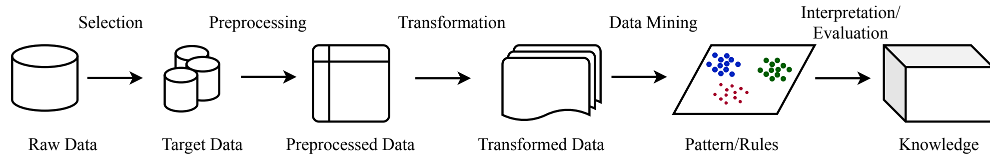


Figure 1.1: The KDD process.

According to the definition above, KDD is a process which aims at extracting meaningful and human interpretable patterns from data. It consists of an iterative sequence of steps as illustrated in the following (cf. Figure 1.1):

1. **Selection**. The first step of KDD process is to create a target data set by selecting a data set or focusing on a subset of attributes or data samples. The criteria of data selection often include data availability, quality, type, format, semantics, etc.

2. **Preprocessing**. Performing data cleaning operations, such as removing noise, handling missing data fields, etc. The data from multiple sources should be combined together.

3. **Transformation**. Finding useful features to represent the data, e.g. using feature selection approaches or transformation methods such as PCA to reduce the number of attributes or to find compact representation for the data.

4. **Data Mining**. Searching for patterns of interest in a particular representation form, e.g. clustering, classification and association rule mining, etc. Efficient and effective algorithms are used to extract novel, unknown and useful patterns from the transformed or original data.

5. **Interpretation and Evaluation**. Using visualization and knowledge representation techniques to present the interesting patterns extracted in the data mining step. Results are evaluated by domain experts.

Data mining is the core step of the KDD process. The notions "KDD" and "Data Mining" are often used as synonyms. By definition [36], the step of data mining consists of applying data analysis algorithms, with acceptable computational efficiency limitations, to produce a particular enumeration of patterns over the data. In general, existing data mining tasks can be classified as in the following:

- **Association Rules Analysis**: Discovering association rules showing attribute value conditions that are interesting (e.g. occur frequently together) in a given data set. Association rules express as a specified set of items appearing together in the same transaction with a certain support/probability.

- **Clustering**: Grouping objects in the data set such that objects in the same group are more similar to each other than to those in other groups.

- **Classification**: Learning a function from a given training data set to map new data objects to one or several classes in a predefined class set.

- **Prediction**: Learning a function from a given training data set to predict the numerical output values of new data objects.

- **Outlier Analysis**: Identifying irregular data objects in the data set that can not be described by the general model of the data, e.g. do not belong to any classes or clusters.

- **Characterization and Discrimination**: Summarization and comparion of general features of objects in the data set.

1.2 Pattern Mining in This Thesis

This thesis studies pattern mining under new conditions arisen in modern applications that are different from conventional scenarios. Many data mining tasks mentioned above will generate patterns. Topics discussed in this thesis mainly relate to three research areas: itemset mining, sequential pattern mining, and clustering. Itemset mining and sequential pattern mining extract itemsets and sequences as patterns. Clustering identifies clusters as patterns. Those three research areas are closely related to each other.

Itemset Mining

Itemset mining aims at discovering interesting co-occurrences of items or events as patterns in transactional databases. Each row of the database is a transaction which logs the types of items (events) that happened together. Various criteria can define the interestingness of a pattern. One of the most common criteria to define interestingness is the frequency (or support) of patterns. Frequent patterns are useful in generating association rules, which provide essential insights to the database. For example, given a database that tracks customers' transactions in a supermarket. Frequent itemset mining may find out patterns, such as people usually by beers and bread, which are useful for customer behavior analysis.

Other itemset mining tasks, such as high-utility itemset mining, are developed upon frequent itemset mining. In high-utility itemset mining, each item is also aligned with quantity and profit values. High-utility itemsets are not just frequently happened, but also bring a large amount of profit. Using the example above, high-utility itemset mining does not only tell us that customers buy beers and bread together, but also tell us that the combination of beers and bread brings a lot of profit.

Sequential Pattern Mining

Itemset mining extracts sets of items that occurred together as patterns. The order of items inside patterns is not considered. Sequential pattern mining

is introduced to tackle this drawback. By considering the order of events, sequential patterns can describe more complicated phenomena. For instance, instead of just knowing that customers always by beers and bread together, sequential patterns might tell us that customers tend to pick up beers before looking at bread. Besides transaction data, such sequential information is also essential in many other application areas, such as medical, financial, and scientific data analysis. For example, in medical data analysis, the order of symptoms' occurrences is essential for doctors. Most sequential pattern mining techniques are developed upon itemset mining.

In some cases, temporal information is also available. Patterns that only provide "before" and "after" information are not able to properly uncover insights in the database. Temporal pattern mining is developed upon sequential pattern mining to address this problem. Usually, sequential patterns are extracted first. Each sequential pattern is supported by a group of subsequences. Clustering techniques are applied to each group of subsequences to generate temporal patterns. Subsequences that support the same temporal pattern share the same order of events and similar temporal information.

Clustering

Clustering aims to group a set of objects in such a way that objects in the same cluster are more similar to each other than those in other clusters. Clustering is one of the most important tasks in pattern mining. Each identified cluster can be considered as a pattern.

Clustering and itemset mining are closely related to each other. Each item type in itemset mining can be seen as a dimension in clustering. Therefore, each transaction in itemset mining is a binary vector that records if an item type exists or not. A subset of all item types can form a subspace in which each transaction is a data point. If a transaction contains all items in the subset, its corresponding data point is located at $(1, 1, \ldots, 1, 1)$. If the number of data points at $(1, 1, \ldots, 1, 1)$ is larger than a given threshold, a frequent pattern is reported. In clustering, data points are numerical vectors and thus rarely locate in the same position. Usually, a distance function is

needed to measure the similarity between data points. Data points that are close to each other are assigned to the same group. If the number of data points in a group is large, it is returned as a cluster. Indeed, itemset mining is equivalent to subspace clustering with binary values.

1.3 New Challenges in Pattern Mining

New challenges in the three research areas mentioned above are investigated in this thesis, as listed in the following:

Rare Itemset Mining

Itemset mining aims at extracting interesting itemsets from transaction data sets. Each itemset contains items that occurred together in transactions. Conventionally, itemset mining implies frequent itemset mining, i.e., people only interested in very frequent itemset. However, with the expansion of the itemset mining technique's application scope, the new challenge, rare itemset mining, arise, which can not be addressed by frequent itemset mining methods. Rare itemset mining, or infrequent itemset mining, aims at extracting itemsets that appeared not that frequently, which may contain unknown knowledge. In applications such as scientific research or medical data mining, such kind of itemsets is more interesting than frequent itemset.

Interval-based Temporal Pattern Mining

Most existing sequential pattern mining algorithms aim at extracting frequent subsequences that appeared in a sequential data set. Each sequence in the data set records the order of events in some activities, such as the business process. However, using order information along is not enough to describe events that happened in the real world. The temporal information is also essential. Obviously, "Happened one second before" is different from "happened one day before". Furthermore, events in the real world are intervals aligned with two timestamps: the starting and ending times. Conventional sequential pattern mining algorithms treat events as single time

stamps. Some relations between events, such as a one event overlapping with another event, can not be modeled by sequential patterns.

*k*NN Density-based Clustering

Density-based clustering is one of the most important categories of clustering methods. It is widely used in identifying clusters in arbitrary shapes. Most density-based clustering approaches do not need to know the number of clusters in prior, which is a big advantage. However, they usually require more than one parameter, which makes parameter tuning more difficult. Recently, novel density-based clustering algorithms are proposed to address the problem by reducing the complexity to the use of one parameter. They introduce the concept of k Nearest Neighbor (kNN) and Reverse k Nearest Neighbor (RkNN) for density estimation. kNN density-based clustering approaches are useful but failed when dealing with the following challenges:

Shape Alternation Adaptability

Existing clustering pipelines assume that there is only one data set waiting for analyzing. However, in applications such as scientific research, experiments may be repeated many times, which leads to a series of data sets. Those data sets contain clusters with similar density but different shapes. When multiple data sets arrived, scientists need to turn parameters for each data set, run the clustering algorithm, and output the result, which is time-consuming.

High-noise Robustness

Another challenge is clustering on extremely noisy data sets. In real-world, many applications generate data sets that contain more noise than normal observations. Existing clustering algorithms only considered noise under a moderate level. None of then can produce meaningful results for high-noise data set.

1.4 Outline of this thesis

In this thesis, novel rare itemset mining, interval-based temporal pattern mining and clustering algorithms are proposed to address new challenges under different conditions. The content of this thesis is organized as in the following:

Chapter 1 presents an overview on the field of Knowledge Discovery in Databases and Data Mining. The three pattern mining research areas investigated in this thesis are introduced.

Part I presents efficient rare itemset mining algorithms.

Chapter 2 introduces the importance of rare itemset mining and the basic notions of itemset mining. Fundamental algorithms of frequent itemset mining are surveyed in detail.

Chapter 3 describes a novel data structure, the negative itemset tree, which is used as the basis for our novel rare itemset mining algorithm. A naive rare itemset algorithm using the negative itemset tree is described. Part of the contents in this Chapter has been published in [71], where Y.L. designed the research and implemented algorithms. F.R and T.S. gave important suggestions. Y.L., F.R. and T.S. wrote the paper.

Chapter 4 proposes a more advanced rare itemset mining algorithm based on the negative itemset tree. The idea of residual counts is used. Part of the contents presented in this Chapter has been published in [73], where Y.L. designed the research and implemented algorithms. F.R and T.S. gave important suggestions. Y.L., F.R. and T.S. wrote the paper.

Chapter 5 studies the concept of closed patterns in the context of rare itemset mining. A general bi-directional closed rare itemset mining framework is proposed. Part of the contents presented in this Chapter has been published in [74], where Y.L. and T.S. designed the research, implemented algorithms and wrote the paper.

Chapter 6 proposes a closed rare itemset mining algorithm based on the negative itemset tree. Part of the contents presented in this Chapter has been published in [72], where Y.L. designed the research and implemented

algorithms. F.R and T.S. gave important suggestions. Y.L., F.R. and T.S. wrote the paper.

Part II presents a novel interval-based temporal pattern mining algorithm for data streams.

Chapter 7 introduces the concept of interval-based temporal patterns and their relationship with sequential patterns and temporal patterns. Necessary notations and fundamental algorithms for sequential pattern mining and temporal pattern mining are described.

Chapter 8 describes the application and preliminaries of temporal pattern mining under a stream environment.

Chapter 9 proposes an incremental temporal pattern mining algorithm for data streams. Part of the contents presented in this Chapter has been published in [70], where Y.L. and M.H. designed the research. Y.L. implemented algorithms and wrote the paper. T.S. gave essential suggestions.

Chapter 10 adapts the algorithm introduced above to interval-based events streams. Part of the contents presented in this Chapter has been published in [70], where Y.L. and M.H. designed the research. Y.L. implemented algorithms and wrote the paper. T.S. gave important suggestions.

Part III describes two kNN density-based clustering algorithms designed for special use cases.

Chapter 11 introduces basic notations and concepts of density-based clustering. Fundamental algorithms are surveyed in detail.

Chapter 12 proposes a k Nearest Neighbor based clustering algorithm with shape alternation adaptability. Part of the contents presented in this Chapter has been published in [75], where Y.L. designed the research. Y.L. and Y.Z. implemented algorithms. Y.L., F.R. and Y.Z. wrote the paper. T.S. gave important suggestions.

Chapter 13 describes a k Nearest Neighbor based clustering algorithm for an extremely noisy dataset.

Part I

Rare Itemset Mining

Chapter 2

Introduction

2.1 Rare Itemset Mining

Frequent itemset (pattern) mining is a well-studied topic in the last decades and has been successfully applied to many application areas. Infrequent itemset mining (rare pattern mining), on the other hand, also shows its usefulness as a valid mining subject but attracts less attention. In many applications, frequent patterns represent known, mainstream behaviors while infrequent patterns are unknown knowledge which can be seen as hints for unexpected problems.

For example, in a medical database which contains logs about treatments, medications and symptomatic consequences of patients, the pharmacologists may already know that the medications A and B might help against disease C while frequent pattern analysis can also only yield this obvious information. The rare case that both medications $\{A, B\}$ together can lead to a lethal health condition will probably be pruned if we only consider frequent patterns. Another example is scientific data analysis. In scientific experimental data, frequent patterns usually represent phenomena or theories that scientists may already know. Such unknown information hidden in infrequent patterns is more important and interesting for scientists since they might lead to new knowledge or new theories. Infrequent pattern mining is also important in many other application scenarios such as financial fraud

detection or network security analysis. Such mining task is challenging while efficient approaches are absent.

Most existing itemset mining algorithms mainly focus on extracting frequent patterns. Those algorithms start the mining process from the empty set $\emptyset$. Frequent patterns are extracted by extending the candidate itemset recursively. This recursive process is terminated until the current candidate itemset is infrequent due to the well-known *anti-monotonicity* property. Anti-monotonicity property tells us that the support of an itemset is always larger or equal to the support of its super-set while smaller or equal to the support of its sub-set. Thus, when using frequent itemset mining algorithm to solve infrequent itemset mining problem, the corresponding minimum support value must be set to 1. All frequent patterns have to be traversed first before accessing infrequent itemsets, which wastes unnecessary traversing time.

Few infrequent itemset mining approaches are also proposed in recent years while most of them are adapted from frequent itemset mining algorithms which also start from the empty set $\emptyset$. In principle, such kind of approaches must perform the same as frequent itemset mining algorithms since they also traverse frequent itemsets first. Only a few algorithms start the mining process from long infrequent itemsets so that we can avoid traversing the frequent pattern part. Items are removed from candidate patterns recursively and the algorithm stopped until current candidates become frequent. Such kind of algorithms could be more efficient since no time is wasted on traversing frequent itemsets.

Besides, many condensed representations have been proposed for frequent patterns to reduce the size of final results and increase the performance. Adapting condensed representation for infrequent patterns is beneficial. However, there is a lack of effective approaches focusing on extracting condensed representations.

2.2 Related Works

In the research field of itemset mining, most approaches discover the occurrences of frequent itemsets and association rules. A traditional approach is Apriori [3] by Agrawal and Srikant, which extends the size of candidate itemsets step-wise. The search-space is pruned by utilizing a minimum support threshold. To avoid explicit candidate generation, FP-growth [51] by Han et al. uses a data structure called FP-Tree. Both approaches mine frequent itemsets. However, rare itemsets only occur in the result for a low support threshold, which yields very large results and makes mining for rare itemsets inefficient.

Apriori-like Approaches

First of all, shifting from a static minimum support threshold to a dynamic minimum support or avoiding a minimum support at all offers some chance to include rare itemsets in the result. Typical approaches here are to attach a separate minimum item support for each item in the database [67] or let the minimum supports adapt in case of lower frequencies or changing significance [106][101][93]. Seno and Karypis [88] propose an approach to favor smaller itemsets over larger ones. Koh and Rountree [61] inverted the idea of Apriori by defining a maximum support threshold. Szathmary et al. followed this idea of mining the rare itemsets with ARIMA[92], which uses the pruned itemsets of Apriori in a first mining step to generate rare itemset candidates bottom-up in a second step. FRIMA[56] also follows a bottom-up traversal based approach. Wang et al.[100], Cohen et al.[29], Xiong et al. [103] and Haglin et al. [48] propose methods to avoid a minimum support threshold. Jain et al.[58] proposed a prepruning step to reduce the search space for rare pattern mining. All these methods employ the bottom-up strategy, i.e., scan all frequent itemsets first, which is not efficient.

To the best of our knowledge, AfRIM[92] and Rarity[95] are the only two approaches traverse the search space in a top-down fashion. However, Apriori-like paradigm is employed in these approaches which leads to a huge number of candidates during mining process.

Pattern-growth Approaches

Approaches in the second field are mainly based on the FP-growth [51] algorithm, such as the RP-tree algorithm [96]. The RP-tree is constructed similar with the FP-tree but for rare items. Transactions with only frequent items are pruned. It is more efficient than the previous works due to its avoidance of candidate generation. However, only a subset of rare itemsets is provided. Rare itemsets made up of frequent items are ignored. This kind of itemsets are interesting especially in dense dataset. The Inverse FP-Tree approach by Gupta et al. [44] also utilizes FP-growth and mines minimally infrequent itemsets. These are infrequent itemsets and any subset is frequent. Kiran et al.[59] used multiple minimum support values in the FP-tree. The tree is constructed in descending item order regarding the minimum item support. Lavergne et al. [65] consider user interest to focus the search on certain rule sets. The pattern-growth based approaches increase the performance due to their avoidance of candidate generation. However, they still use some sort of user-defined input or constraint to mine for rare items. A detailed overview about rare itemset mining methods can be found in [60] by Koh and Ravana.

Knowledge Embedding

The last field here utilizes knowledge about the data and it focuses specific rules and itemsets. Liu et al. [68] prune rules using a chi-square significance test. Bayardo et al. [9] proposed the Dense-Miner, which introduces rule constraints in addition to minimum support. This allows to mine rules with low support, however some knowledge about the data is needed to specify the constraints. Li et al.[66] developed emerging patterns, which involves using predefined consequences to iteratively remove rules. Rahal et al. [84] supplied an approach that yield the highest support rules by only specifying the minimum confidence. Koh et al. [62] developed MIISR, which is based on the apriori-inversed approach [61] and uses an absolute maximum support threshold to mine rare rules with a candidate itemset. Cagliero et al. [17] uses weights in transactional databases to mine infrequent items. Although these approaches can be used to mine rare or sporadic itemsets and rules,

they assume some background knowledge about the data in advance.

2.3 Powerset Lattice Traversing

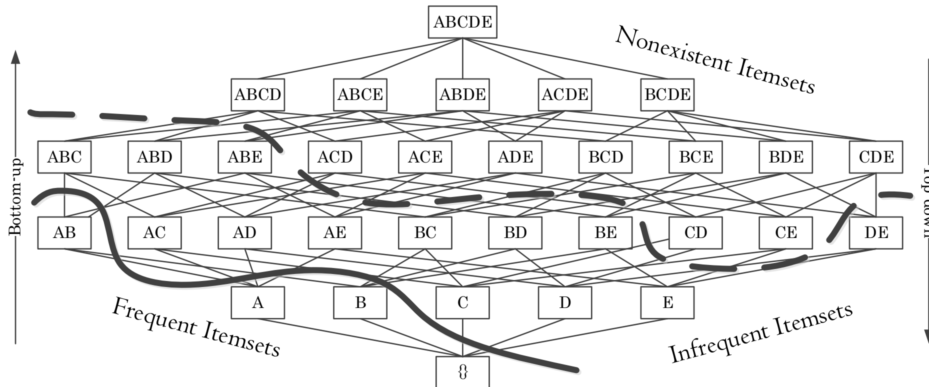


Figure 2.1: Powerset lattice. The minimum support splits the lattice into three parts.

In general, both frequent itemset mining and infrequent itemset mining algorithms can be seen as a process of traversing through the itemset powerset lattice, where each node in the lattice represents an itemset. Figure 2.1 gives an example lattice. With some minimum support threshold, the lattice can be divided into three parts: frequent part, infrequent part, and nonexistent part. The frequent part stays at the bottom of the lattice while the infrequent part takes a relatively higher position.

Priority \ Direction	Bottom-up	Top-down
Breadth-first	Apriori[3]	Rarity[95]
Depth-first	FP-Growth[51]	–

Table 2.1: Powerset lattice traversing strategies and example algorithms.

Counting the support of all possible itemsets is impossible since the size of the powerset lattice is exponential. Efficient mining algorithm will fully utilize given constraints, like *minSup*, and only traverse nodes that fulfill the criteria. Different algorithms differ in their traversing directions and

priorities, as shown in Figure 2.1. In summary, all frequent itemset mining algorithms and most existing rare itemset mining algorithms are traversing bottom-up. Some rare itemset mining algorithms use top-down and breadth-first traversing.

In this part of the thesis, the problem of rare itemset mining is studied. Several novel top-down depth-first traversing based algorithms are described. The basic data structure, negative itemset tree, is introduced in Chapter 3. A naive rare itemset mining algorithm using the negative itemset tree is also introduced. Then Chapter 4 introduces an advanced rare itemset mining algorithm using residual counts on the negative itemset tree. Chapter 5 provides a framework to combine both top-down and bottom-up traversing algorithms. Finally, Chapter 6 describes how to utilize the negative itemset tree for closed rare itemset mining.

2.4 Frequent Itemset Mining Algorithms

As mentioned above, existing rare itemset mining algorithms are mainly adapted from frequent itemset mining algorithms. In this section, a brief survey on fundamental frequent itemset mining algorithms is given.

When itemset mining was first introduced decades ago, it was mainly used for association rules mining on transactional datasets. In order to generate significant association rules, people only focus on patterns that occurred very frequently in the data set. Table 2.2 gives an example of transactional dataset. Let $\mathcal{I}$ be the universe of items, the transactional dataset $\mathcal{T}$ contains $N = 6$ transactions. Each transaction $T \in \mathcal{T}$ is a non-empty subset of $\mathcal{I}$. An itemset (pattern) X is also a subset over $\mathcal{I}$. Its support is defined as the number of transactions $T \in \mathcal{T}$ such that $X \subseteq T$, denoted as $X.supp = |\mathcal{T}(X)| = |\{X \subseteq T, T \in \mathcal{T}\}|$. Frequent itemset mining aims to identify all itemsets with support greater than or equal to the given minimum support threshold *minSup*.

Conventional frequent itemset mining algorithms can be divided into two categories: Apriori-based approaches and Divide-and-conquer-based approaches. Both of them can be seen as a traversing process on the powerset

Tid	Transactions
1	A B C
2	A B D
3	B C
4	A B
5	A B E
6	D E

Table 2.2: An example of transaction database

lattice of itemset. They start the traversing from the bottom of the lattice (the empty set). The Apriori approach employed a breadth-first traversing while the Divide-and-conquer approach employed a depth-first traversing.

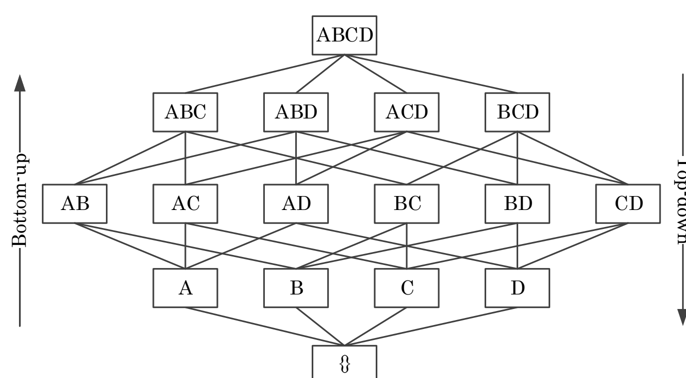


Figure 2.2: A simple powerset lattice with four distinct items.

Apriori Approach

The naive way to generate all frequent itemsets is by counting the frequency of all possible subsets of $\mathcal{I}$ in the dataset $\mathcal{T}$. However, such an approach is too expensive since there are $2^{|\mathcal{I}|}$ distinct itemsets for $|\mathcal{I}|$ items.

Apriori approach [3] was the first efficient frequent itemset mining algorithm. It contains two major steps: candidate generation and candidate pruning. The key principle used in this approach is the so-called Apriori principle. It described the anti-monotonicity property of itemsets.

Theorem 2.4.1 (Apriori Principle (anti-monotonicity))

- Any (non-empty) subset of a frequent itemset X is also frequent:

$$\forall X' \subseteq X : X.supp \geq minSup \Rightarrow X'.supp \geq minSup$$

- Any superset of a non-frequent itemset X is also non-frequent:

$$\forall X'' \supseteq X : X.supp < minSup \Rightarrow X''.supp < minSup$$

The basic idea of the Apriori algorithm is to use the anti-monotonicity property to prune infrequent itemsets. Let l-itemset denotes an itemset with l items. Apriori algorithm starts from frequent 1-itemset (frequent items). In each iteration, it combines two frequent l-itemsets together to generate a candidate $(l+1)$-itemset. Let P and Q be two frequent l-itemsets, and items are sorted by any order. P and Q are joined if they share the same $(l-1)$ items. Such a join strategy is complete. All frequent $(l+1)$-itemsets are contained due to the anti-monotonicity property. It is also selective. The number of candidate $(l+1)$-itemsets is much smaller than the number of all $(l+1)$-itemsets. The pruning step also employed the anti-monotonicity property. Instead of checking the support of every candidate $(l+1)$-itemsets, the Apriori algorithm removes candidate $(l+1)$-itemsets that contain an infrequent l-subset.

As an example, given the dataset in Table 2.2 and let $minSup = 2$. The set of 1-itemsets, i.e., frequent items, is $L_1 = \{A, B, C, D, E\}$. The Apriori algorithm starts from this set of itemsets. In the candidate generation step, itemsets in L_1 are joined with themselves, which leads to the candidate set of 2-itemsets $C_2 = \{AB, AC, AD, AE, BC, BD, BE, CD, CE, DE\}$. The pruning step first filtered out itemsets in C_2 that contain infrequent 1-subset. In this example, all 1-itemsets are frequent and therefore, no candidates are filtered out. Then Apriori algorithm checks the support of each candidate left in C_2 and generates the set of frequent 2-itemsets $L_2 = \{AB, BC\}$. The above generation and pruning steps are repeated until the set of frequent l-itemsets is empty. In this example, Apriori algorithm will generate $C_3 =$

$\{ABC\}$ and $L_3 = \emptyset$, and then stop. The final frequent itemsets list is $\{A, B, C, D, E, AB, BC\}$. Algorithm 2.1 illustrates the detail of the Apriori algorithm.

Algorithm 2.1: Apriori Algorithm

```
   // C_l ←candidate itemsets of size l
   // L_l ←frequent itemsets of size l
1  L_1 ← {Frequent items} ;
2  for l = 1; L_l ≠ ∅; l + + do
       // Candidate generation step.  Join L_l with itself to
          produce C_{l+1}
       // Discard (l + 1)-itemsets from C_{l+1} that contain
          non-frequent l-itemsets as subsets
3      C_{l+1} ←candidates generated from L_l;
       // Candidate pruning step.
4      foreach c ∈ C_{l+1} do
5          foreach s ⊂ c do
6              if s ∉ L_l then
7                  Delete c from C_{l+1};
8              end
9          end
10     end
11     foreach Transaction T ∈ 𝒯 do
12         Increase the support of all candidates in C_{l+1} that are
             contained in T;
13     end
14     L_{l+1} ←candidates in C_{l+1} with support at least minSup;
15 end
16 return ∪_l L_l
```

Generally speaking, the Apriori algorithm performs a breadth-first traversing on the powerset lattice. All frequent itemsets on one level are identified before accessing the next level. Such a breadth-first traversing strategy leads to huge candidate sets. To discover a frequent pattern of 100 items, one needs to generate $2^{100} \approx 10^{30}$ candidates. Furthermore, if l is the length of the longest pattern, the Apriori algorithm needs to scan the database for l times.

Divide-and-conquer Approach

To overcome the Apriori algorithm's problem, frequent itemset mining algorithms without the candidate generation step are proposed. The main idea is to employ the divide-and-conquer paradigm with database projection. Assuming items are sorted in any order, the database can be projected onto each item step by step, which divides the frequent itemset mining problem on the original dataset into a set of problems on smaller subdatasets. The prefix of projection forms valid itemsets, which avoids the expensive candidate generation step. A naive divide-and-conquer approach using database projection is illustrated in Algorithm 2.2.

Algorithm 2.2: Database Projection

Input: Database $\mathcal{T}$, *minSup*
Output: The set of frequent itemsets $\mathcal{F}$

```
1  F ← Projection(T, ∅, minSup);
2  return F;
3  Function Projection(Database T′, Prefix pre, minSup):
       // Let I′ be the set of distinct items in T′, T′|i be
          the projected database of T′ on item i ∈ I′.  T′|i is
          initialized to ∅.
4      foreach Transaction T ∈ T′ do
5          if i ∈ T then
6              T′ ← {i′ ∈ T, i′ ≻ i}   // i′ are items after i, in any
                  order
7              Add T′ to T′|i;
8          end
9      end
10     F′ ← ∅;
11     foreach T′|i do
12         if |T′|i| ≥ minSup then
13             pre′ ← pre ∪ {i};
14             Add pre′ to F′;
15             F′ ← F′ ∪ Projection(T′|i, pre′, minSup);
16         end
17     end
18     return F′
19 end
```

As an example, let $minSup = 2$ and the dataset is shown in Table 2.2. Assuming that items are sorted in lexicographic order, the projected sub-database on item A is $\{BC, BD, B, BE\}$. The size of this projected database is larger than $minSup = 2$. Therefore, $\{A\}$ is a frequent itemset. Then, the sub-dataset is recursively projected onto items B, C, D, and E. Here we show the projection on B, which gives a projected database $\{C, D, \emptyset, E\}$ with prefix $\{AB\}$. The size of this projected dataset is also larger than $minSup$, so that the itemset $\{AB\}$ is frequent. The database project process is applied recursively on all items. Frequent itemsets are generated without the candidate generation step.

The divide-and-conquer approach described above performs a depth-first traversing on the powerset lattice, which is different from the breadth-first traversing of the Apriori approach. In general, most efficient frequent itemset mining algorithms also followed the divide-and-conquer approach. For example, the FP-Growth [51] algorithm, which is one of the most famous frequent itemset mining algorithms, utilizes the database projection technique. A special data structure, so-called FP-Tree, is employed to compress the database and provide better performance.

Approximation Summarization Approach

Algorithms mentioned above extract the exact set of frequent itemsets. However, when the size of the dataset becomes really large, most approaches are in trouble. To further improve the efficiency of frequent itemset mining, approximation algorithms are proposed. Such algorithms do not guarantee that identified itemsets are frequent. For example, the algorithm SLIM [90] proposed to mine interesting itemsets based on the minimum description length (MDL). It followed the idea of compression, which considered the set of interesting itemsets as a variable-length code table for encoding the original transaction dataset.

Chapter 3

Negative Itemset Tree

As mentioned above, conventional itemset mining algorithms use a bottom-up traversing strategy, which is not suitable for the task of rare itemset mining. To address this problem, the concept of negative itemset is used. Mining negative itemset is also an important topic in itemset mining. It aims at extracting interesting patterns that are not in the dataset.

In this thesis, negative itemsets are used to form negative dataset. A bottom-up based algorithm is proposed to mine frequent itemset in the negative dataset, which is equivalent to top-bottom traversing in the original dataset. To achieve this goal, a novel data structure, negative itemset tree, is developed.

In this chapter, the negative itemset tree is introduced. A naive rare itemset mining algorithm using the negative itemset tree is described. Experiments show that the naive method's performance is much better than using conventional methods on the rare itemset mining task. Parts of the material presented in this Chapter have been published in [71].

3.1 Preliminaries

Consider $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ to be a set of all distinct items. Any non-empty subset $X \subseteq \mathcal{I}$ is an *itemset*. Any itemset X with size $|X| = k$ is referred to as a *k-itemset*. A tuple $T = (tid, X)$ is called a *transaction*, where *tid* is the transaction identifier. For simplicity, a transaction T also refers to its itemset X if not specified. Any non-empty itemset $Y \subseteq X$ is *contained* by a transaction $T = (tid, X)$ and we just write $Y \subseteq T$. A set of transactions establish a *transaction database* $\mathcal{T}$. Table 3.1 illustrates an example transaction database.

Tid	Transactions
1	A B C
2	A B D
3	B C
4	A B
5	A B E
6	D E

(a)

Tid	Transactions
1	¬D ¬E
2	¬C ¬E
3	¬A ¬D ¬E
4	¬C ¬D ¬E
5	¬C ¬D
6	¬A ¬B ¬C

(b)

Table 3.1: Example transaction database (a) and its corresponding neg-rep dataset (b).

Given a transaction database $\mathcal{T}$, the (absolute) *support* of an itemset X is defined as the number of transactions $T \in \mathcal{T}$ containing X: $X.supp = |\{T \in \mathcal{T} | X \subseteq T\}|$. The minimum support threshold (*minSup*) categorize all itemsets (patterns) into three types: *nonexistent, infrequent* and *frequent*. An itemset X is *infrequent* if and only if: $0 < X.supp < minSup$. Otherwise, it is *frequent* ($X.supp \geq minSup$) or *nonexistent* ($X.supp = 0$). With $|\mathcal{I}|$ distinct items, a dataset contains $2^{|\mathcal{I}|}$ patterns while most of them are nonexistent.

We are aiming at extracting all infrequent itemsets with support smaller than the given minimum support threshold and larger than 0. However, it is still worth to note that extracting low support patterns does not necessarily mean to extract all infrequent patterns. User might interest in patterns with support fall in a range, for example between 10 and 20, which is still small

in a dataset with hundred thousands of long transactions. A bottom-up based approach can extract patterns in this range with minimum support threshold set to 10 rather than 0. However, the number of frequent patterns to be identified is still massive. In contrast, a top-down based approach will be more efficient since it only extracts "infrequent" patterns occurred less than 20 times. In our experiments later, different threshold values are assigned to simulate the low support pattern mining scenario.

Neg-Rep and Negative Itemsets

In the conventional notation of an itemset, each symbol expresses the *existence* of an item. For example, given an itemset $X = \{A, B, C\}$. Its notation implies that items A, B and C exist in X. For simplicity, we call symbol of items in this notation as *positive items* and the notation as *positive itemsets*. Similarly, we can also represent the itemset X by utilizing those items that *do not exist in* X. This *negative representation* is the basic concept for support counting in our mining process.

Definition 3.1.1 (Negative Item)
Given the set of items $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$, the corresponding negative item of $i \in I$ is denoted as $\neg i$.

The symbol $\neg$ is used to represent the idea of *not exist*, which can be dropped in some notations below for simplicity.

Definition 3.1.2 (Neg-Rep Itemset and Negative Itemset)
Given a positive itemset $X = \{x_1, x_2, \ldots, x_n\} \subseteq \mathcal{I}$, its neg-rep (**neg**ative **rep**resented) itemset is the set of items that X does not have, denoted as $\overline{X} = \{\neg i | i \in I \land i \notin X\} = \mathcal{I} \setminus X$. The *negative itemset* of X is denoted as $\neg X = \{\neg x_1, \neg x_2, \ldots, \neg x_n\}$.

A positive itemset X and its neg-rep itemset $\overline{X}$ are two different notations of the same pattern. This concept is important for the support definition described later. Converting each transaction in $\mathcal{T}$ into the their neg-rep itemset yields the corresponding *neg-rep transaction database* $\overline{\mathcal{T}}$ (Table 3.1b).

Two support values, *intersection support* and *joint support*, are defined on $\mathcal{T}$ and $\overline{\mathcal{T}}$ respectively.

Definition 3.1.3 (Intersect Support and Joint Support)
Given a non-empty itemset $X = \{x_1, \dots, x_n\}$:

- The intersect support of X in a transaction database $\mathcal{T}$ is the number of transactions that contains **all** items of X: $X.isupp = |\{T \in \mathcal{T} \mid x_1 \in T \wedge x_2 \in T \wedge \dots \wedge x_n \in T\}|$.
- The joint support of the negative itemset $\neg X$ is defined in the corresponding neg-rep dataset $\overline{\mathcal{T}}$. It is the number of transactions that contains **at least one** item of $\neg X$: $\neg X.jsupp = |\{\overline{T} \in \overline{\mathcal{T}} \mid \neg x_1 \in \overline{T} \vee \neg x_2 \in \overline{T} \vee \dots \vee \neg x_n \in \overline{T}\}|$.

Obviously, the intersect support is equivalent to the original definition for (absolute) support, i.e.: $X.isupp = X.supp$. The join support, on the other hand, has the following property:

Theorem 3.1.1
Given itemset X, dataset $\mathcal{T}$ and the corresponding neg-rep dataset $\overline{\mathcal{T}}$, then $X.isupp = |\mathcal{T}| - \neg X.jsupp$.

Proof. If a transaction T does not contain X, then $T \not\supseteq X \Leftrightarrow \overline{T} \cup \neg X \neq \emptyset$. Based on the definition of joint support, $\neg X.jsupp = |\{\overline{T} \in \overline{\mathcal{T}} | \overline{T} \cup \neg X \neq \emptyset\}| = |\{T \in \mathcal{T} | T \not\supseteq X\}|$. Furthermore, $X.isupp = |\{T \in \mathcal{T} | T \supseteq X\}|$. Thus, $X.isupp + \neg X.jsupp = |\mathcal{T}| \Rightarrow X.isupp = |\mathcal{T}| - \neg X.jsupp$. □

Thus, the support of patterns in $\mathcal{T}$ can be computed equivalently using the joint support of neg-rep patterns in $\overline{\mathcal{T}}$. Mining rare itemset with support smaller than a given threshold $minPts$ in $\mathcal{T}$ is the same a mining frequent itemset with support larger than the threshold $|\mathcal{T}| - minPts$ in $\overline{\mathcal{D}}$.

3.2 Negative Infrequent Itemset Tree

In this section, I describe the naive rare itemset mining algorithm using *negative itemset tree (NI-tree)*. The NI-tree is a prefix tree which compresses

the neg-rep database. Each node $n = [\neg i, c, l]$ is a triple consisting of a negative item $\neg i$, a count value c and a successor list l. The root node $r = [is, c, l]$ stores an itemset is which is empty at the beginning. Direct successors of the root, i.e. in the list $r.l$, are called the $1st$-*layer* nodes. All negative items on a path from the root to any node form a negative itemset by concatenation. The count value c represents the number of occurrences of the corresponding itemset. The count of the root node represents the size of the database. The negative itemset tree is built in the same way as the well known FP-tree by scanning the whole neg-rep database. Negative items are sorted in descending order which leads to a smaller tree. The corresponding negative itemset tree for the database in Table 3.1 is shown in Figure 3.1a.

Infrequent itemsets are extracted by recursively subtracting (excluding) nodes from the NI-tree. The new tree after subtraction, called deducted tree (*de-tree*), is also an NI-tree. Items that have been subtracted so far are stored in the root node. Figure 3.1 illustrates two de-trees by excluding $\neg C$ and $\neg C, \neg D$ from the original NI-tree, respectively.

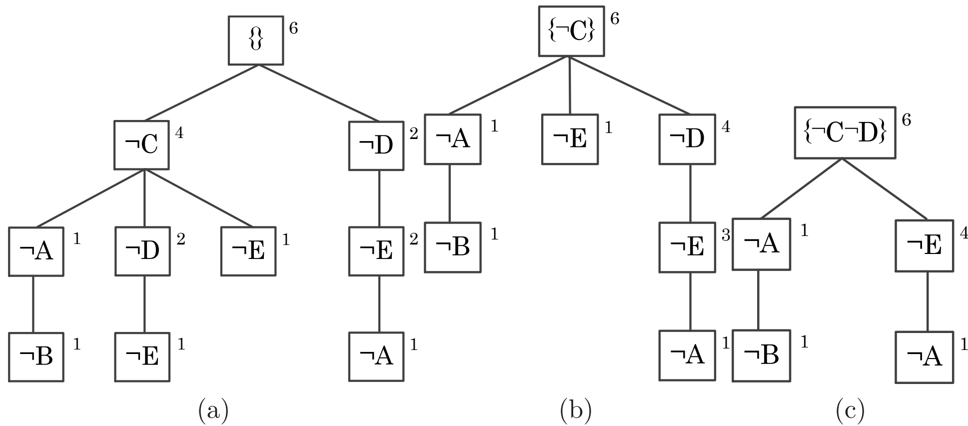


Figure 3.1: Examples of (a) Negative Itemset tree and its corresponding de-tree by excluding (b) $\neg C$ and (c) $\neg C, \neg D$. (c) is also a de-tree of (b).

When new negative item is added to the root node, corresponding nodes in the tree are also removed. Sub-trees rooted at the excluded node will be attached and merged recursively to the node above. For example, from Figure 3.1a to Figure 3.1c, the path $\boxed{\neg E}$ is attached to the root node and

the path $\boxed{\neg D}\boxed{\neg E}$ is merged to the existing path. Algorithm 3.1 illustrates the pseudo code of the subtraction process.

Algorithm 3.1: Naive NI-tree Substraction

```
   Input: NI-tree Root Node r, Negative Item ¬i
   Result: New Root Node r'
   /* New root with identical itemset, count but empty
      children list                                          */
1  r' ←new NI-treeNode(r)
   /* Add the given negative item to the new root            */
2  Add ¬i to r'.is
3  foreach Child node n ∈ r.l do
4  |  if n.¬i ∈ r'.is then
5  |  |  TraverseSubtree(r', n)
6  |  else
7  |  |  Add n to r'.l
8  |  end
9  end
10 return r'
11 Procedure TraverseSubtree(NI-tree Root r, NI-tree Node n)
12 |  foreach Child n' ∈ n.l do
13 |  |  if n'.¬i ∈ r.is then
14 |  |  |  TraverseSubtree(r, n')
15 |  |  else
16 |  |  |  Add n' to r.l
17 |  |  end
18 |  end
19 end
```

Each NI-tree during the mining process corresponds to a specific itemset X whose neg-rep itemset $\overline{X}$ is stored in the root node. Other negative items, which remained in the tree, form the corresponding negative itemset $\widetilde{X}$.

Theorem 3.2.1
Given a NI-tree root r and an itemset X with $\overline{X} = r.is$, the joint support of the negative itemset is: $\widetilde{X}.jsupp = \sum_{n \in r.l} n.c$

Proof. Since $r.is = \overline{X}$, paths remained in the tree correspond to transactions containing at least one item in $\widetilde{X}$. Thus, the joint support of $\widetilde{X}$ is the number

of transactions remained. Furthermore, the NI-tree is a prefix tree. Let $n.T$ denotes the set transactions in CTD that contribute to the count of the corresponding itemset, we have: $n'.T \subseteq n.T$, if node n' is in the subtree of node n. Thus, the count of a node summarize all counts in its subtree, $\widetilde{X}.jsupp = \sum_{n \in r.l} n.c.$ □

For example, the NI-tree in Figure 3.1c corresponds to the itemset $\{A, B, E\}$ as its neg-rep itemset $\{\neg C, \neg D\}$ is excluded. The joint support of the corresponding negative itemset $\{\neg A \neg B \neg E\}$ is $1 + 4 = 5$. The joint support is used as the stopping criteria for our mining process.

In practice, a pseudo-subtraction strategy is employed to avoid unnecessary tree construction. Firstly, only a new root node is created in the subtraction step. Related child nodes will be attached to the new root immediately. The subtraction process remains correct since the value of the joint support is not affected, as illustrated in Figure 3.2a. Secondly, the subtraction process is terminated when no negative item in the 1st-layer needs to be excluded. For example, excluding $\neg D$ from the original tree will lead to the tree in Figure 3.2b. The remaining $\boxed{\neg D}$ node will only be removed after the removing of node $\boxed{\neg C}$. Again, the correctness is not affected since the computation of joint-support only depends on 1st-layer nodes.

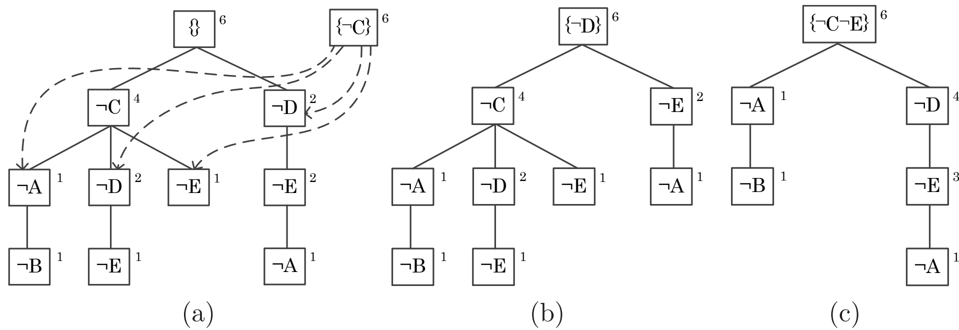


Figure 3.2: (a) The pseudo-subtraction of $\neg C$ in practice, only root is created, no merging happened. (b,c) Examples of de-tree by excluding $\neg D$ and $\neg C \neg E$. Only 1st-layer nodes are checked and removed.

Algorithm 3.2 illustrates the overall procedure of the naive negative itemset tree miner. Negative items are excluded one by one recursively. The whole

Algorithm 3.2: Naive Miner

```
   Input: Transaction Database 𝒯, Minimum Support minSup
   Result: Infrequent Itemset List L
 1 r ←NI-tree(𝒯)
 2 ϵ ← |𝒯| − minSup
 3 L ← ∅
 4 Extend(r, ϵ, L)
 5 return L
 6 Procedure Extend(NI-tree Node r, Threshold ϵ, L)
 7   foreach Negative Item ¬i ∈ I \ r.is do
       /* Items to be excluded must after items exist r.is
          */
 8     if r.is ≺ ¬i then
 9       r′ ←NItreeSub(r,¬i)
10       if JointSupport(r′)≥ ϵ then
11         Add \overline{r′.is} to L
12         Extend(r′, ϵ, L)
13       end
14     end
15   end
16 end
```

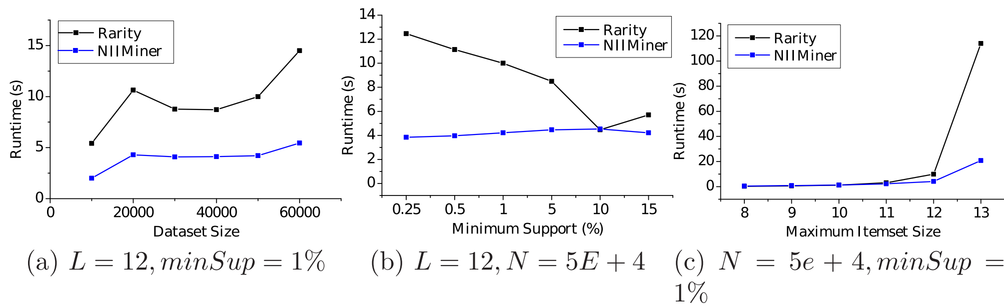
process terminated until the joint support is lower than the given threshold.

3.3 Experimental Evaluation

The naive method described above is compared to the Rarity [95] algorithm, which is the state-of-the-art top-down Apriori-like infrequent itemset mining approach. Other rare pattern mining algorithms, such as ARIMA [92] and AfRIM [1], are not included in our experiments since they use a breadth-first search with bottom-up traversal similar to Rarity, and their performance compared to Rarity have been conducted in detail in Rarity papers.

All algorithms are implemented in Java and executed on an Intel Core i7 3.4 GHz machine running Ubuntu 16.04. Real dataset Connect-4 from

UCI repository[1] is used in our experiment. Our experiments are conducted on different dataset sizes, minimum supports, and maximum itemset sizes. Given dataset size set to N and the maximum itemset size set to L, the first N transactions in a dataset and the first L items in each itemset are used. We limit the maximum itemset size since otherwise, the Rarity algorithm won't be able to finish the mining task on our machine. Experiment results are illustrated in Figure 3.3



(a) $L = 12, minSup = 1\%$ (b) $L = 12, N = 5E + 4$ (c) $N = 5e + 4, minSup = 1\%$

Figure 3.3: Runtime on Connect-4 dataset

Taking the advantage of the depth-first traversal over the breadth-first traversal, the new method is significantly faster than Rarity under most of the settings. The only exception is under large minimum support value and small maximum itemset size settings. This is because the Connect-4 dataset has a limited number of unique items, which leads to fewer candidates during candidate generation step. Rarity also suffers from its pruning step under small minimum support settings. As shown in Figure 3.3, the runtime performance of Rarity is increasing when the minimum support is decreasing, which is unusual as a top-down based approach. It costs much more time but output less rare patterns with a smaller minimum support. In real applications, extracting infrequent itemsets usually implies to find itemsets with small support rather than large support. In summary, the new approach, with the ability to extract the complete set of rare patterns, is very efficient.

[1]https://archive.ics.uci.edu/ml/index.php

3.4 Conclusion and Future Works

The novel rare itemset mining algorithm has proven to solve the problem of rare itemset mining in an efficient and successive manner. By utilizing the negative representations of rare itemsets to frequent itemsets, this task is addressed from its dual perspective. However, this approach traverses all infrequent itemsets, including those patterns with support equal to 0, known as *nonexistent patterns*. There are a huge number of nonexistent patterns, especially in a sparse dataset, which should be skipped since they are not important in many applications. It has to spend a lot of time on traversing those patterns while Rarity only returns rare patterns that exist and could be faster than this new method on sparse datasets. Further investigations are necessary to avoid the expensive traversing step on nonexistent patterns.

Chapter 4

Negative Itemset Tree with Residual Counts

The naive method described in Chapter 3 does not fully utilize the advantage of a negative itemset tree. When computing the joint support, the naive method needs to traverse all nodes connected to the root in the NI-tree. In this Chapter, a more efficient rare itemset mining algorithm: **N**egative **I**nfrequent **I**temset tree miner (NIIMiner), is introduced which utilizes the NI-tree more efficiently. This approach is inspired by the diff-set in [108]. Instead of computing the joint support of rare itemsets directly, the residual count is computed. Experiments show that the new residual counting strategy is more efficient. Parts of the material presented in this Chapter have been published in [73].

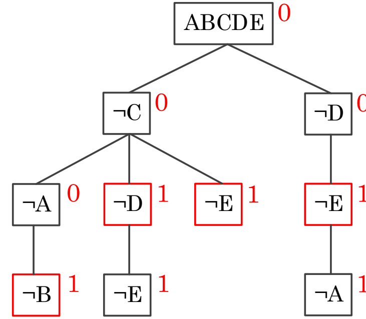


Figure 4.1: Negative Itemset tree built based on dataset in Figure 6.1. Termination nodes are marked in red.

4.1 Negative Itemset Tree and Support Counting

As mentioned in Chapter 3, *negative itemset tree (NI-tree)* is a prefix tree generated based on the neg-rep database $\overline{\mathcal{T}}$ which summarizes the itemset information. The mining process extracts infrequent itemsets from the tree by deleting nodes recursively. For residual counting, the NI-tree is adapted accordingly.

Each node $n = [\neg i, c, l]$ is a triple consisting of a negative item $\neg i$, a count value c and a child list l. The root node $r = [is, c, l]$ stores an itemset $\{is\}$, which is initialized as $\mathcal{I}$. The root node is on the 0*th-layer*. Nodes that are direct successors of the root, i.e. in the list $r.l$, are called the 1*st-layer* nodes and so on. All negative items on a path from the root to any node form an itemset with negative items. The count value c is the number of neg-rep transactions that end at the node. l is the list of child nodes.

To build a negative itemset tree, the dataset $\mathcal{T}$ is converted into its neg-rep database $\overline{\mathcal{T}}$. Negative items in each transaction are sorted in descending order based on their occurrence in $\overline{\mathcal{T}}$. Transactions in $\overline{\mathcal{T}}$ are inserted to the NI-tree one by one in ascending order with respect to their length.

This time, only the count c of the last node in each insertion is increased by 1. Furthermore, if the count of all nodes on the path during an insertion are 0, then the last node will be marked as a *termination node*, since it is the end of a negative itemset. In another words, termination nodes are the first

node with non-zero count on each path from root to leaf. The corresponding NI-tree built based on the dataset in Figure 6.1 is shown in Figure 6.2.

Removing items from the root node, as well as the corresponding nodes in the NI-tree, lead to a new NI-tree, called the deducted tree (*de-tree*). Detailed excluding process is shown in Algorithm 4.1. The support of a pattern can be computed efficiently during such removing process. The itemset $\{is\}$ in the new root node of the de-tree is a new pattern.

Given the initial NI-tree constructed from the neg-rep dataset $\overline{\mathcal{T}}$, we first check each node on the $1st$-*layer* of the NI-tree (Step 6-13, Algo 4.1). If a node is marked with an item in the itemset is, it will be attached to the new root node. Otherwise, we will skip this node and its child nodes will be recursively checked (Step 11, Algo 4.1).

Algorithm 4.1: NI-treeSubtraction

Input: Root Node r, Items to be removed R

Result: New Root Node r'

```
1  r' ←new NI-treeNode({r.is \ R}, r.c, ∅)
2  TraverseSubtree(r, r', r'.is)
3  return r'
4
5  Procedure TraverseSubtree(Node n, Node p, Itemset is)
6      foreach Child n' ∈ n.l do
7          if n'.i ∈ p.is then
8              Add n' to p.l
9          else
10             p.c ← p.c + n'.c
11             TraverseSubtree(n', p, is)
12         end
13     end
14 end
```

We add the count value of removed nodes to the new root node (Step 10, Algo 4.1). When the removing process is finished, the count in the new root node of the de-tree is the support of the corresponding pattern. It is worth to note that the subtraction process is terminated when all *1st-layer* nodes in the original NI-tree are checked. Thus, nodes below the *1st-layer* of

the de-tree are not checked yet and may still contain items not covered by the itemset in the root node. Such scheme avoids the scanning of the whole NI-tree. Those nodes can be removed later and the count value is still correct as proved later.

Three examples are illustrated in Figure 4.2 by excluding $\neg C$, $\neg C \neg D$ and $\neg C \neg D \neg B$ from the NI-tree in Figure 6.2 respectively. In Figure 4.2a, the node of $\neg C$ is removed and all its child nodes are attached to the new root node. The support of the pattern $\{ABDE\}$ is 0 after the subtraction since the count of the removed node is 0. The NI-tree in Figure 4.2b is achieved by further subtracting node of $\neg D$ from the previous tree. The count of node $\neg D$ is added to the new root. Thus, the support of the pattern $\{ABE\}$ is 1. However, if we further exclude item $\neg B$ from the above NI-tree, the node of $\neg B$ is kept since the node of $\neg A$ above it is on the *1st-layer*. Thus, nothing is removed and the support of pattern AE is still 1 as shown in Figure 4.2c.

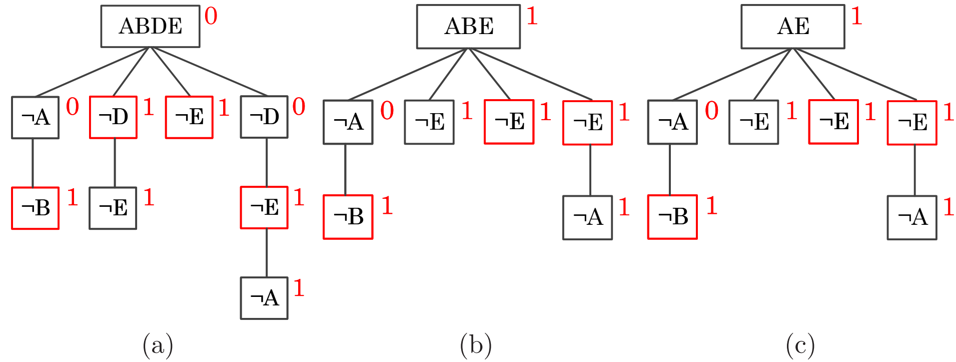


Figure 4.2: Examples de-trees by excluding $\neg C$, $\neg C \neg D$ and $\neg C \neg D \neg B$ from the NI-tree in Figure 6.2. The node of $\neg B$ is not removed in tree (c) since the subtraction process is terminated when all *1st-layer* nodes are covered by the itemset in root.

Theorem 4.1.1

The items removing process on the NI-tree described above generates the support of patterns correctly.

Proof. According to the construction process of the initial NI-tree, only the count of the last node is increased by 1 for each transaction. Thus, the total

count of all nodes in a NI-tree is $|\overline{\mathcal{T}}| = |\mathcal{T}|$.

Assume the itemset in the new root node of the de-tree is X. Given a transaction T in $|\mathcal{T}|$, if $\overline{T} \cap \neg X \neq \emptyset$, then at least the last node of $\overline{T}$ must be in the de-tree since otherwise, all nodes on the path corresponding to $\overline{T}$ will be removed.

Thus, the number of such transaction T is just the total count remained in the de-tree. By definition, it equals to $\neg X.jsupp$. As all other counts are added to the new root node, its count value is $|\mathcal{T}| - \neg X.jsupp = X.isupp$. $\square$

Furthermore, the items removing and support counting process can be applied recursively. Given two set items to be removed: R_1 and R_2. Obviously, first removing R_1 then R_2 is equivalent to first removing R_2 then R_1. Both removing order will finally generate the same de-tree as removing the joint set of $R_1 \cup R_2$. Thus, we can enumerate all patterns in the dataset without starting from the initial NI-tree.

The removing process is simple and efficient. Considering one removing process which generate a new de-tree with k items in the new root node of the de-tree. The time to determine if an item is in the new root is $O(\log k)$. Moreover, let M be the number of nodes in the original NI-tree labeled with items that not in the new root node. Then, in the worst case, all M nodes are removed during the process. Thus, the overall complexity of the process is $O(M \log k)$. The value of k, which is the length of a pattern, is relatively small compared the size of dataset. Therefore, $\log k$ can be treated as a constant. The number of nodes M is linear to the size of the dataset $|\mathcal{T}|$. Thus, the complexity of our support counting process is also linear to the size of the dataset, which is the same as other efficient bottom-up pattern mining algorithms [97].

4.2 Infrequent Pattern Mining with Termination Nodes Pruning

The initial NI-tree contains the full itemset $\mathcal{I}$ in the root node, which means that the NI-tree represents the support of the pattern $\mathcal{I}$. One item is excluded

in each recursive step and a new de-tree will be generated with a shorter itemset in the root node. All combinations of items in $\mathcal{I}$ will be enumerated recursively. This is a typical divide-and-conquer paradigm, which is employed by many other pattern mining algorithms as well. The difference is that we remove items in the NI-tree rather than project on items in the tree.

Algorithm 4.2: NIIMiner

Input: Transaction Database $\mathcal{T}$, Minimum Support $minSup$
Result: Infrequent Itemset List $\mathcal{IL}$

```
1  r, L_t ←BuildNI-tree(T) ;   // root node r, termination node
    list L_t
2  IL, FL ← ∅ ;        // Infrequent and frequent pattern list
   /* Start with excluding items towards termination nodes
      */
3  foreach Termination Node N_t ∈ L_t do
4      l_t ← {Items on path from r to N_t}
5      is' ← r.is \ l_t
6      if is' ∉ IL ∧ is' ∉ FL then
7          r' ←NI-treeSubtraction(r, l_t)
8          if r'.c < minSup then
9              IL ← IL ∪ {is'}
10             RecursiveRemove(r',minSup,IL,FL,null) ;
               // null≺ i ∈ I
11         else
12             FL ← FL ∪ {is'}
13         end
14     end
15 end
16 return IL
```

More specifically, items are excluded in ascending order with respect to their frequency in the original dataset. Let the operator $\prec$ denotes "less frequent", $A \prec B \prec C$, we exclude items using the following divide-and-conquer paradigm: 1. excluding A and all its combinations, 2. excluding all combinations of B but without A, 3. excluding all combinations of C but without A and B. The support value will be computed for each excluded itemset as described above. Infrequent patterns found so far are stored in the infrequent

Algorithm 4.3: RecursiveRemove

```
   /* Typical divide-and-conquer step                                */
1  Procedure RecursiveRemove(Node r, minSup, IL, FL, Last item
    iX)
2     foreach i ∈ r.is, iX ≺ i do
3        is' ← r.is \ {i}
4        if is' ∉ IL ∧ is' ∉ FL then
5           r' ←NI-treeSubtraction(r,{i})
6           if r'.c < minSup then
7              IL ← IL ∪ {is'}
8              RecursiveExtend(r',ϵ,IL,FL,i)
9           else
10             FL ← FL ∪ {is'}
11          end
12       end
13    end
14 end
```

pattern list $\mathcal{IL}$. The recursive process is terminated if the current pattern is frequent. Such divide-and-conquer paradigm is known as the depth-first traversing for itemset mining [2]. The procedure `RecursiveRemove` in Algorithm 4.2 illustrates such process.

It is obvious that there are a huge number of nonexistent patterns in a real dataset. Intuitively, they should be skipped. However, the simple divide-and-conquer procedure described above starts from the full set and excluding items one by one. All nonexistent patterns have to be traversed before considering existent infrequent patterns, which might cost even more time than bottom-up traversing. For example, the NI-tree in Figure 4.2a, which only excludes C, should be skipped since its corresponding pattern $\{A, B, D, E\}$ does not exist in the dataset.

To address this problem, patterns stored in termination nodes mentioned before are used as the starting point, rather than the full pattern $\mathcal{I}$. Algorithm 4.2, step 1-16, illustrates the overall NIIMiner procedure in detail. For each termination node, items in its corresponding neg-rep itemset, which is formed by all items on the path up to the root node, will be removed at

once in the first recursive step, which guarantees that the generated de-tree represents an existing pattern in the dataset (Step 4-6, Algo 4.2).

The divide-and-conquer paradigm is then applied similarly on each de-tree for the rest of items after removing termination nodes. But, introducing termination nodes will lead to duplicates in the recursive traversing process. For example, given termination nodes in the initial NI-tree in Figure 6.2, excluding CD and B is equivalent to excluding BD and C. If a duplicate arises, the recursion should be terminated sine removing more items will also lead to duplicates.

An extra pruning step is necessary to check if the current pattern has been accessed before (Step 5 and 20, Algo 4.2). Infrequent and frequent patterns found so far are stored in two hash sets. Before generating a new de-tree, its corresponding pattern is tested. If it already exists in one of the hash sets, itself and all its subsets must have been accessed already. Thus, further divide-and-conquer process on this pattern can be terminated.

4.3 Infrequent Pattern Mining with 1st-layer Nodes Pruning

In experiments on real-world datasets, we noticed that the number of duplicates is enormous. Thus, it is worth to rethink about the non-existent pattern skipping schema. First of all, the structure of NI-tree can address the nonexistent pattern problem implicitly, without suffering from the expensive duplicates checking schema. For example, in the NI-tree of Figure 6.2, we have $C \prec D \prec E \prec A \prec B$. Only $\neg C$ and $\neg D$ are on the 1st-layer. Removing items that are not on the 1st-layer does not lead to any existent pattern. For instance, according to the divide-and-conquer paradigm, if we first remove E, then in the following recursive step, we remove EA, EB and EAB. All of them correspond to nonexistent patterns. However, if we remove C, though the pattern $\{ABDE\}$ is nonexistent, removing C will still lead to a valid pattern later when further items after C are removed under the divide-and-conquer paradigm.

Thus, when the current pattern is nonexistent, we should only remove items in a 1st-layer node. We can guarantee that each removing action will eventually lead to at least one valid infrequent pattern. Our NIIMiner is modified respectively. The first part (step 3-16, Algo. 4.2) of removing termination nodes is skipped. We perform the divide-and-conquer paradigm directly on the initial NI-tree. When the pattern in the current root is nonexistent, we only remove items on its 1st-layer, i.e., changing step 18, Algo. 4.2 to "**foreach** $i \in \{n.i | n \in r.l\}, iX \prec i$ **do**". The duplicate checking step (step 20, Algo. 4.2) is then not necessary since the divide-and-conquer paradigm for pattern mining guarantees that no duplicate happen.

Moreover, only removing items in the 1st-layer is also more efficient since the child list of the root is always short when compared with $\mathcal{I}$. The reason is simple. Assume that the order of items in the given dataset $\mathcal{T}$ is $A \prec B \prec C \prec D \prec \dots$, i.e., A is the rarest item in $\mathcal{T}$. There will be a node $\neg A$ on the 1st-layer of the initial NI-tree. A node labeled with $\neg B$ should also be there since otherwise, A is an item that does not exist in any transactions, which is contradict to the fact that A is in $\mathcal{T}$. Thus, the root node in our initial NI-tree must contain two children: A and B.

The story is different for item C. If $\neg C$ is on the 1st-layer, then there must be at least one transaction $T \in \mathcal{T}$ such that $\{\neg A, \neg B\} \notin \overline{T} \Rightarrow \{A, B\} \in T$. Let $P(i)$ be the probability that the item i is in a randomly selected transaction of $\mathcal{T}$. Then, the probability that $\{A, B\} \in T$ is $P(A)P(B)$, assuming items are independently distributed. Since A and B are the two rarest items in $\mathcal{T}$, we know that $P(A)P(B)$ is minimal. Therefore, it is unlikely that $\neg C$ is on the 1st-layer. In fact, we have $P(i \in r.l) = \prod_{j \prec i} P(j)$, which means that all frequent items are unlikely to appear on the 1st-layer. In consequence, the modified mining process is more efficient.

4.4 Experimental Evaluation

Four real datasets obtained from the frequent itemset mining dataset repository (`http://fimi.ua.ac.be/data/`) are investigated. Figure 4.3 lists main features of these four datasets. It is worth to note that they have a larger

average transaction length when compared with a typical business/supermarket dataset. LCMfreq [97] is used as the baseline. LCMfreq is one of the most efficient frequent itemset mining algorithms which represents the performance of bottom-up and depth-first lattice traversing approach. Our NIIMiner is implemented in JAVA while LCMfreq is obtained from the SPMF library [37].

An early approach proposed in [71], which starts from the full itemset and tests all nonexistent patterns, is not included since it can not finish on those real datasets. Existing top-down breadth-first algorithms are also not included in our experiments since they can not finish the mining task too, due to the expensive candidate generation step. In fact, they are much slower than bottom-up depth-first based approaches, such as FPGrowth, as shown in [96]. We also test the first top-down depth-first pattern mining algorithm mentioned in [107]. This algorithm treats items as transaction ids and transaction ids as items and identifies patterns with limited length using typical bottom-up traversing method. However, such algorithm is also extremely slow since it has to mine patterns from a dataset with very long transactions. In our early experiments, this first top-down depth-first algorithm can only handle datasets with hundreds of transactions. As a consequence, our experiments in this section only compare three algorithms: LCMfreq, which represents bottom-up depth-first approach, and our NIIMiner with different nonexistent pattern skipping methods (Tnode: Termination Nodes, 1node: 1st-layer nodes).

Dataset	$\|\mathcal{T}\|$	$\|X\|$	$\|I\|$	$\|L\|$	$sparsity$
Chess	3k	118k	75	37	0.49
Mushrooms	8k	193k	119	23	0.19
Connect	67k	2904k	129	43	0.33
Accidents	340k	11500k	468	33.8	0.07

Figure 4.3: Statistics on real datasets. $|\mathcal{T}|$: transaction number, $|X|$: number of items, $|I|$: number of distinct items, $|L|$: average itemset size.

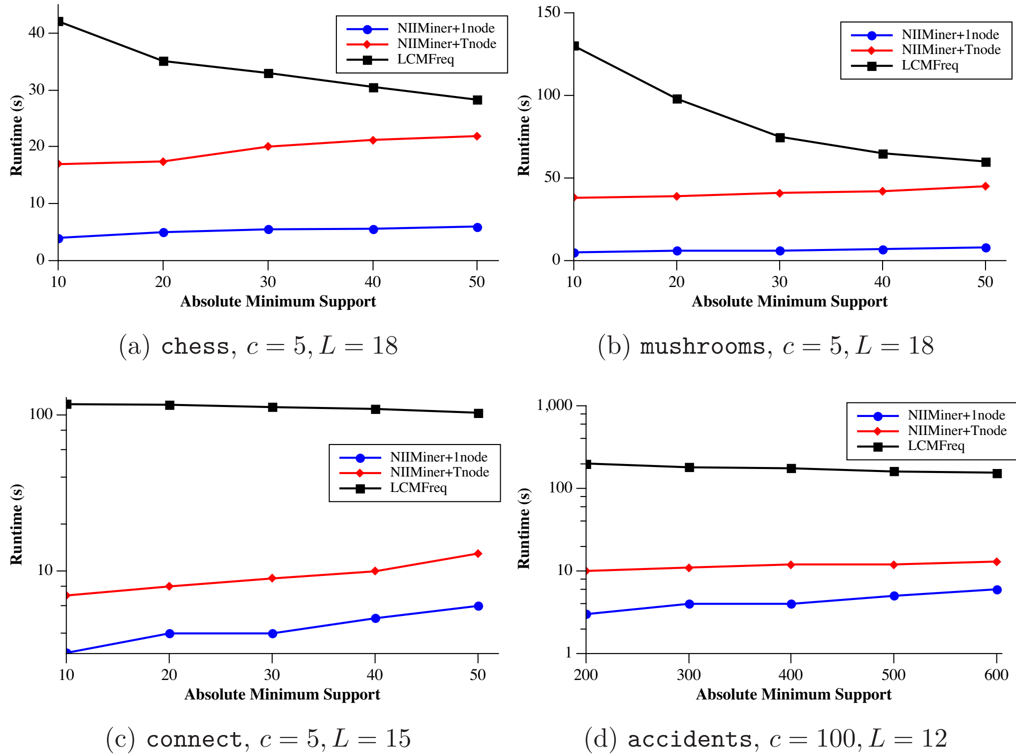
We first investigate the runtime performance with respect to different minimum support values for each dataset. Note that the main goal of this work is to extract low support patterns. Given a small minimum support

value $minSup$, the NIIMiner will access all patterns with support smaller than $minSup$. In contrast, the bottom-up based LCMfreq algorithm needs to traverse all patterns with support larger than 1 in $\mathcal{T}$ to generate the same set of low support patterns. It is too slow to make a reasonable performance comparison. A small value c is introduced so that the LCMfreq algorithm only traverse patterns with support larger than $minSup - c$, rather than 1. Thus, our experiments can also be interpreted as comparing the runtime performance of accessing patterns in the support range of $[minSup - c, minSup)$. The value c is fixed for each dataset. Furthermore, the maximum transaction length L is restricted since otherwise, no algorithm can finish the low support mining task.

As shown in Figure 4.4, our NIIMiner is more efficient than the LCMfreq approach since it has to access a huge number of high support patterns. When the minimum support increasing, the NIIMiner needs to traverse more patterns while the LCMfreq algorithm traverses less. Thus, a bottom-up based approach might be more efficient if desired supports are not that small. However, since we focus on the low support scenario, our NIIMiner will be a better choice. NIIMiner with 1st-layer nodes pruning is even faster since it avoids nonexistent patterns as well as expensive duplicate checking step.

We also studied the runtime performance with respect to different dataset size (total number of items $|X|$). The dataset size is adjusted by limiting the maximum transaction length L. Our top-down based NIIMiner behaviors similar to the bottom-up based approach. Obviously, runtime is positively correlated to the dataset size. However, the runtime of our NIIMiner increased much slower than its competitor. This is because the NI-tree employed in our approach compresses the dataset as well as provides efficient counting ability. Moreover, bottom-up traversing strategy accessed more frequent patterns when the dataset size increased.

It worth to note that our negative itemset tree is actually the same as the FP-tree used by the FPGrowth algorithm [51]. Thus, the space complexity of our approach is the same as the FPGrowth algorithm, which is known to be efficient. More specifically, our consumption is a constant multiple of the space used by FPGrowth while the constant value is determined by how the



(a) chess, $c = 5, L = 18$

(b) mushrooms, $c = 5, L = 18$

(c) connect, $c = 5, L = 15$

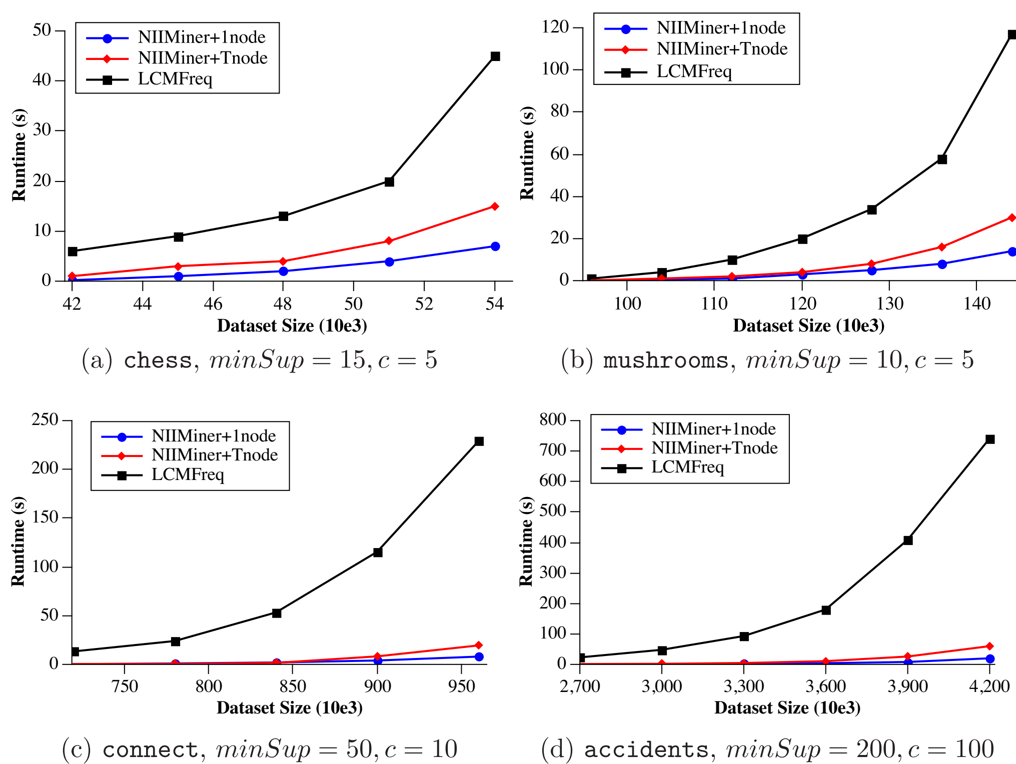
(d) accidents, $c = 100, L = 12$

Figure 4.4: Runtime experiments on different absolute minimum support values.

negative dataset is larger than the original one.

4.5 Conclusion and Future Works

Our novel rare itemset miner NIIMiner has proven to solve the problem of rare itemset mining in an efficient and successful manner. By utilizing the negative representations of rare itemsets, we addressed this task from its dual perspective. Two different nonexistent pattern pruning methods is proposed and the 1st-layer nodes pruning method is more efficient. The major limitation of our NIIMiner appears on extreme sparse datasets, such as a typical supermarket dataset, since the corresponding neg-rep dataset can be thousand times larger than the original one with very long neg-rep transactions. An integration of both bottom-up and top-down traversing strategies should be investigated to overcome this problem. Furthermore,



(a) chess, $minSup = 15, c = 5$

(b) mushrooms, $minSup = 10, c = 5$

(c) connect, $minSup = 50, c = 10$

(d) accidents, $minSup = 200, c = 100$

Figure 4.5: Runtime experiments on different dataset size.

condensed representations for rare patterns, such as closed pattern or non-derivable patterns, and their corresponding algorithms could be used for real applications and should be conducted as future works.

Chapter 5

Bi-directional Rare Closed Itemset Mining

Previous chapters introduced novel rare itemset mining algorithms using negative itemset tree. Approaches proposed uses top-down depth-first traversing strategy. However, experiments show that when the dataset is sparse, conventional bottom-up based approaches can be more efficient in mining rare patterns. The reason is that patterns in sparse datasets are short. Bottom-up based algorithms only need to access a few frequent patterns. In this case, the overhead of top-down traversing becomes significant.

In this chapter, a bi-directional itemset mining framework is proposed, which combines both bottom-up and top-down traversing strategy. The motivation is simple: even in dense datasets, there are sparse parts. Combining both bottom-up and top-down traversing strategy is a good way to achieve better performance.

Furthermore, instead of extracting the full list of rare itemset, this chapter aims at finding closed rare patterns. Closed itemset is a lossless condensed representation for patterns. In real applications, the full list of patterns is usually too large to process. Mining closed itemsets can significantly reduce the size of redundant information.

Parts of the material presented in this chapter have been published in [74].

5.1 Closed Infrequent Itemset

For the ease of reading, the basic definitions and notations for itemset mining is introduced here again. Consider $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ to be the set of distinct item. Any non-empty subset $X \subseteq \mathcal{I}$ is an *itemset*. Any itemset X with length $|X| = l$ is referred to as a *l-itemset*. Itemset $Y \subseteq X$ is a *subset* of X while X is a *superset* of Y. A tuple $T = (tid, X)$ is called a *transaction*, where *tid* is the transaction identifier. For simplicity, a transaction $T = (tid, X)$ also refers to its itemset if not specified. Any non-empty itemset $Y \subseteq X$ is *contained* by a transaction $T = (tid, X)$ and we just write $Y \subseteq T$. A set of transactions establish a *transaction database* $\mathcal{T}$. Given an itemset X, a *projected database* with respect to X is the set of transactions in $\mathcal{T}$ that contain X, denoted as $\mathcal{T}_X = \{T \in \mathcal{T} | X \subseteq T\}$. A dataset is sparse if most of its transactions only contains few items in $\mathcal{I}$, i.e., the value of $\sum_{T \in \mathcal{T}} |T|/(|\mathcal{T}| \cdot |\mathcal{I}|)$ is small. Table 5.1 illustrates an example transaction database.

Tid	Itemset
1	{a,b,c}
2	{a,b,d}
3	{b,c}
4	{a,b}
5	{a,b,e}
6	{d,e}

Table 5.1: Example dataset

Given a transaction database $\mathcal{T}$, the (absolute) *support* of an itemset X is defined as the number of transactions $T \in \mathcal{T}$ containing X, i.e.: $X.supp = |\{T \in \mathcal{T} | X \subseteq T\}|$. An itemset (pattern) X is *infrequent* if and only if: $0 < X.supp < minSup$, where *minSup* is a user-defined minimum support threshold. Otherwise, it is *frequent* ($X.supp \leq minSup$) or *nonexistent* ($X.supp = 0$). The main goal of this work is to extract closed infrequent itemsets from a large transaction database efficiently.

The concept of closed frequent itemset proposed in [109] is a lossless condensed representation of frequent patterns. Given the set of closed frequent itemsets, we can determine whether an itemset is frequent or not. If it is

frequent, we can further determine the value of its support. We adapt this concept as a lossless condensed representation for infrequent patterns.

Definition 5.1.1
An itemset X is closed if and only if $\nexists Y \supseteq X$ such that $Y.supp = X.supp$. In addition, if $0 < X.supp < minSup$, X is called a closed infrequent itemset.

The set of closed infrequent itemsets is denoted as $\mathcal{CI}$. To guarantee the lossless property of closed infrequent itemset, we also introduce the frequent border set $\mathcal{FB}$.

Definition 5.1.2
The frequent border $\mathcal{FB}$ is a set of closed frequent itemset. An itemset $X \in \mathcal{FB}$ if and only if $X.supp \geq minSup$ and $\nexists Y \supset X, Y.supp \geq minSup$.

In fact, the definition of $\mathcal{FB}$ is equivalent to the concept of maximal frequent itemset [16]. If an itemset X is frequent, then it must be covered by an itemset in $\mathcal{FB}$, otherwise, itself is in $\mathcal{FB}$ by definition. Furthermore, it is obvious that itemsets in $\mathcal{FB}$ are also closed.

Theorem 5.1.1
Given the set of closed infrequent itemset $\mathcal{CI}$ and the corresponding frequent border set $\mathcal{FB}$, we can determine if an itemset X is infrequent or not, and if it is, we can also determine the support $X.supp$.

Proof. If X is frequent, then it must be in $\mathcal{FB}$ or covered by an itemset in $\mathcal{FB}$, due to the definition of maximal frequent itemset [16]. If X is infrequent, then:

- if $X \in \mathcal{CI}$, we have its support.
- if $X \notin \mathcal{CI}$, then $X.supp = Y.supp$, where $Y = \underset{Y \in \mathcal{CI}, X \subset Y}{\operatorname{argmin}} |Y|$.

All infrequent itemset that exists in the dataset must be covered by $\mathcal{CI}$. This can be proofed by contradiction: if an infrequent itemset that exists in the dataset is not covered by any closed itemset in $\mathcal{CI}$, then by the definition of the closed itemset, itself is a closed itemset and must be in $\mathcal{CI}$. □

In summary, the set of closed infrequent itemset $\mathcal{CI}$ with frequent border set $\mathcal{FB}$ form a concise condensed representation of infrequent patterns.

5.2 Bi-directional Infrequent Itemset Mining

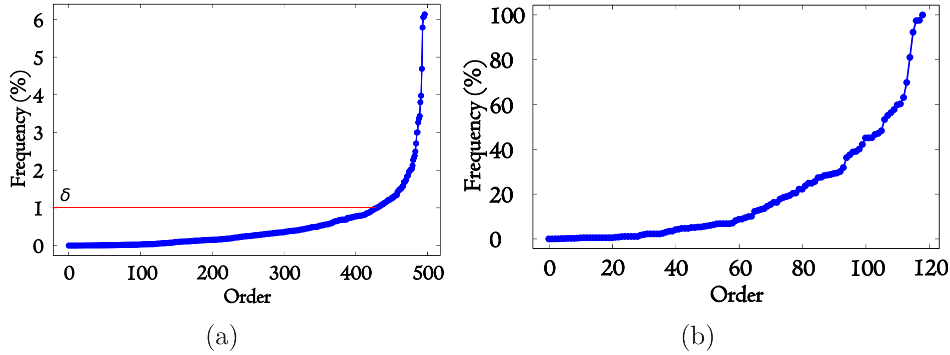
Let t be the average traversing time cost for each accessed pattern and S be the total number of accessed patterns, the overall traversing runtime can be roughly considered as $t \times S$.

Considering the two performance factors for itemset mining t and S mentioned above, one might conclude that an efficient infrequent itemset mining algorithm should traverse 1. depth-first (small t) and 2. top-down (small S) with 3. condensed representation (small S) applied. However, experiments in [96] show that existing breadth-first top-down based infrequent itemset mining algorithms such as Rarity [95] are actually much slower than extracting all patterns using bottom-up depth-first based frequent itemset mining algorithms. Furthermore, a top-down based algorithm with depth-first traversing applied is only efficient on very dense datasets [71].

Thus, we assume that the value of S is reduced due to the top-down traversing but the value of t becomes much larger. As a consequence, top-down traversing is meaningful only if there are much more frequent patterns than infrequent patterns, so that the $t \times S$ value can be smaller on top-down based approach. Such case might happen in a very dense dataset while most real-world datasets are very sparse, i.e. traversing the frequent part only wastes an insignificant amount of time.

However, we notice that in most real datasets, items are not uniformly distributed. There exist both sparse and dense sub-parts in one dataset. Figure 5.1 illustrates the frequency of items in BMS1 and mushrooms dataset (`http://fimi.ua.ac.be/data/`) in ascending order. A few items are very frequent while most of the others are less frequent.

If we assume that items are distributed independently, the (relative) support of an itemset X can be estimated as the product of all its items' probability: $X.supp = \prod_{i \in X} P(i)$, where $0 \leq P(i) \leq 1$ is the frequency of item i. Thus, if an itemset X contains a less frequent item, this itemset tends to be infrequent even though it is short. Therefore, the projected database with respect to a less frequent item i only contains few frequent patterns, which is suitable for bottom-up traversing. Similarly, the sub-database that only



(a) (b)

Figure 5.1: Ascending ordered frequency plot of items in BMS1 and mushrooms dataset.

contains more frequent items tends to contain more frequent patterns. Thus, top-down traversing should be a better choice for infrequent itemset mining task on such sub-database.

In fact, if we sort items based on their frequency in ascending order, a bottom-up depth-first based algorithm spent most of its runtime on processing sub-datasets projected on more frequent items to extract infrequent patterns. Figure 5.2 illustrates the cumulative runtime of the LCM algorithm spent on each projected dataset of the BMS1 dataset. Traversing projected datasets of 150 least frequent items only costs 1 second. However, projected datasets of 150 most frequent items consume more than 50 seconds. The latter group of projected datasets should be suitable for top-down traversing under the assumption we made above. Thus, combining bottom-up and top-down traversing for infrequent itemset mining has a good opportunity to improve the performance.

Based on the idea of divide-and-conquer, we could split a single dataset into two parts: 1. contains *less frequent* items and 2. does not contain less frequent items. A bi-directional itemset mining framework can be applied to extract closed infrequent patterns as the following:

1. Traverse itemsets that contain less frequent items in a bottom-up direction.

2. Traverse itemsets that only contain more frequent items in a top-down

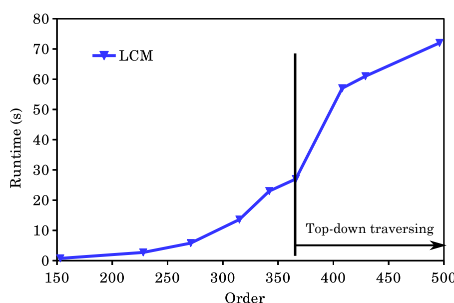


Figure 5.2: Cumulative traversing time of LCM [97] on each projected databases of the BMS1 dataset.

direction.

3. Combine results from above.

In other words, we first extract closed infrequent patterns from projected datasets with respect to less frequent items. Then, we remove all less frequent items and perform top-down traversing to extract all closed infrequent patterns that only contain more frequent items. Finally, we join results from the two steps above and get the complete set of patterns.

A threshold δ is necessary to determine if an item is less frequent or not. The optimized value of δ might depend on many factors such as the relative efficiency between bottom-up and top-down traversing, how items distributed across the dataset, which is very difficult to determine. In practice, the corresponding frequency value around the inflection point on the frequency-order plot, as shown in Figure 5.2 (a), is chosen as the value of δ, which performs pretty good in our experiments. Obviously, the value of δ should be larger than $minSup$.

Based on the bi-directional traversing framework described above, we propose a *Closed Infrequent Itemset Mining* algorithm, Rare Closed Itemset Miner (RaCloMiner).

Bottom-up Closed Infrequent Itemset Mining

The LCM [97] algorithm is adapted to generate infrequent closed itemsets which contain at least one item with support smaller than δ. LCM is one

of the most efficient closed frequent itemset mining algorithms based on database projection (bottom-up depth-first traversing) with efficient closure checking function. Algorithm 5.1 illustrates the pseudo code of the bottom-up traversing step.

```
Algorithm 5.1: BottomUpTraverse
   Input: Transaction Database 𝒯, Minimum Support minSup,
          Threshold δ
   Result: Closed Infrequent Patterns 𝒞ℐ, Frequent Border Set ℱℬ
1  foreach Transaction T ∈ 𝒯 do
2      foreach Item i ∈ T do
3          if i.supp < δ then
4              𝒯_i ← 𝒯_i' ∪ T
5          end
6      end
7  end
8  foreach Projected database 𝒯_i do
9      CloseCheck(𝒯_i)
10     if 𝒯_i is closed then
11         LCM(i, {i}, 𝒯_i)
12     end
13 end
```

In the original LCM algorithm, items are sorted in descending order according to their support. Here we sort items in ascending order since less frequent items must be included in the pattern. Operator $\prec$ is used to express the order of items.

In the first projection iteration, the original transaction database is projected only onto items with support smaller than the given threshold δ. Each projected database is then traversed by the adapted LCM algorithm. All closed patterns in projected databases are enumerated. Infrequent closed itemsets will be added to IC. A frequent closed itemset will be added to $\mathcal{FB}$ if all its supersets are infrequent (step 34). Details of the LCM algorithms, such as the `CloseCheck()` function, can be found in the original LCM paper [97].

Other bottom-up based closed frequent itemset mining algorithms could

Algorithm 5.2: LCM

```
1  Function LCM(Current Item iX, Current Closed Itemset iS,
     Projected Database T'_iS)
2      ∀i, iX ≺ i, T'_i ← ∅
3      foreach Transaction T ∈ T'_iS do
4          foreach Item i ∈ T, iX ≺ i, i ∉ iS do
5              T'_i ← T'_i ∪ T
6          end
7      end
8      AllInfreq←true
9      foreach Projected database T'_i do
10         CloseCheck(T'_i)
11         if T'_i is closed then
12             r←LCM(i, iS ∪ {i}, T'_i)
13             AllInfreq←AllInfreq∧r
14         end
15     end
16     if |T'_iS| < minSup then
17         CI ← CI ∪ iS
18         return true
19     else
20         if AllInfreq = true then
21             FB ← FB ∪ iS
22         end
23         return false
24     end
25 end
```

also be adapted and employed in this bottom-up traversing step. The performance of this bottom-up traversing step is at least as the same as the employed frequent itemset mining algorithm, which should be very efficient as discussed above.

Top-down Closed Infrequent Itemset Mining

To the best of our knowledge, there is no closed infrequent itemset mining algorithm in the literature. Thus, we propose a simple algorithm which

traverses the lattice top-down by adapting the idea of database projection. At the beginning of the top-down traversing step, all items with support smaller than the given threshold δ are removed. This guarantees that all patterns found in this step do not contain less frequent items.

In bottom-up based traversing, such as the LCM algorithm, the database is projected with respect to single items. Contrarily, in the top-down based traversing, a database is projected onto itemsets called *projection itemsets* (p-itemsets). Each top-down projected database is aligned with one p-itemset which covers all transactions in the database. An itemset X in the database $\mathcal{T}$ is a projection itemset if and only if it is not covered by others, i.e., $\nexists Y \in \mathcal{T}, X \subset Y$. For example, the set of p-itemsets in the database in Table 5.1 are: $\{a, b, c\}$, $\{a, b, d\}$, $\{a, b, e\}$ and $\{d, e\}$.

p-itemset	count	T-List	R-List
$\{a, b, c\}$	1	$\{a, b\}$, $\{b, c\}$, $\{a, b\}$, $\{a, b\}$	$\emptyset$
$\{a, b, d\}$	1	$\{d\}$	$\{a, b\}$
$\{a, b, e\}$	1	$\{e\}$	$\{a, b\}$
$\{d, e\}$	1	$\emptyset$	$\{d\}$, $\{e\}$

(a)

p-itemset	count	T-List	R-List
$\{a, b\}$	4	$\{b\}$	$\emptyset$
$\{b, c\}$	2	$\emptyset$	$\{b\}$

(b)

Table 5.2: Projected databases extracted (a) from the database in Table 5.1 and (b) from the projected database of $\{a, b, c\}$ (the first row in (a)). Each row is a projected database.

The pseudo code of the top-down traversing and its subroutines are illustrated in Algorithm 5.3 and 5.4. All transactions are sorted based on their length in descending order so that an itemset will not be covered by another itemset after it. Each projected database $\mathcal{T}'$ contains three components: the count of its p-itemset $\mathcal{T}'.count$, a list of projected transactions $\mathcal{T}'.T$ and a list of restriction itemsets $\mathcal{T}'.R$. If a transaction is equal to the p-itemset of $\mathcal{T}'$, the value of $\mathcal{T}'.count$ will be increased (Step 8, Algorithm

5.4). If an itemset is a subset of or partially covered by the p-itemset $\mathcal{T}'$, the common part is added to the transaction list $\mathcal{T}'.T$ (Step 12, 16, Algorithm 5.4). The restriction list of a projected database $\mathcal{T}'.R$ is created by intersects its p-itemset with all p-itemsets of previously projected databases (Step 8, Algorithm 5.4). The restriction list tells if a pattern has been processed in previously projected databases and should be skipped. The count of the projection itemset $\mathcal{T}'.count$ and the restriction list $\mathcal{T}'.R$ are inherited and aggregated from previous recursion step. This recursive projection process is terminated until the count of the projection itemset is larger or equal to the minimum support. Projection itemsets with different support values are added to $\mathcal{CI}$ and $\mathcal{FB}$ respectively. Figure 5.2 (a) and (b) illustrates an example of the first 2 recursion steps in the top-down projection process. Table (a) is the set of projected databases extracted from the original database in Figure 5.1 while table (b) lists projected databases w.r.t. the first database in the table (a).

Algorithm 5.3: TopdownTrverse

Input: Transaction Database $\mathcal{T}$, Minimum Support $minSup$,
Threshold δ

Result: Closed Infrequent Pattern $\mathcal{CI}$, Frequent Border Set $\mathcal{FB}$

1 **foreach** *Item* $i \in \mathcal{T}$ **do**
2   **if** $i.supp < \delta$ **then**
3   `Remove` i `from` $\mathcal{T}$
4   **end**
5 **end**
6 $\mathcal{T}.count \leftarrow 0$
7 $\mathcal{T}.R \leftarrow \emptyset$
8 `TopdownProject`($\mathcal{T}$, $minSup$)
9 Remove all $X \in \mathcal{FB}$ if $\exists Y \in \mathcal{FB}, X \subset Y$

Theorem 5.2.1
Our top-down projection algorithm is correct and complete as all patterns returned are projection itemsets.

Proof. The completeness property is obvious since the recursive process re-

Algorithm 5.4: TopdownProject

```
1  Function TopdownProject(Projected database 𝒯′,
     Minimum support minSup)
2      Sort(𝒯′)
3      L ← ∅
4      foreach Transaction X ∈ 𝒯′.T do
5          R ← 𝒯′.R
6          covered←false
7          foreach Projected database 𝒯′_Y ∈ L do
8              if X = Y then
9                  𝒯′_Y.count ← 𝒯′_Y.count + 1, covered←true
10                 break
11             else if X ⊂ Y then
12                 add X to 𝒯′_Y.T, covered←true
13                 break
14             else if X ∩ Y ≠ ∅ then
15                 Z ← X ∩ Y
16                 if ¬Restricted(R, Z) then
17                     R ← R ∪ Z, add Z to 𝒯′_Y.T
18                 end
19             end
20         end
21         if covered = false then
22             𝒯′_X.T ← ∅
23             𝒯′_X.count ← 𝒯′.count + 1
24             𝒯′_X.R ← R
25         end
26     end
27     foreach Projected database 𝒯′_X ∈ L do
28         if 𝒯′_X.count < minSup then
29             𝒞ℐ ← 𝒞ℐ ∪ X
30             TopdownProject(𝒯′_X, minSup)
31         else
32             ℱℬ ← ℱℬ ∪ X
33         end
34     end
35 end
```

Algorithm 5.5: Ristricted

```
1 Function Ristricted(Restriction list R, Itemset Z)
2     foreach Itemset X ∈ R do
3         if Z ⊆ X then
4             return true
5         end
6     end
7     return false
8 end
```

turns all projection itemsets that are infrequent or its supersets are all infrequent.

To prove the correctness property, we need to show those projection itemsets and closed itemsets are equivalent. Given a database $\mathcal{T}$, a p-itemset $Y \in \mathcal{T}$ and its corresponding projected database $\mathcal{T}'_Y$. Let an itemset Z be a p-itemset in $\mathcal{T}'_Y.T$. By definition, Z is closed in $\mathcal{T}'_Y.T$ since no itemset in $\mathcal{T}'_Y.T$ covers Z.

According to the projection process (step 7-23, Algorithm 5.4), itemset Z is assigned to $\mathcal{T}'_Y.T$ when $Z \subset Y$ or $\exists X \in \mathcal{T}, X \supset Z, X \cap Y = Z$. In both cases, $Z \subset Y, Z.supp > Y.supp$. Assume that Z is not closed in $\mathcal{T}$, then there must be an itemset $Z' \in \mathcal{T}, Z' \supset Z, Z'.supp = Z.supp$.

- If $Z' \in \mathcal{T}'_Y.T$, then Z is not a p-itemset in $\mathcal{T}'_Y.T$.
- If $Z' \notin \mathcal{T}'_Y.T$, then $Z' \nsubseteq Y$. Given $Z \subset Z', Z \subset Y, Z' \nsubseteq Y$, we have $Z.supp > Z'.supp$.

The first situation violates the assumption that Z is a p-itemset in $\mathcal{T}'_Y.T$. The second situation violates the fact that $Z'.supp = Z.supp$. Thus, a p-itemset in any projected database is also a closed itemset in the original database.

Furthermore, if itemset Z is not a p-itemset in any projected database during the recursive projection process, then $\exists Z' \supset Z$ such that Z' always appears together with Z. Thus, if Z is not a p-itemset in any projected

database, it is not closed itemset in $\mathcal{T}$. In other words, a closed itemset in the original database must be a p-itemset in one of the projected databases.

In summary, projection itemsets and closed itemsets are equivalent and our top-down projection is correct. □ □

Results Joining

The last step of our RaCloMiner is to join the temporary results from both bottom-up and top-down traversing steps above.

Closed patterns from the bottom-up traversing step, denoted as $\mathcal{CI}_1$ and $\mathcal{FB}_1$, must contain items with support smaller than δ while patterns from the top-down step, denoted as $\mathcal{CI}_2$ and $\mathcal{FB}_2$, do not. Given an itemset $X \in \mathcal{CI}_1$, there might be an itemset $Y \in \mathcal{CI}_2$ such that $Y \subset X$. However, $\nexists Y \in \mathcal{CI}_2$ such that $Y \supset X$. The same relationship also applies to patterns in $\mathcal{FB}_1$ and $\mathcal{FB}_2$. The join process iterates each pattern from the first step and removes the corresponding itemset from the second step if it exists. Algorithm 5.6 illustrates the pseudo code of the joining step.

Algorithm 5.6: JoinResults

Input: Closed infrequent itemsets $\mathcal{CI}_1, \mathcal{CI}_2$, Frequent border set $\mathcal{FB}_1, \mathcal{FB}_2$

Result: Closed infrequent itemsets $\mathcal{CI}$, Frequent border set $\mathcal{FB}$

```
1  foreach Itemset X ∈ CI_1 do
2  |  X' ← {i ∈ X | i.supp ≥ δ}
3  |  if ∃Y ∈ CI_2 ∧ Y = X' ∧ Y.supp = X.supp then
4  |  |  Remove Y from CI_2
5  |  end
6  end
7  CI ← CI_1 ∪ CI_2
8  foreach Itemset X ∈ FB_2 do
9  |  X' ← {i ∈ X | i.supp ≥ δ}
10 |  if ∃Y ∈ FB_2 ∧ Y = X' then
11 |  |  Remove Y from FB_2
12 |  end
13 end
14 FB ← FB_1 ∪ FB_2
```

5.3 Experiments

To the best of our knowledge, there is no algorithm designed specifically for closed infrequent itemset mining. Thus, we select the LCM [97] and the Rarity [95] algorithms as the baseline but mainly focus on the time spent on lattice traversing by each algorithm under different settings. The LCM algorithm represents the most efficient bottom-up closed frequent itemset mining approach. In order to generate all closed infrequent patterns, the (absolute) minimum support value of LCM algorithm will be always set to 1. Extra post-processing steps, such as removing all frequent patterns, is skipped since we only focus on the lattice traversing performance. The Rarity algorithm is the most efficient approach that extracts infrequent itemsets using top-down traversing strategy. Similarly, the post-processing step to generate closed patterns is also skipped. The algorithm proposed in [71] is not used since it returns all nonexistent patterns and thus only works under a very dense case.

6 real-life datasets, obtained from the frequent itemset mining repository (`http://fimi.ua.ac.be/data/`), are used to conduct the performance of our algorithms. Table 6.11 lists main statistic information of datasets used in our experiments. The first three datasets are very sparse while the last three datasets are relatively dense.

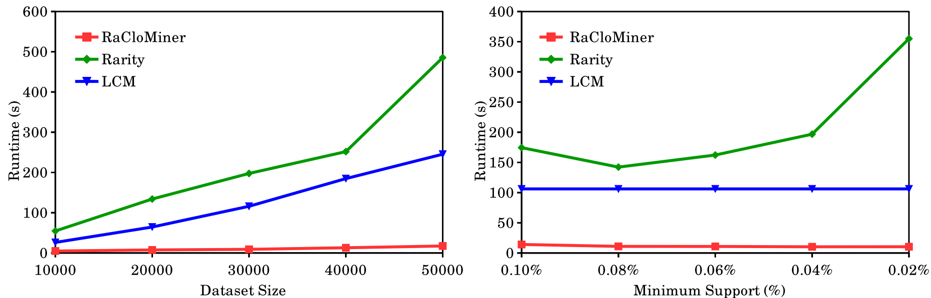
We test the performance of each dataset with different dataset size N and minimum support value $minSup$ settings. First N transactions will be used as the testing dataset. Bottom-up threshold δ is set around the inflection point of the histogram plot as described above for each dataset.

Database	Size (N)	Items ($\|I\|$)	Average length (L)
retail	88162	16470	10.3
BMS1	59602	497	2.5
BMS2	77512	3340	4.6
mushrooms	8415	119	23
chess	3196	75	37
connect	67556	129	43

Table 5.3: Statistics of real-life datasets used in our experiments.

We implement our RaCloMiner and the Rarity algorithm while the LCM algorithm is obtained from the SPMF tool [37]. All of them are implemented in Java. Experiments are executed on a machine with Intel 3.4GHz CPU and Ubuntu 16.04 installed.

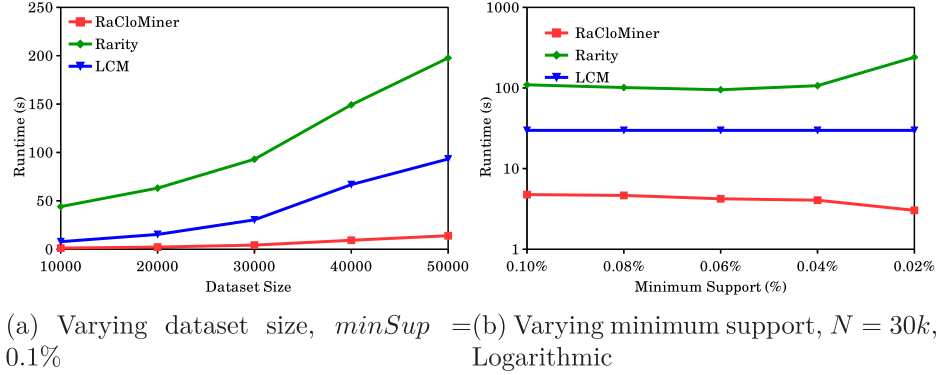
It is worth to note that some experiments will not finish if the maximum transaction length is too large in the dataset. Let L_R be the maximum length value for the Rarity algorithm while L_L is the value for LCM and RaCloMiner. If such value is given, it implies that the corresponding algorithm cannot finish the experiment under the same settings as others and transactions used will be cut down to the specified length. It is well known that the runtime increases monotonically with transactions length. As the Rarity algorithm is the slowest one even with a smaller maximum transaction length, the conclusion on performance comparison will not be affected by such inconsistency in experiments settings.



(a) Varying dataset size, $minSup = 0.05\%$ (b) Varying minimum support, $N = 30k$

Figure 5.3: Runtime performance on retail dataset, $\delta = 1\%, L_R = 10$

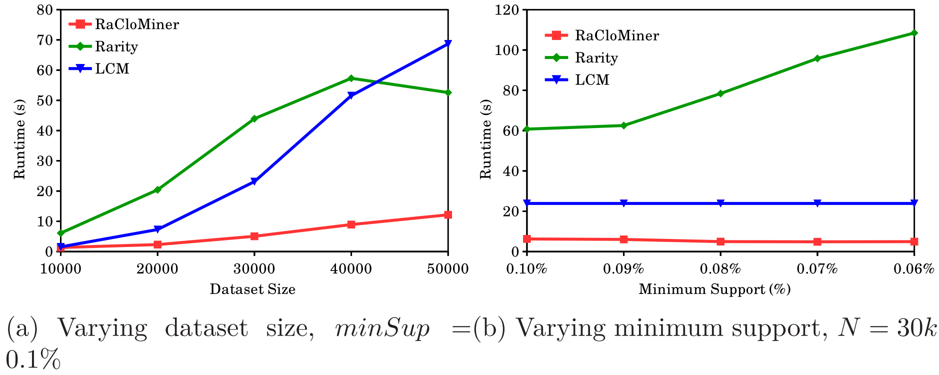
Figure 5.3 5.4 and 5.5 present results from the first three experiments conducted on retail, BMS1 and BMS2 datasets. These three datasets are very sparse since their average transaction length L is much smaller than the number of distinct items $|I|$, as shown in Table 6.11. Rarity cannot finish the task in all cases. A few long transactions in these datasets make the pruning step in Rarity extremely slow so that the maximum transactions length L_R has to be set. The runtime results of Rarity is abnormal in some cases. For example, in Figure 5.5 (a), the runtime or Rarity decreased when



(a) Varying dataset size, $minSup = 0.1\%$

(b) Varying minimum support, $N = 30k$, Logarithmic

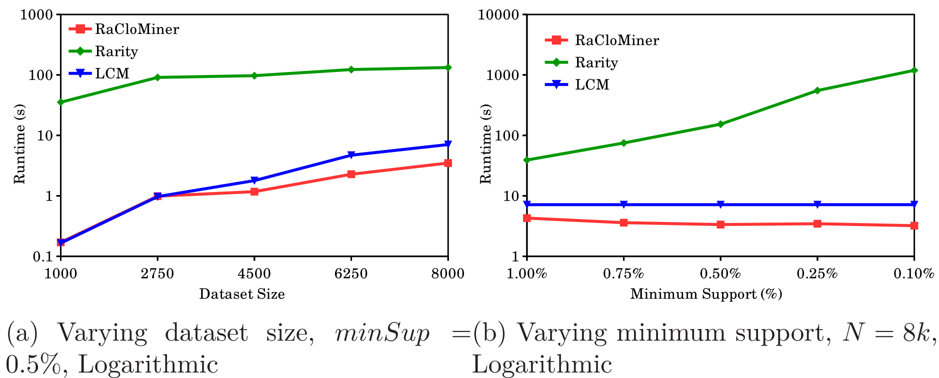
Figure 5.4: Runtime performance on BMS1 dataset, $\delta = 0.3\%, L_R = 15$



(a) Varying dataset size, $minSup = 0.1\%$

(b) Varying minimum support, $N = 30k$

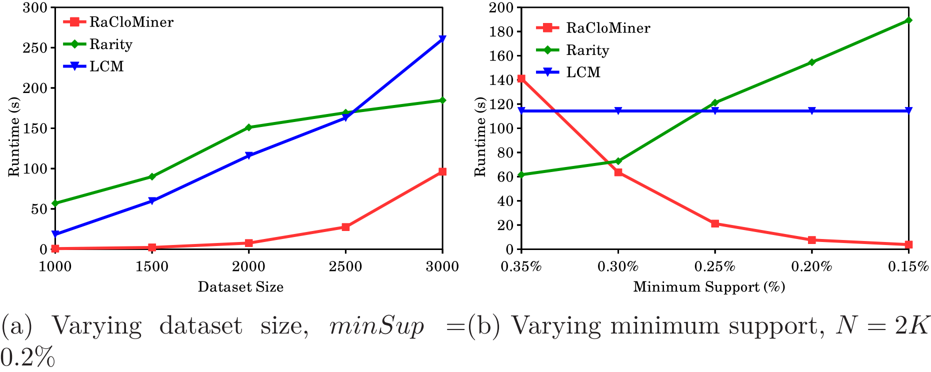
Figure 5.5: Runtime performance on BMS2 dataset, $\delta = 0.3\%, L_R = 15$



(a) Varying dataset size, $minSup = 0.5\%$, Logarithmic

(b) Varying minimum support, $N = 8k$, Logarithmic

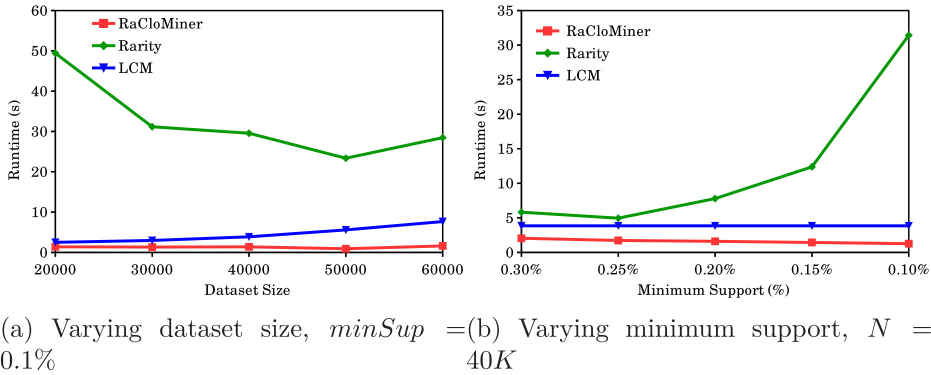
Figure 5.6: Runtime performance on mushrooms dataset, $delta = 30\%, L_R = 10$



(a) Varying dataset size, $minSup = 0.2\%$

(b) Varying minimum support, $N = 2K$

Figure 5.7: Runtime performance on chess dataset, $\delta = 1\%, L_R = 9, L_L = 25$



(a) Varying dataset size, $minSup = 0.1\%$

(b) Varying minimum support, $N = 40K$

Figure 5.8: Runtime performance on connect dataset,$\delta = 30\%, L_R = 9, L_L = 15$

the dataset size increasing. This abnormality is also caused by the expensive pruning step in Rarity.

The delegate of bottom-up based algorithms, LCM, is much faster than Rarity since the number of frequent patterns to be traversed is relatively small in a sparse dataset. Bottom-up based algorithms, with better lattice traversing efficiency, won't waste too much time on traversing the frequent part.

Our RaCloMiner, which is based on the bi-directional framework, is always faster than the other two algorithms. In most cases, we can even achieve 1 to 2 order of magnitude performance boost when compared with the LCM

algorithm. This result meets our expectations. As shown in Figure 5.2, LCM only spend less than 1% of its total runtime on projected databases with respect to less frequent items. In our RaCloMiner, LCM is employed to only traversing this very efficient part. The rest of the dataset, which contains more frequent items and relatively large frequent part on the corresponding lattice, is traversed using a top-down based algorithm. As no time is wasted on traversing the expensive frequent part, our RaCloMiner is much faster than others.

The performance-boosting effect of the bi-directional traversing on the dense dataset is not as dramatic as on the sparse dataset above. Figure 5.6, 5.7 and 5.8 show results of experiments conducted on three dense datasets: mushrooms, chess and connect datasets. The Rarity algorithm is again the slowest one among all three approaches even though it has the shortest transaction length. On the other hand, the LCM algorithm performs pretty well under several settings. For example, when the minimum support value becomes large in chess dataset (Figure 5.7 (b)), the LCM algorithm can even be faster than our RaCloMiner under the same settings. This is because only a small portion of patterns are frequent so that the top-down traversing won't be able to save much runtime. A better threshold value δ could be helpful in such case. Nevertheless, our bi-directional traversing framework should be able to provide at least the same runtime as a bottom-up algorithm by setting the δ value to 1. Overall, our RaCloMiner can still achieve a multiple times performance boosting even on dense datasets.

5.4 Discussion and Conclusion

In this work, we analyzed the factors that might affect the performance in terms of infrequent itemset mining problem. A simple but useful bi-directional traversing itemset mining framework is proposed to mine infrequent patterns efficiently. The closed infrequent itemset mining problem is defined and solved by introducing the RaCloMiner based on our bi-directional traversing framework. Our intensive experiments on real-world datasets show that, by using the bi-directional traversing framework, a simple algorithm,

without advanced data structures or pruning techniques, could still achieve more than 10 times performance boosting when compared with existing options.

However, we should notice that only having this bi-directional traversing is not enough. In some dataset with specific minimum support, the total number of infrequent itemsets is extremely large even if we only consider closed patterns. In such cases, bi-directional traversing might perform almost the same as the bottom-up based algorithm since the time spent on traversing the frequent part is relatively ignorable. In extreme cases, none of those algorithms can finish the task. Directions to improve the itemset mining performance could be introducing further constraints, such as the confidence value, or more condensed representations than the closed set concept.

Many efforts have been put into the area of introducing new condensed representations or restriction criteria in recent years. However, most of them only focus on the frequent itemset mining problem while infrequent patterns may have different requirements in some scenarios. For example, frequent patterns with higher confidence value are usually preferable while infrequent patterns with lower confidence value are more important in some applications since those patterns might represent real unusual events. Designing new condensed representation or restriction criteria for infrequent itemset mining task is still an open question for further investigation.

Chapter 6

Mining Closed Rare Itemsets on NI-Tree

In this chapter, a top-down based closed rare itemset mining algorithm so-called LSCMiner (**L**ow **S**upport **C**losed Miner) is proposed. LSCMiner fully utilized the property of negative itemset tree. The closeness checking step is integrated into the NI-tree subtraction process, which is more efficient than the naive top-down closed rare itemset mining approach described in Chapter 5. Furthermore, LSCMiner can be combined with bottom-up based frequent itemset mining algorithms using the framework proposed in Chapter 5. The combined approach achieves state-of-the-art performance for closed rare itemset mining. Part of the material presented in this chapter has been published in [72].

6.1 Basic Definition

For the ease of reading, basic definitions and notations are introduced here again. Let $\mathcal{I}$ be the universe of items, a subset of $\mathcal{I}$ that contains l items is a l-itemset, denoted as $X = \{x_1, x_2, \ldots, x_l\}$. A transaction dataset $\mathcal{T}$ contains a set of transactions where each transaction $T \in \mathcal{T}$ is an itemset over $\mathcal{I}$. Let $\mathcal{T}(X) = \{T | T \in \mathcal{T}, X \subseteq T\}$ be the set of transactions in $\mathcal{T}$ that contains X, the (absolute) support of X on $\mathcal{T}$ is defined as $|\mathcal{T}(X)|$.

In this work, we tend to find less frequent or low support patterns, i.e., $|\mathcal{T}(X)| \ll |\mathcal{T}|$. Formally speaking, given two user-defined threshold: *minimum support* α and *maximum support* β, we are going to mine patterns X such that $\alpha \leq |\mathcal{T}(X)| < \beta$, where $\alpha \geq 1 \wedge \beta \ll |\mathcal{T}|$. In general, our mining task is the same as infrequent itemset mining since $\beta \ll |\mathcal{T}|$. The parameter α is introduced for more flexibility as users might consider patterns occurred less than α as noise. Conventional frequent itemset mining algorithms can also extract low support patterns by setting their minimum support threshold to α and then removing all frequent patterns with support larger than β.

An itemset X is a *closed* itemset in dataset $\mathcal{T}$ if and only if there is no other itemset Y in $\mathcal{T}$ such that $X \subset Y \wedge |\mathcal{T}(X)| = |\mathcal{T}(Y)|$. The closed itemset concept was first proposed in [109] to address the redundant problem in frequent itemset mining problem. It is a lossless condensed representation: user can determine the support of any frequent itemsets from closed frequent itemsets. The set of closed low support patterns is $\mathcal{LP}$.

A frequent border set $\mathcal{FB}$ is defined as the set of longest patterns such that $|\mathcal{T}(X)| \geq \beta$, which is also known as the maximal frequent itemset [16]. $\mathcal{FB}$ is necessary to make $\mathcal{LP}$ complete. For example, given a pattern $\{ab\}$, if $\exists X \in \mathcal{LP}$ such that $\{ab\} \subseteq X$ but $\nexists X' \in \mathcal{LP}$ such that $X' \subseteq \{ab\}$, then the pattern $\{ab\}$ can be either frequent or not frequent. The border $\mathcal{FB}$ helps in this case to identify whether $\{ab\}$ is frequent or not.

Support Counting on Negative Itemset Tree

The ni-tree [71] is initially proposed in Chapter 3 to mine all infrequent patterns. It stores support information of *negative represented (neg-rep) itemsets*. Neg-rep itemsets are itemsets represented by symbol of items that do not exist in the original itemsets. For example, given $\mathcal{I} = \{a, b, c, d, e, f\}$, an itemset $X = \{a, b, c\}$ can also be represented using the symbol of items not in X, denoted as $\overline{X} = \{d, e, f\}$.

Obviously, X and $\overline{X}$ represent the same information since $X \subseteq T \Leftrightarrow \overline{X} \supseteq \overline{T}$. Let $\overline{\mathcal{T}}$ be the negative dataset formed by neg-rep itemsets, the support of $\overline{X}$ can be defined as the number of neg-rep transactions in $\overline{\mathcal{T}}$ that covered by $\overline{X}$ such that:

$$|\overline{\mathcal{T}}(\overline{X})| := |\{\overline{T}|\overline{T} \in \overline{\mathcal{T}}, \overline{T} \subseteq \overline{X}\}| \Leftrightarrow |\mathcal{T}(X)| = |\overline{\mathcal{T}}(\overline{X})| \quad (6.1)$$

(a)

Tid	Itemset
1	b c
2	d e
3	a e
4	c d e
5	b d e
6	a d e

(b)

Tid	Negative Itemset
1	a d e
2	a b c
3	b c d
4	a b
5	a c
6	b c

Figure 6.1: Transaction dataset and its negative dataset.

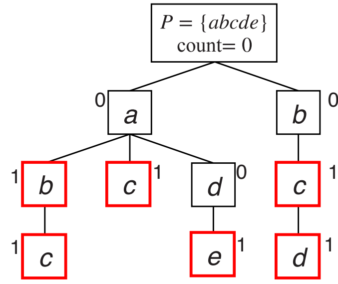


Figure 6.2: The initial ni-tree of dataset in Figure6.1. Red nodes are *t-nodes*.

The ni-tree is a prefix tree, as shown in Figure 6.2. Items are sorted in ascending order concerning their frequency $\mathcal{T}$. Each node n is a triplet $\langle i, c, l\rangle$, where i and c are the item label and its count, l is the list of child nodes. c is initialized to 0. The root node $r = \langle P, c, l\rangle$ stores the current pattern P.

Each transaction $T \in \mathcal{T}$ is converted to $\overline{T}$ and inserted to the ni-tree. The last node, known as the *termination node* or *t-node* for short, will increase its count by 1. Thus, the count of a node n is the number of its corresponding

transactions, i.e. $n.c = |\{\overline{T} \in \overline{\mathcal{T}}, \overline{T} = n.L\}|$, where $n.L$ is the set of items on the path from root to n. According to equation 6.1, $|\mathcal{T}(X)|$ can be computed by aggregating all nodes whose path from the root is fully covered by $\overline{X}$:

$$|\mathcal{T}(X)| = |\overline{\mathcal{T}}(\overline{X})| = \sum_{n.L \subseteq \overline{X}} n.c \tag{6.2}$$

For example, to identify the support of itemset $X = \{de\}$, the count of nodes on paths that covered by $\overline{X} = \mathcal{I} \setminus X = \{abc\}$ are aggregated, which equals to 4. Therefore, the ni-tree can be used to compute the support of a given pattern.

Moreover, given patterns X and X', $X \subset X' \Leftarrow \overline{X} \supset \overline{X'}$, the set of nodes for computing $|\mathcal{T}(X)|$ can be decomposed as: $\{n|n.L \subseteq \overline{X}\} = \{n|n.L \subseteq \overline{X'}\} \cup \{n|n.L \subseteq \overline{X}, n.L \nsubseteq \overline{X'}\}$. Thus, the aggregating process can be decomposed and computed recursively. For example, let $X = \{de\}, X' = \{bcde\}$, the support of X can be obtained by removing nodes on the path covered by $\overline{X'} = \{a\}$, which leads to a new ni-tree that represents the pattern $\{bcde\}$, as shown in Figure 6.3. Then, removing nodes covered by $\overline{X}$ from the second ni-tree will generate the pattern $\{de\}$. Such process is in top-down style. In practice, we only need to create a new root node rather than a brand new ni-tree.

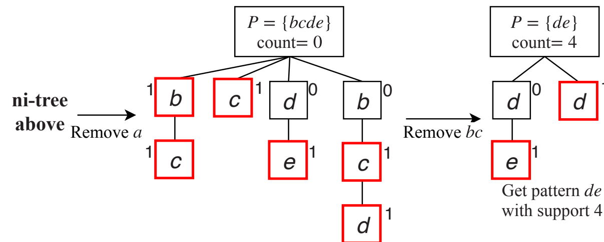


Figure 6.3: Counting support on ni-tree.

6.2 Closed Itemset on Negative Itemset Tree

Though the ni-tree is not initially designed for closed pattern mining, we found that the closeness can be determined readily by using t-nodes.

Closed Itemset Determination

According to the definition, an essential property of a closed pattern X is that its support must be different from its supersets. In ni-tree, the count value of any node, except t-nodes, is 0. Thus, if the count of all removed nodes is 0, the generated pattern is not closed. A closed pattern can only be achieved if at least one t-node is involved in the aggregating process. Formally speaking:

Theorem 6.2.1
Given $\mathcal{I}$ and the initial ni-tree, let N_X be the set of nodes been removed from the initial ni-tree to achieve the pattern X. Let $N_X^t \subseteq N_X$ be the set of t-nodes been removed. Then pattern X is closed if and only if the set of items been removed ($\overline{X}$) equals to the set of items on paths to t-nodes:

$$\mathcal{I} \setminus X = \overline{X} = \bigcup_{n \in N_X^t} n.L \tag{6.3}$$

Proof. Obviously, $N_X = \{n | n.L \subseteq \overline{X}\}, N_X \supseteq N_X^t$. Thus,

$$\overline{X} = \bigcup_{n \in N_X} n.L \supseteq \bigcup_{n \in N_X^t} n.L \tag{6.4}$$

As non-terminated nodes are counted at 0 in the ni-tree, the support of pattern X is the sum of all t-nodes:

$$|\mathcal{T}(X)| = |\overline{\mathcal{T}}(\overline{X})| = \sum_{n.L \subseteq \overline{X}} n.c = \sum_{n \in N_X^t} n.c \tag{6.5}$$

Let $M = \overline{X} \setminus (\bigcup_{n \in N_X^t} n.L)$. Thus, any node $n' \in N_X$ with item $n'.i \in M$ is not on the path to a t-node in N_X^t. Removing such nodes or not won't affect the support value, i.e. $|\mathcal{T}(X)| = |\mathcal{T}(X \cup M)|$. By closeness definition, $M = \emptyset \Leftrightarrow X$ is closed. □ □

In short, a pattern X is closed if all removed items can be found on paths towards removed t-nodes. For example, in the ni-tree of Figure 6.2, pattern

$X = \{be\}$ is not closed since item d is removed but its corresponding nodes are not on a path towards t-nodes covered by $\overline{X} = \{acd\}$. On the other hand, itemset $X = \{de\}$ is closed.

Naïve Method

According to Theorem 6.2.1, top-down closed pattern mining can be realized by simply enumerating and removing all combinations of paths towards t-nodes. The ni-tree is slightly adapted. The root node and each t-node stores a list of pointers (l_t) linked to their child t-nodes, as shown in Figure 6.4. In each step, nodes on the path from one t-node (excluding) to its child t-node (including) are removed together, which guarantees that only closed patterns are generated. Figure 6.4 illustrates an example. By removing all nodes on the path to t-node 1, a new ni-tree is generated, and the corresponding closed pattern $\{de\}$ is returned. Then removing t-node 2 in the new ni-tree lead to another closed pattern $\{e\}$. Enumerating all removing combinations generate all closed patterns.

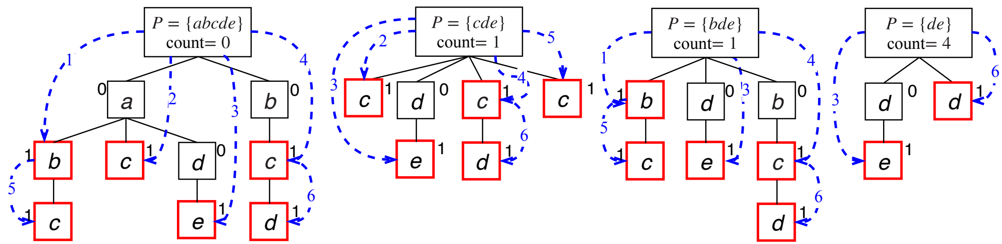


Figure 6.4: The adapted initial ni-tree with t-node links (blue) and corresponding ni-tree by removing t-node 1, 2 or 1, 2 together. Each link is marked with the t-node id. In each step, only child t-nodes of root are considered (e.g. node 6 can only be removed after node 4).

The main disadvantage of this naive approach is the duplicate accessing problem. For example, given $\mathcal{I} = \{abcde\}$, the pattern $X = \{ab\}$ can be achieved by either removing $\{cd\}$ and $\{ce\}$ or removing $\{cd\}$ and $\{de\}$. A pattern X might be accessed repeatedly up to $O(2^{|\mathcal{T}(X)|})$ times. Extra duplicate checking and pruning step are necessary. By examining patterns discovered so far, we can avoid the majority of duplicates. However, the over-

head of the pruning step plus remaining duplicates are still time-consuming. Indeed, this naïve method is the top-down part used in the bi-directional traversing framework [74].

6.3 Algorithm: LSCMiner

Divide-and-Conquer Paradigm

The naive approach described above is a top-down based algorithm. However, it is not efficient due to the expensive duplicate accessing problem. To take advantage of top-down traversing, we propose our **L**ow **S**upport **C**losed Miner (LSCMiner), which employs the depth-first traversing strategy with novel closeness checking and pruning steps. The general mining process employed a divide-and-conquer paradigm, which is commonly used in bottom-up based algorithms. The main difference is that we remove items recursively, rather than grow patterns.

First of all, let operators $\prec$ and $\succ$ denote the concept of *"before (smaller)"* and *"after (larger)"* with respect to the ascending frequency order used by the ni-tree. Given $\mathcal{I} = \{a \prec b \prec c \prec \dots\}$, the top-down mining process removes items recursively, which can be represented as a tree as shown in Figure 6.5. We call the tree above as the *deletion tree*. Each node in the tree is the set of items to be removed, known as the *deletion set*. Given a node in the deletion tree, we say that deletion sets in its sub-tree and right to it are *under* or *after* the deletion set in the node, as shown in Figure 6.5.

The first challenge is to combine the closeness checking process with the divide-and-conquer paradigm. According to Theorem 6.2.1, we need to check if every removed item can be found on paths to removed t-nodes. To solve the problem, we let each t-node n^t contains a list $n^t.L$, which stores items on the path from itself (including) to its proceeding t-node (excluding), as shown in Figure 6.6. During the removing process, a set U is maintained to track items that are not covered by paths towards removed t-nodes yet. In one recursive step, we first add the current item to U. If a t-node n^t is removed, all items exist in $n^t.L$ are removed from U. When $U = \emptyset$, we knew

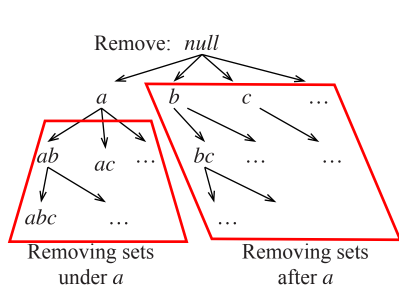


Figure 6.5: We solve the mining task by removing items in a recursive way. Such process can be represented as a tree.

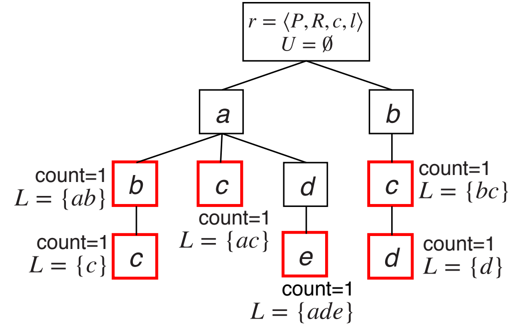


Figure 6.6: The adapted ni-tree used in LSCMiner. R is the set of items been removed so far.

that the current pattern is closed.

Figure 6.7 gives an example of the closed pattern mining process. We first remove item a, no t-node is removed right now. Thus, $U = \{a\}$ and the pattern $\{bcde\}$ is not closed. Then, we recursively remove b from the current ni-tree. There is a t-node of b is removed and $U = \{ab\} \setminus \{ab\} = \emptyset$. Pattern $\{cde\}$ is closed and should be added to the result set.

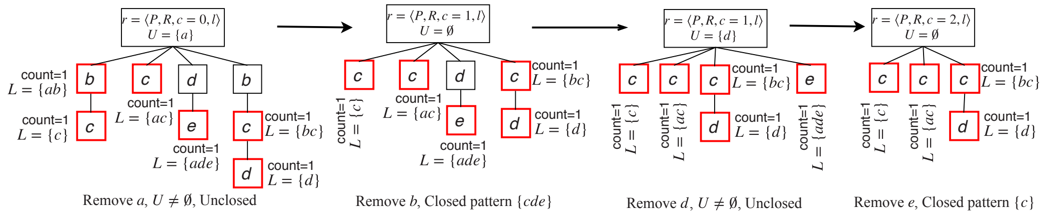


Figure 6.7: Recursive steps of the removing process $a \to ab \to abd \to abde$.

Algorithm 6.1 illustrates the pseudo code of the LSCMiner. Each iteration step removes one item (Line 5). If there are t-nodes in removed nodes list l_i, we remove items from U and aggregate counts (Line 10-13). If the current candidate pattern is not frequent, we attach nodes with larger item to the new ni-tree root for the next recursive call (Line 18). The recursive mining process is continued until the aggregated count is larger than the given maximum threshold β. Variables iM_1 and iM_2 are pruning thresholds as described later.

Algorithm 6.1: LSCMiner

Input: Ni-tree root r, Minimum support α, Maximum support β
Output: Infrequent Itemset List $\mathcal{LP}$, Frequent Border List $\mathcal{FB}$

1 $\mathcal{LP} \leftarrow \emptyset, \mathcal{FB} \leftarrow \emptyset$;
2 `LSCMiner`(r,$\emptyset$,$+\infty$, $-\infty$) ;
3 **return** $\mathcal{LP}, \mathcal{FB}$;
4 **Function** `LSCMiner`(r, U, iM_1, iM_2)
5   **foreach** *Item* $i \in r.l \wedge i \preceq iM_1$ **do**
6         $l_i \leftarrow$List of nodes in $r.l$ with label i ;
7         $U' \leftarrow U \cup \{i\}$, $P' \leftarrow r.P \setminus \{i\}$, $c' \leftarrow r.c$;
        `/* Closeness checking */`
8   **foreach** *Termination node* $n \in l_i$ **do**
9             $U' \leftarrow U' \setminus n.is$, $c' \leftarrow c' + n.c$;
10   **end**
11   **if** $c' < \beta$ **then**
12   **if** $c' \geq \alpha \wedge U' = \emptyset$ **then**
13   Add P' to $\mathcal{LP}$
14   **end**
15             $l' \leftarrow \{n' \in r.l | n'.i \succ i\} \cup \{\bigcup_{n \in l_i} n.l\}$, $r' \leftarrow \{P', c', l'\}$;
            `/* Initial new end index */`
16   **if** P' *is closed* **then**
17                 $iM_1', iM_2' \leftarrow +\infty, -\infty$;
18   **else**
19                 $iM_1', iM_2' \leftarrow$`UpperBound`(l, iM_2);
20   **end**
21   `LCSMiner`(r', U', iM_1', iM_2') ;
22   **if** *No closed pattern generated in the recursive call above* **then**
23   **Break**; `// Trial-and-Error Pruning`
24   **end**
25   **else**
26   **if** $U' = \emptyset$ **then**
27   Add P' to $\mathcal{FB}$;
28   **end**
29   **end**
30   **end**
31 **end**

Pruning

An efficient algorithm should be able to prune unclosed itemsets as early as possible, known as the *"look ahead"* ability [109][97]. In our LSCMiner, we fully utilized the closeness property of the ni-tree. Two types of pruning methods are utilized.

Trial-and-Error Pruning

Our first pruning method (Line 27, Algorithm 6.1) is based on the following observation:

Theorem 6.3.1
Given the current ni-tree root r and the current unclosed items set $U \neq \emptyset$. Let R be the set of items been removed so far. If $\exists i \in r.l$ such that no closed pattern in deletion sets under $\{R \cup i\}$, then there is also no closed pattern in deletion sets after $\{R \cup i\}$.

Proof. The sub-ni-tree under item i must contain at least one t-node. Let l be the set of items from i (including) to a t-node n^t (including) in its sub-ni-tree. Obviously, we have $n^t.L \supseteq l$ and $i \in n^t.L$.

Let deletion sets after i be $R_{\succ i}$. Assuming the deletion set of i and deletion sets under i are not closed. If $\exists p \in R_{\succ i}$ which will lead to a closed pattern, then removing all items in $\{p \cup l\}$ will also lead to a closed pattern (by further removing the node n^t mentioned above). Obviously, $\{p \cup l\}$ is a deletion set of i or under i, which is contradict to our assumption. □ □

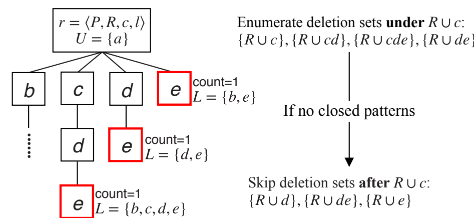


Figure 6.8: Given the current unclosed ni-tree, Trial-and-Error pruning will try all deletion sets under $R \cup c$. If no closed patterns exists, then later deletion sets can be skipped.

In short, if removing i does not generate a closed pattern, the recursion call will be executed (Line 21, Algorithm 6.1). This recursive call will try all possible combinations of items with respect to i. If no closed pattern is generated, iterations on items (Line 5, Algorithm 6.1) after i can be canceled.

Upper-bound Pruning

The second pruning technique computes the largest possible item as an upper bound for the next recursion step. Given the current removed item i and the list of nodes to be removed l_i, assuming all nodes in l_i are not terminated, then the largest possible item iM'_1 that can be removed in the next recursion step is the largest item among all children of nodes in l_i.

The reason is straightforward: the item i will be covered by a t-node if and only if at least one of its children is removed. If we remove an item $i' \succ iM'_1$ in the next recursion step, all items to be removed in the future are also larger than iM'_1. Thus, it is impossible to reach a t-node that covers i. For example, given the left ni-tree in Figure 6.9, assuming now we are removing item a, which results in the right ni-tree in Figure 6.9. However, t-nodes that cover a only exist in the sub-ni-tree of a. Thus, the upper bound for item removing on the second ni-tree is b. Further removing process on items after b is pruned since the item a will never be covered.

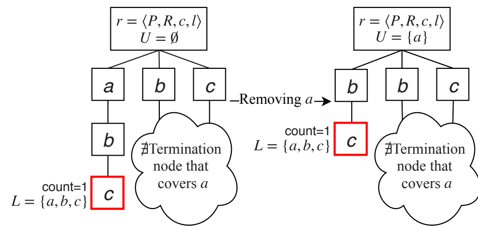


Figure 6.9: The maximum item under nodes of a is $iM'_1 = b$. Later removing process on the right ni-tree must removing b first since otherwise, t-nodes that cover a will not be removed.

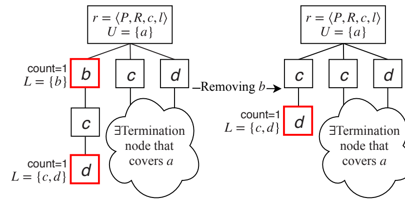


Figure 6.10: There is a t-node that covers b. Then the maximum upper bound up to now (iM'_2), which equals to the upper bound when removing a, is used as the upper bound for further removing process on the right ni-tree.

The above upper bound assumes that $\nexists n \in l_i$, i.e., item i can only be

covered by children of nodes in l_i. However, if one node of item i is terminated, then i is covered by a node of itself. Removing items larger than the upper bound iM_1' can still lead to closed patterns. Another weaker upper bound iM_2' is introduced for this case, which is defined as the largest upper bound, except for infinity, among all previous recursion steps. For example, assuming the left ni-tree in Figure 6.10 is achieved by removing item a, and the right ni-tree is achieved by further removing item b. Since node b is terminated, removing items larger than its children is valid. However, the previously removed item a needs to be covered so that the upper bound iM_2' is the upper bound when removing a. Algorithm 6.2 computes both upper bounds described here.

Algorithm 6.2: Compute the new upper bound.

```
1  Function UpperBound(l_i, iM_2)
2      iM'_1 ← −∞
3      foreach n ∈ l_i ∧ n is not t-node do
4          x_last ←Last item in n.l
5          if iM'_1 ≺ x_last then
6              iM'_1 ← x_last
7          end
8      end
9      if iM'_1 ≠ −∞ then
10         iM'_2 ← max(iM'_1, iM_2)
11     end
12     if ∃n ∈ l_i, n is terminated then
13         iM'_1 ← iM'_2
14     end
15     return iM'_1, iM'_2
16 end
```

Complexity

Pattern mining is an NP-hard problem. The overall runtime is highly dependent on the number of desired patterns. For instance, one of the most efficient frequent closed pattern mining algorithms, LCM [97], declares that

it extracts each closed pattern in polynomial time: $O(P(|\mathcal{T}|))$. Let $\mathcal{U}$ and $\mathcal{UC}$ be the set of desired and undesired patterns, the time complexity per itemset of the LCM algorithm can be written as: $O(\frac{|\mathcal{U}|+|\mathcal{UC}|}{|\mathcal{U}|}P(|\mathcal{T}|))$, where $\mathcal{UC}$ contains frequent patterns in the low support closed pattern mining scenario.

Our approach can also achieve the same level of complexity. Given the current ni-tree root r and the current unclosed items set U, removing item i from the child list $r.l$ involves the following steps:

1. aggregate counts in removed nodes, which requires $O(|l_i|)$ time, where l_i is the list of nodes in $r.l$ labeled with i.
2. closeness checking if t-nodes exist, which requires $O(|U|\log(|n^t.L|))$ time, where $n^t.L$ is the set of items in a t-node and binary search is employed
3. add children of nodes in l_i to the new root node r', which takes $O(\sum_{n\in l_i}|n.l|)$ time.
4. add all nodes in $r.l$ with label larger than i to the new root node r', which requires $O(|l_{\succ i}|)$ time, where $l_{\succ i}$ is the list of nodes.

The total complexity is $O(|l_i| + |U|\log|n^t.L| + \sum_{n\in l_i}|n.l| + |l_{\succ i}|)$. $|U|$ and $|n^t.L|$ are limited to the size of a single transaction so that the second term can be seen as a constant. The length of $l_i, l_{\succ i}$ and $n.l$ are limited to the size of the dataset. Thus, the complexity of removing item i is polynomial. A closed itemset X is achieved by removing items in $\overline{X}$. The complexity to extract X is $O(\sum_{i\in\overline{X}}P(|\mathcal{T}|)) \in O(P(|\mathcal{T}|))$ since $|\overline{X}|$ is small compared to $|\mathcal{T}|$. Considering that our approach also accesses some unclosed patterns, its complexity per itemset is also $O(\frac{|\mathcal{U}|+|\mathcal{UC}|}{|\mathcal{U}|}P(|\mathcal{T}|))$, where $\mathcal{UC}$ are those unclosed patterns.

In terms of memory complexity, it is obvious that our approach is limited by the size of the dataset, similar to algorithms such as FPGrowth [51]. However, LSCMiner has to store the negative dataset such that a scale factor $s = \frac{|\mathcal{I}|}{\text{avg. transaction length}}$ exists, known as the sparsity of the dataset. s can be huge on sparse dataset. In this case, the bi-directional traversing framework

proposed in [74] can be used so that our LSCMiner only need to handle the densest part of a dataset.

6.4 Experiments

We first conduct the runtime performance of our LSCMiner. In experiments, the naive approach described in Section 6.2 represents the performance of a simple top-down based algorithm. The LCM [97] algorithm represents the most efficient bottom-up based algorithm in solving low support pattern mining problem. We also conduct the bi-directional traversing framework [74] by combining the LCM algorithm with our LSCMiner. Other infrequent pattern mining algorithms are not included since they are either represented by LCM (bottom-up based) or too memory expensive to finish (apriori alike).

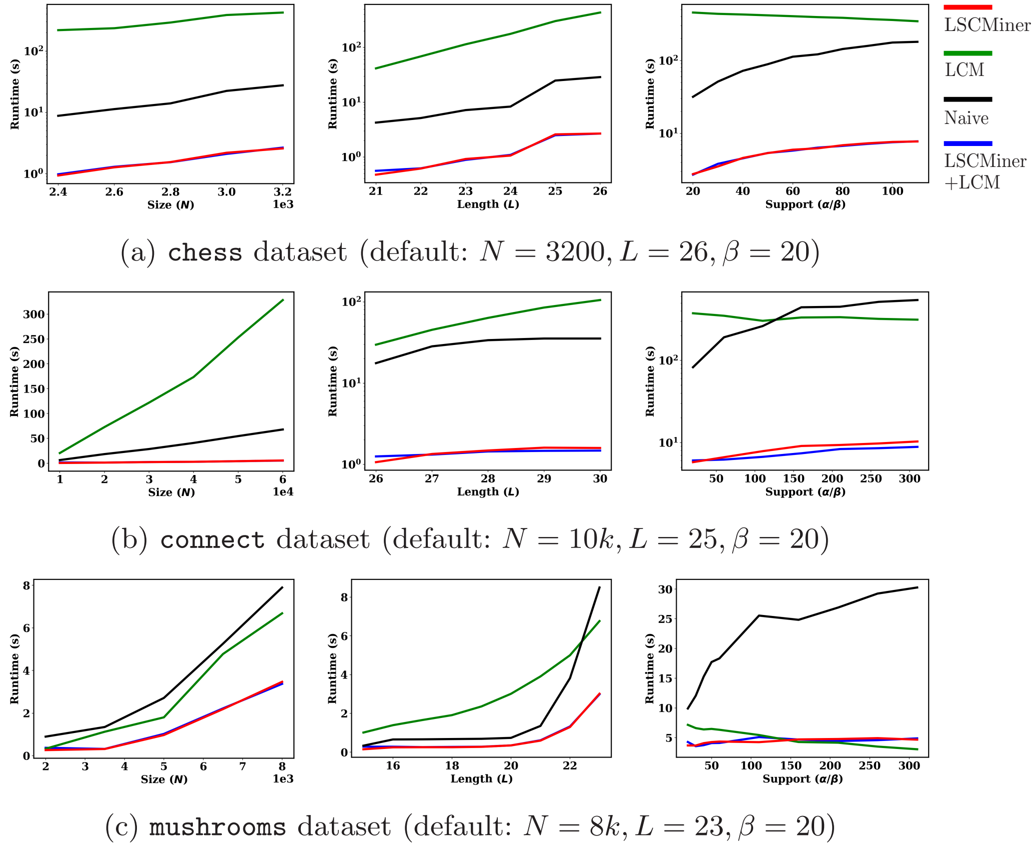
Database	Size (N)	Items ($\mathcal{I}$)	Avg. length (L)
mushrooms	8k	119	23
chess	3k	75	37
connect	67k	129	43
accident	340k	468	33.8
kddcup99	1000k	135	16
BMS1	59k	497	2.5

Figure 6.11: Real-life datasets in our experiments.

Algorithms are implemented using Java. The LCM implementation comes from [37]. 6 real-world datasets obtained from the fimi repository (`http://fimi.ua.ac.be/data/`) are used as our test datasets. Necessary information of those datasets are listed in Figure 6.11. There are three small dense datasets, two large dense datasets, and one sparse dataset. First N transactions and first L items in each transaction are used in our experiments. We are interested in the time difference in accessing patterns on a certain level of support from different directions. Thus, we set $\alpha = \beta - 10$. The default value of N, L and β are provided in each experiment. The splitting threshold for the bi-directional framework is set to $\delta = 1\%$.

Dense Data

Figure 6.12 illustrates the performance on small dense dataset. On these datasets, the bottom-up algorithm, LCM, is up to two order of magnitude



(a) chess dataset (default: $N = 3200, L = 26, \beta = 20$)

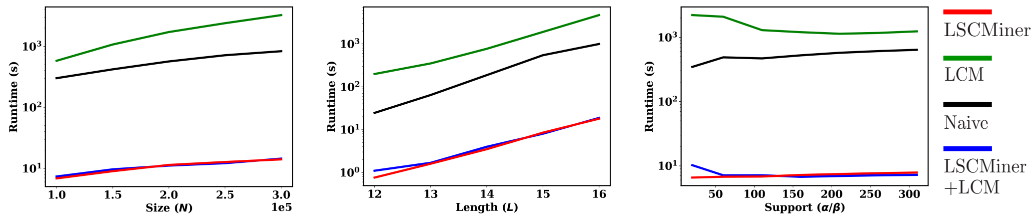
(b) connect dataset (default: $N = 10k, L = 25, \beta = 20$)

(c) mushrooms dataset (default: $N = 8k, L = 23, \beta = 20$)

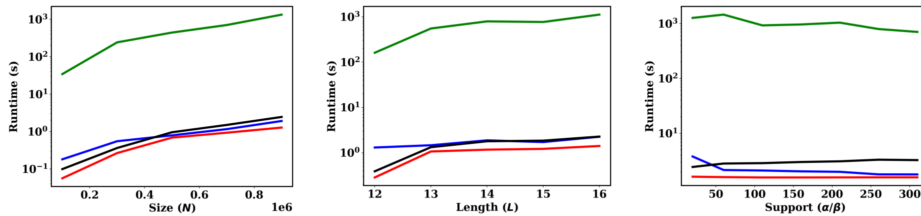
Figure 6.12: Runtime on small dense dataset.

slower than our top-down LSCMiner on the first two datasets. It is even slower than the naive approach under some settings. This is mainly because the bottom-up LCM algorithm has to traverse all frequent patterns. When β increased, i.e., we become more interested in frequent patterns, the runtime of LCM is reduced and may surpass top-down approaches since it needs to traverse less frequent patterns. On the `mushrooms` dataset, top-down approach is slower with $\beta > 150$. This is mainly because that the `mushrooms` dataset has less number of patterns. Our LSCMiner is very efficient. Its runtime grows similar to the LCM approach with increasing dataset size, which indicates that the time complexity of both approaches is on the same level. The combined approach is also efficient under most settings. Our LSCMiner under the bi-directional framework only need to handle the densest part of the dataset, which reduces the memory consumption, as discussed in

Section 16. However, the slowness of the bottom-up part under some settings drag down its performance. The performance gap between top-down and bottom-up approaches is further enlarged on large dense datasets, as shown in Figure 6.13. Both `accidents` and `kddcup99` datasets have larger size and longer transactions. The LCM algorithm is up to 3 order of magnitude slower than our LSCMiner. Even the naive approach is better under most cases. Though increasing β slows down our LSCMiner, it is still hard for LCM algorithm to overtake in the low support pattern mining scenario.



(a) `accidens` dataset (default: $N = 300k, L = 15, \beta = 50$)



(b) `kddcup99` dataset (default: $N = 1000k, L = 16, \beta = 20$)

Figure 6.13: Runtime on large dense datasets.

Sparse Data

In theory, bottom-up algorithms should perform better than our top-down approach on a sparse dataset. According to our analysis above, both LCM and our LSCMiner have a complexity of $O(\frac{|\mathcal{C}|+|\mathcal{UC}|}{|\mathcal{C}|}P(|\mathcal{T}|))$. In the case of sparse datasets, $|\mathcal{UC}|$ in LCM is much smaller since the number of frequent patterns is tiny. On the other hand, a sparse dataset leads to a huge ni-tree, which is a substantial overhead for LSCMiner. Figure 6.14 illustrates the runtime performance on a sparse dataset `BMS2`. The results fulfill our expectations. The combined approach performs the best since we can take

advantage of both bottom-up and top-down algorithms.

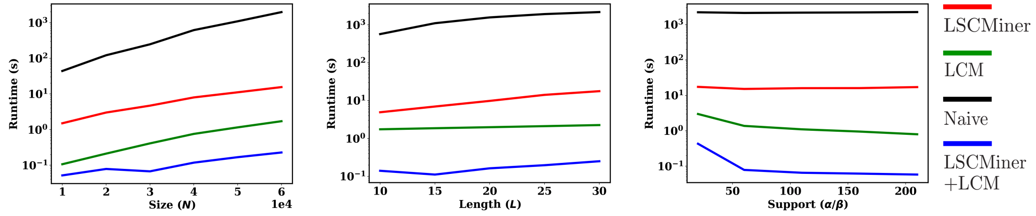


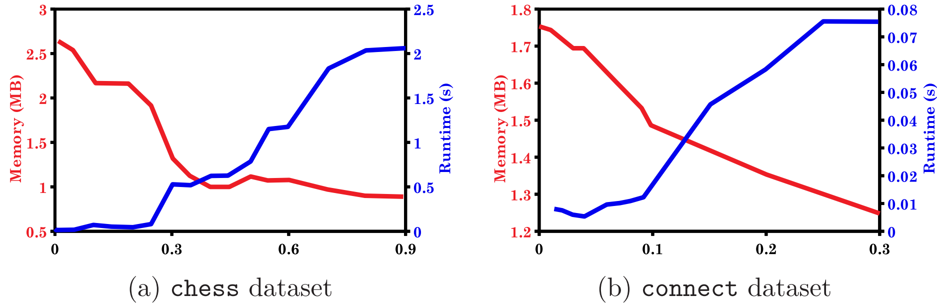
Figure 6.14: Runtime on BMS1 dataset (default: $N = 60k, L = 30, \beta = 20$)

Memory-Performance Trade-off

As discussed above, the performance of our LSCMiner is limited by the dataset size. An extra constant scale factor s exists since the ni-tree stores the negative dataset. Sparsity is a crucial factor that affects the memory consumption of LSCMiner. By applying the bi-directional traversing framework and adjusting the value of splitting threshold δ, the LSCMiner only need to traverse the densest part of the dataset, which costs much fewer memories. We can still benefit from the efficient top-down traversing since top-down traversing is powerful on dense dataset while bottom-up traversing is good at the sparse dataset, as shown in experiments above. In this section, we study the relation between memory consumption and runtime performance by investigating how the trade-off behaviors with respect to different (relative) dividing threshold δ.

Two dense datasets, `chess` and `connect`, are selected as representatives since they have both sparse and dense parts. We measure the memory consumption using the first 1k transactions in each dataset. The runtime value for `connect` dataset is measured with the first 10k transactions instead. When the value of δ is close to 0, all patterns are extracted by the LSCMiner. When the value of δ is close to the relative support of the most frequent item, the bottom-up approach extracts all patterns. Thus, by increasing δ, the bi-directional traversing is moving from purely LSCMiner to purely LCM algorithm.

On the `chess` dataset, the runtime of the bi-directional framework in-



(a) `chess` dataset

(b) `connect` dataset

Figure 6.15: Memory consumption and runtime under different δ values. δ is set up to 0.4 on `connect` dataset since almost all items occurred less than 40%.

creased about 20 times while the memory consumption decreased about 2.5 times when moving from LSCMiner to LCM approach. On the `connect` dataset, the runtime increased about 7 times while the memory consumption decreased about 30%. LSCMiner is beneficial on both datasets: we spend some memory but get much better performance.

6.5 Conclusion

We present a very efficient low support closed pattern mining algorithm, LSCMiner, which avoids traversing undesired frequent patterns. It is particularly effective on datasets with huge amounts of frequent patterns. Though it is memory expensive to store the ni-tree, much better runtime performance is achieved in return. Furthermore, we can balance the memory consumption and runtime performance by using the bi-directional traversing framework. If only those dense datasets are considered, our LSCMiner guarantees to provide the best performance in time complexity.

Part II

Interval-based Temporal Pattern Mining

Chapter 7

Introduction

The need to analyze information aligned with the temporal attribute from streams arises in various applications, such as smart home data analysis or stock fluctuation data analysis. According to the characteristics of these data, challenges need to be overcome in three most important aspects: mining *sequential pattern from streams*, interpreting the *temporal information* and handling *interval-based event*.

7.1 Towards Interval-based Temporal Pattern Mining on Streams

The story of interval-based temporal pattern mining starts from sequential pattern mining. Sequential pattern mining (SPM) is a well-studied topic for decades. Algorithms such as PrefixSpan [50] were proposed in the last decades to generate sequential patterns from a static database efficiently. Conventional sequential pattern mining approaches treat events as points on the timeline. Predefined relationships such as *before, equal* and *after* are considered. However, the limited diversity of predefined relationships makes it hard to describe heterogeneous temporal information accurately.

Temporal pattern mining (TPM) is an extension to SPM in which temporal information of events is also considered. Temporal information is valuable in a variety of applications. For example, in disease dataset, one symptom

is *"1 minute before"* and *"1 week before"* another symptom must be considered as different cases. Treating different temporal information as the same relationship in SPM is problematic in this case. The noise introduced by the temporal information can not be handled by the SPM approaches either. If noise exists in the temporal information, two events that happened simultaneously are hard to retrieve as *"equal"*. Researches such as [39] were proposed in recent years to generate temporal patterns from a static temporal database. However, to the best of our knowledge, the task of mining temporal patterns from stream data has not been conducted in the literature yet.

Stream data analysis is one of the most important fields in recent years. Additional constraints on the analysis method are posed when considering the complexity of streaming data: it is impossible to store all the stream data in memory due to the immense size of data; each stream event can only be accessed once to avoid possible blocking operations; current results should be available immediately on demand. Furthermore, for a correct and more balanced extraction of the underlying knowledge in a stream environment, recent objects should be given more importance than older ones as they usually reflect current trends of the data more accurately.

Sequential pattern mining on stream data is the one of the challenges to be tackled in this part. The interest is in the patterns that are frequently happening on a stream and their evolution over time. Mining frequent sequential patterns from a single stream is non-trivial since it is hard to identify the support of patterns. Moreover, frequent patterns may appear not only in a *single* stream but also across *multiple* streams. For example, each stock market is a stream of stock fluctuation. The price of a stock may also be affected by other stocks in different markets. Patterns happened within a single stream are called *intra-pattern* while patterns happened across multiple streams are called *inter-pattern*. As in [53], multiple streams will be merged into a single stream to extract both types of patterns, as illustrated in Figure 7.1. Thus, sequential pattern mining on a single stream is the core problem conducted in here. Existing stream sequential pattern mining approaches either require multiple streams or cut the single stream into batches. A static

database is generated during the stream mining process, and conventional sequential pattern mining algorithms are applied. Both strategies have their own drawbacks. In this part, we proposed to employ the sliding window model and an estimated support to extract frequent patterns.

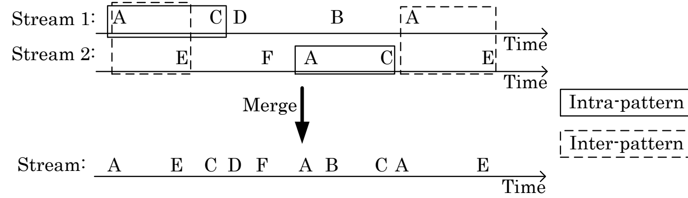


Figure 7.1: Streams will be merged together in order to find *intra-* and *inter-*pattern.

Furthermore, events in the real world usually persist for a period. Consider the case of a smart home data consisting of the ON/OFF states of electrical appliances in a normal household usage [63]. The microwave is used for 5 minutes *"during"* the usage of the electrical oven. The point-based event model used in SPM can be seen as a special case of the interval-based event model. More complicated relationships among interval events need to be considered [5]. In this part, we show that our model can be extended easily to handle the interval-based event streams further.

In a nutshell, the contributions can be summarized as follows. Similarity measures are defined based on the temporal information, and frequent patterns are generated based on clustering results. Multiple streams are merged and processed as a *single* stream so that both intra-and inter-patterns are kept. A stream clustering approach and an estimated support are introduced to mine frequent patterns incrementally.

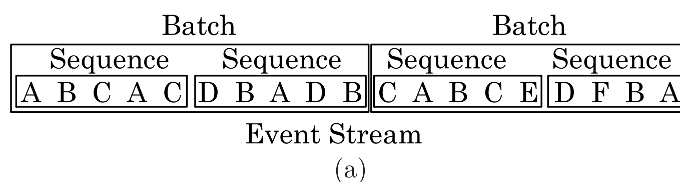
7.2 Related Works

In the literature, there exist many researches conduct the problem of mining sequential patterns from streams and mining temporal patterns from a static database. However, to the best of our knowledge, the issue of mining temporal patterns from streams has not been studied. In this section, we

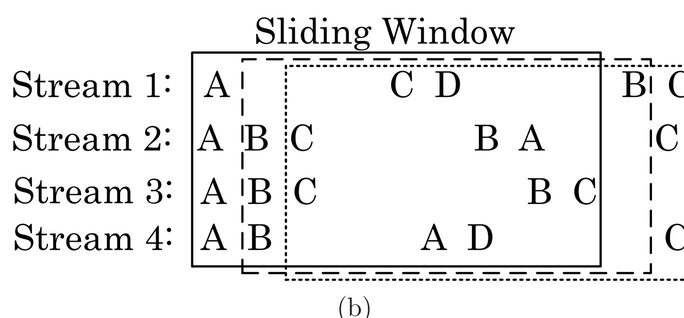
divide our related works into two parts: SPM on streams and TPM on a static database.

Point-based Sequential Pattern Mining on Streams

Existing stream mining approaches only focus on the point-based case. Furthermore, a static database is maintained during the process so that the conventional concept of support could be inherited, i.e., the count of pattern divide by the total number of sequences. Old patterns are not decaying but pruned based on the current status of the static database. Applying stream clustering algorithm on these approaches is not a trivial matter.



(a)



(b)

Figure 7.2: Mining models employed in point-based stream mining. Both (a) Batch and (b) Sliding Window provide a static database.

Batch based approaches [81, 24] cut the stream into sequences. Each batch contains multiple sequences to form a static databases. Conventional sequential pattern mining approach, such as PrefixSpan [50], is then applied to each batch to generate the latest patterns. This approach is efficient but loses patterns came across multiple batches. Other approaches, such as [22, 55], extract sequential patterns incrementally based on the sliding window model. Multiple streams are processed individually so that patterns came across multiple streams might be lost. Both types of approach are

illustrated in Figure 7.2.

In contrast, multiple streams are merged into a single stream, and a sliding window is applied in this work. No static database is created during the process. To the best of our knowledge, [47] is the only work that makes use of a similar setting except for the temporal information. However, the count of a pattern in the current window, rather than a percentage value, is used as the support. When the speed of stream (the number of events arrived in one unit time) is changing, it was difficult to tell whether a pattern frequent or not with its count value. For example, if the speed of stream increased dramatically, everything might become frequent.

Point-based Temporal Pattern Mining on Static Database

In SPM, predefined relationships such as *"before"* and *"after"* are used, and the temporal information is ignored during the mining process. TPM approaches extend SPM ideas by taking the temporal information into consideration, which provides more informative patterns.

One of the strategies to handle temporal information is introducing more predefined relationships, for example, *"small before"* and *"large before"*. The length of the gap between events was quantized, and more accurate relationships between events were defined in approaches [105, 54]. Other approaches, such as [21, 26], employed the fuzzy logic technique to express the temporal information. However, the quantization functions or relationships were still predefined in those approaches, which makes it difficult to reveal the temporal information precisely.

Instead of using predefined temporal relationships, temporal patterns can also be learned from the data by clustering. One of the examples is [39] which generates temporally annotated patterns ($\mathcal{TAS}$) by the clustering of temporal information. The length of the gap between consecutive events is used to present the temporal information, for example, "$A \xrightarrow{3.5} B \xrightarrow{1.5} C$" represents "$A$ *before* B *before* C" with gap length 3.5 and 1.5 in between. Each sequence is treated as a point in high-dimensional space. For example, the sequence above is a point in space $\{AB, BC\}$ with value $(3.5, 1.5)$.

The idea of PrefixSpan is employed in most of these approaches which require sequences that support the same pattern mast share the same events order. When ordering event based on their happening time, sequences with similar temporal information may generate entirely different patterns due to noise. For example, events B and C might have a *equal* relationship in sequences shown in Figure 7.3. Clustering approaches such as [39] can not alleviate the problem since the gap information is used. Similar sequences will locate in different spaces due to noise. For example, similar sequences in Figure 7.3 are points in various spaces $\{AB, BC\}$ and $\{AC, CB\}$. In this work, we propose to use the temporal information aligned with each event to represent temporal sequences and patterns.

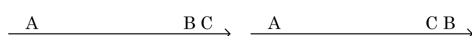


Figure 7.3: Two sequences with the same event types and similar temporal information, but different events order due to some natural noise.

7.3 Fundamental Sequential Pattern Mining Algorithm

In general, sequential pattern mining algorithms are similar to frequent itemset mining algorithms. Given a dataset of event sequences, it extracts frequent subsequences as sequential patterns. The main difference between sequential pattern mining and frequent itemset mining is that the order of events is also considered in sequential patterns. Therefore, the same type of events can occur multiple times in one sequence. Table 7.1 gives an example of a sequence dataset. Let $\mathcal{I}$ be the universe of events, the sequence dataset $\mathcal{S}$ contains $N = 6$ sequences. Each sequence $S \in \mathcal{S}$ is a non-empty list formed by events in $\mathcal{I}$. The support of a pattern (subsequence) X is defined as the number of sequences $S \in \mathcal{S}$ such that X is a subsequence of S, denoted as $X.supp = |\mathcal{S}(X)|$. Sequential pattern mining aims at extracting all subsequences with support greater than or equal to the given minimum support threshold $minSup$.

Sequential pattern mining algorithms can also be divided into Apriori-based and Pattern growth (divide-and-conquer) based. The apriori-based algorithm, GSP [91], was first introduced to address the sequential pattern mining problem. Then, more efficient algorithms such as FreeSpan [49] and PrefixSpan [50] are proposed. They utilized the divide-and-conquer paradigm with the dataset projection technique. Other popular sequential pattern mining algorithms also followed a similar structure but equipped with more advanced data representations and pruning techniques—for example, SPAM [8] stores the sequence dataset using a vertical bitmap representation.

Sid	Sequences
1	A B C A D
2	A B D A
3	B C B A
4	A B D C
5	E B A C
6	D E B A C

Table 7.1: An example of sequence database

Sid	Sequences (temporal information)
1	A(0.6) B(0.8) C(1.2) A(1.3) D(2.0)
2	A(0.5) B(0.7) D(1.9) A(2.5)
3	B(0.8) C(1.3) B(2.1) A(2.4)
4	A(1.0) B(1.3) D(2.2) C(3.0)
5	E(0.5) B(1.1) A(2.1) C(4.0)
6	D(0.9) E(1.0) B(1.8) A(2.1) C(2.2)

Table 7.2: An example of temporal sequence database, each event is aligned with a timestamp.

Temporal pattern mining is developed upon sequential pattern mining by taking the temporal information of each event in the sequence into account. The set of subsequences that support a temporal pattern share the same order of events and similar temporal information. Table 7.2 gives an example of temporal sequence dataset. Each event in sequences is aligned with temporal information. The basic idea to mine temporal patterns involves two

steps: 1) extracting frequent sequential patterns; 2) identifying clusters in each sequential pattern based on the temporal information of subsequences. The first step usually utilizes sequential pattern mining algorithms such as PrefixSpan [50].

In this section, the algorithm PrefixSpan [50] is briefly explained.

PrefixSpan

PrefixSpan (Prefix-projected Sequential pattern mining) [50] is the most famous frequent sequential pattern mining algorithm. It is used as the backbone in many other advanced sequential pattern mining or temporal pattern mining algorithms. The main idea of PrefixSpan is to project the sequence dataset recursively onto prefix subsequences. This idea is very similar to divide-and-conquer based algorithms in itemset mining.

Using the dataset in Table 7.1 as an example and let the minimum support $minSup = 3$. The first step of PrefixSpan is removing all infrequent items from the dataset. Item E is removed from the dataset in the first step since it only occurred for two times in the dataset. Then the dataset is projected onto each remaining item in the dataset, leading to a set of projected datasets, respectively. The item is the prefix sequence, while the part after the item in each sequence is the postfix in the projected dataset. For instance, when the original dataset is projected onto the item A, the projected dataset is $\{BCAD, BDA, \emptyset, BDC, C, C\}$. The size of the projected dataset is 6. Therefore, its prefix A is a frequent sequential pattern. The dataset projection process can be applied recursively. For example, the projected dataset of item A and be projected onto the item C. The new projected dataset is $\{AD, \emptyset, \emptyset, \emptyset\}$ and its prefix AC is a frequent sequential pattern. Sequential patterns are identified by repeating the projection process until no projected dataset can be generated.

The pseudo-code of PrefixSpan is described in Algorithm 7.1. In this brief description, some cases, such as multiple events that happened simultaneously, are not discussed. Nevertheless, the overall structure of PrefixSpan, i.e., recursive dataset projection, is clear. Other advanced sequential pattern

mining algorithms, such as interval-based sequential pattern mining [102] or temporal pattern mining algorithms [39] are mostly developed upon PrefixSpan.

Algorithm 7.1: PrefixSpan

```
   Input: Sequence Dataset S, minSup
   Output: The set of frequent sequential patterns F
1  F ← Projection(S, ∅, minSup);
2  return F ;
3  Function Projection(Database S', Prefix pre, minSup):
4  |  Scan S' once, find the set of frequent items b;
5  |  For each frequent item b, append it to pre to form a sequential
   |    pattern X, and add X to F;
6  |  For each X, construct X-projected dataset S'|X, and call
   |    Projection(S'|X, X, minSup);
7  end
```

The remainder of this part is organized as follows: Chapter 8 introduces some preliminaries needed for proposing our approach for temporal pattern mining in streams in Chapter 9. Chapter 10 extends the idea to handle the interval-based temporal pattern mining tasks on streams, with Section 10.5 shows the results of our extensive experimental evaluation and Section 10.6 concludes this work with an outlook. Part of the contents presented in this part has been published in [70].

Chapter 8

Temporal Patterns in Streams

Formally speaking, a *stream* S is an infinite sequence of interval events which evolves continuously, as shown in Figure 8.1. People can only retrieve the current value from the stream. Storing the whole stream into the main memory is unrealistic.

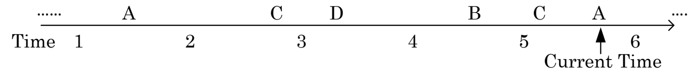


Figure 8.1: Small example of point-based events stream.

8.1 Temporal Event and Temporal Sequence

Given an event label set Σ, an point-based *event* E has two attributes: (l, t), where $l \in \Sigma$ is the label or the type of the event. $t \in \mathbb{R}$ is the time or the temporal information aligned with the event.

A *sequence* S is defined as a list of events. In order to handle the noise exists in temporal information as mentioned above, events are ordered based on the alphabet of their labels rather than the temporal information as is used in other approaches. Formally speaking, $S = \{E_1, E_2, \ldots, E_n\}$, where $\forall i \in \{1, \ldots, n-1\}, E_i.l \leq E_{i+1}.l$. Sequence contains n events is called n-sequence for short.

Events with the same label will be ordered based on their temporal information. Besides, each event will preserve their temporal information. For

example, the sequence appeared in the snapshot of a stream in Figure 8.1 is denoted as

$$S = \{(A, 1.5), (A, 5.8), (B, 4.6), (C, 2.9), (C, 5.1), (D, 3.3)\}.$$

A *subsequence* S' is a sublist of sequence S, denoted as $S' \subseteq S$. For example, $S' = \{(A, 1.5), (B, 4.6), (C, 2.9)\}$ is a subsequence of the sequence above.

Furthermore, the concept of *sequence string* is introduced for simplicity. A *string* of a sequence is the label list, denoted as $S.str = \langle l_1, l_2, \ldots, l_n \rangle$, where $l_i = E_i.l$. For example, the string of the sequence in Figure 8.1 is $\langle A, A, B, C, C, D \rangle$. In the rest of this part, $\{\cdot\}$ and $\langle \cdot \rangle$ will be used to represent sequence and string respectively.

Given string $str' \subseteq S.str$, the corresponding subsequence is denoted as $S' = S(str')$. When multiple events in S share the same label, more recent events are chosen first. For example, let $str' = \langle A, B, C \rangle$, the corresponding subsequence is $S' = S(str') = \{(A, 5.8), (B, 4.6), (C, 5.1)\}$.

8.2 Similarity Measures and Temporal Pattern

Commonly speaking, sequential patterns or temporal patterns describe the relative relationship between events. Thus, the similarity between sequences should only depend on the relative difference within sequences. The absolute happening time should not affect the distance value. For example, the event sequence *drinking coffee then read a book*, happened in the morning or the afternoon should be considered as the same and support the same pattern. This invariant in absolute happening time is implicitly provided if the gap information is employed, while in consequence the noise problem mentioned above could not be avoided. In contrast, we let each event store their temporal information and guarantee the invariant property in the distance function.

We first consider the similarity between two events. The distance between events with different labels is set to infinity. Furthermore, events with the same label are considered as equivalent despite the difference in their happening time. Under such settings, we define the distance between two events E_1 and E_2 as follows:

$$d(E_1, E_2) = \begin{cases} \infty & , l_1 \neq l_2 \\ \min_{h \in \mathbb{R}} |t_1 - (t_2 - h)| = 0 & , l_1 = l_2 \end{cases} \tag{8.1}$$

l, t are the corresponding properties of each event. When $h = t_2 - t_1$, equation 8.1 is minimized with value 0, i.e., h is an argument of the distance function that helps to eliminate the difference in happening time between events, which will be used later in the distance function of sequence.

Based on the distance function given above, the distance between sequences with different strings is also set to infinity. When sequences contain the same string, the distance is defined as the minimum sum of distances between events.

$$d(S_1, S_2) = \begin{cases} \infty & , S_1.str \neq S_2.str \\ \frac{1}{n} \min_{h, f(\cdot)} \sum_{i=1}^{n} d(E_{1i}, f(E_{1i})) & , S_1.str = S_2.str \end{cases} \tag{8.2}$$

where n is the sequence length, $h \in \mathbb{R}$, $E_{1i} \in S_1, E_{2i} \in S_2$. $f : E_{1i} \rightarrow E_{2j}$ is a bijective mapping of events in one sequence to that in another sequence.

An event will only be mapped to another event with the same label. Otherwise, the distance is infinity. When multiple events share the same label, the mapping is determined by the temporal order. Thus, for point-based temporal sequences with the same string, the best mapping between events is: $f(E_{1i}) = E_{2i}$.

The optimized value of h is independent from the mapping function $f(\cdot)$. Assuming $f(\cdot)$ is given, equation 8.2 can be written as:

$$d(S_1, S_2) = \frac{1}{n} \min_{h} \sum_{i=1}^{n} |E_{1i}.t - (f(E_{1i}).t - h)|$$

Let $\frac{\partial d}{\partial h} = 0$, we have:

$$
\begin{aligned}
h_{opt} &= \frac{1}{n}\sum_{i=1}^{n}(f(E_{1i}).t - E_{1i}.t) = \frac{1}{n}\sum_{i=1}^{n} f(E_{1i}).t - \frac{1}{n}\sum_{i=1}^{n} E_{1i}.t \\
&= \frac{1}{n}\sum_{i=1}^{n} E_{2i}.t - \frac{1}{n}\sum_{i=1}^{n} E_{1i}.t
\end{aligned}
$$

which means that the value of the distance function is computed by moving sequences horizontally until their mean points coincide, as shown in Figure 8.2.

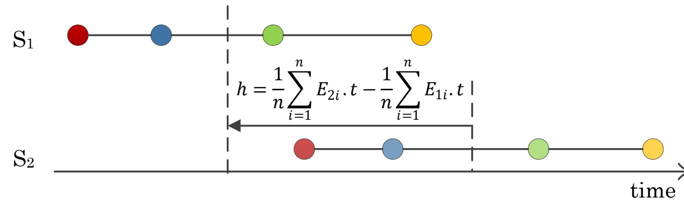


Figure 8.2: Similarity between temporal sequences is computed by moving sequence first. Points represent events, colors represent event labels.

A *temporal pattern* is an abstract sequence of a group of temporal sequences which is determined by the clustering results of temporal information, as will be described later. Sequences in the group *support* the pattern. Patterns with support larger or equal to some user defined *minimum support* are called *frequent patterns*.

Based on the distance given above, sequences been clustered together must share the same string. An average sequence is used as the temporal pattern which can be computed in 2 steps: move all sequences horizontally to align the mean point; calculate the average temporal value of each corresponding events, i.e., for each event E_{Pi} in a pattern P which is supported by m sequences with length n, $E_{Pi}.t = \frac{1}{m}\sum_{j=1}^{m} E_{ji}.t$.

For example, given the following sequences belonging to the same cluster:

$$S_1 = \{(A, 1.5), (B, 4.6), (C, 2.9)\}$$
$$S_2 = \{(A, 2.7), (B, 6), (C, 7.2)\}$$
$$S_3 = \{(A, 2), (B, 3.3), (C, 5.3)\}$$

the pattern P is $\{(A, 1.66), (B, 4.22), (C, 4.72)\}$. Furthermore, we shift the first event to time 0 as the standard pattern representation, i.e., the pattern above can also be written as $\{(A, 0), (B, 2.57), (C, 3.07)\}$.

Different similarity measures could also be applied depending on the application scenario. For example, the absolute happening time or the invariance of sequence scale might be valuable. To fulfill those different requirements, other distance functions could be employed as long as an average sequence can be defined.

Chapter 9

Incremental Temporal Pattern Mining

The sliding window model is employed as shown in Figure 9.1. Events covered by current sliding window form the *current sequence*.

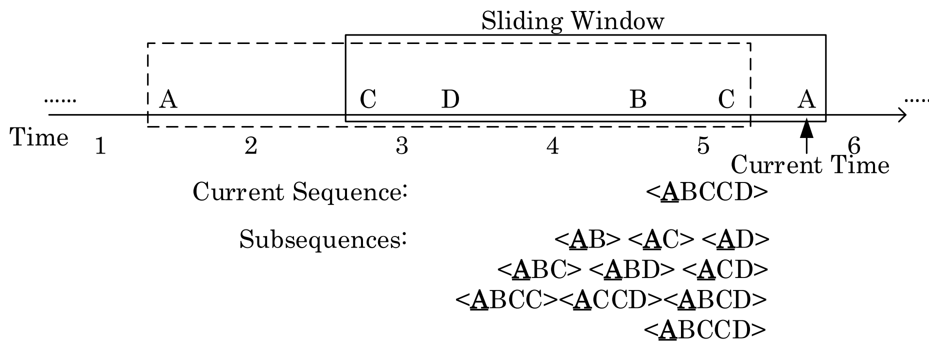


Figure 9.1: Current sequence is covered by the sliding window. All candidate subsequences for pattern updating contain the newly evolved event (**A**).

A subsequence which contains the latest event is a *candidate sequence*. It will be processed and inserted into the closest *micro-cluster* which represents a group of sequences.

The online-offline model acts as the primary structure in our approach, as illustrated in Algorithm 9.1. With the similarity measure defined above, a feature vector will be maintained for each micro-cluster incrementally. Outdated micro-clusters will be pruned based on an estimated support. `Merging`,

`Pruning` and `GeneratePatterns` steps are adopted from the stream clustering algorithm DenStream [19].

In this chapter, we will first introduce the adopted stream clustering process followed by the definition of the estimated support. An indexing system based on prefix tree is presented for efficient micro-cluster access. Part of the materials presented in this chapter have been published in [70].

Algorithm 9.1: Temporal Interval Pattern Mining

Input: Stream $\mathbb{S}$, minimum support *minSup*, decaying factor α

```
1  while S is not end do
2  |   I ← S.NextInstance
3  |   S ← CurrentSequence
   |   /* Online pattern update                              */
4  |   foreach Candidate sequence S' in S do
5  |   |   Merging(S')
6  |   end
7  |   if It is time for pruning then
8  |   |   Pruning()
9  |   end
   |   /* Offline pattern generation                         */
10 |   if A clustering request arrives then
11 |   |   GeneratePatterns()
12 |   end
13 |   Move sliding window
14 end
```

9.1 Stream Clustering

In this section, we will first introduce the adopted stream clustering process followed by the definition of the estimated support. An indexing system based on prefix tree is presented for efficient micro-cluster access.In static database case, related literature such as [39, 46] proposed to extract temporal patterns by clustering sequences. Following the same idea here, we obtain temporal patterns by using a stream clustering approach, which is mainly adopted from the DenStream [19] algorithm. Each candidate sequence S

with n events can be seen as a n dimensional point with a decaying weight w:

$$w(t) = 2^{-\alpha(t-t_0)} \tag{9.1}$$

where α is the decaying factor, t is the current time and t_0 is the appearing time of the data point.

A list of micro-clusters is maintained, each of which represents a *temporary pattern* supported by a group of sequences $\mathbb{S}$ with the same string, defined as $MC = (n, t_c, W, WLS, WSS, str)$, where n is the number of points, t_c is the last updating time of the micro-cluster, str is the string shared by all sequences in the micro-cluster.

$$W = \sum_{i=1}^{n} w_i(t),\ WLS = \sum_{i=1}^{n} w_i(t) \cdot S_i,\ WSS = \sum_{i=1}^{n} w_i(t) \cdot S_i^2,$$

$S_i \in \mathbb{S}$, S_i^2 is the element-wise square of S_i, $w_i(t)$ is the decaying weight corresponding to each sequence, W is the total weight of the micro-cluster. As defined in the similarity measures, all sequences are aligned to a common mean point for the computation of WLS and WSS.

In order to handle outlier, all micro-clusters will be distinguished between *potential core micro-cluster* (p-micro-cluster) and *outlier micro-cluster* (o-micro-cluster) based on their *estimated support* (cf. Section 9.2). The center of a micro-cluster is $c = \frac{WLS}{W}$, which defines the average sequence. Candidate sequence will be first assigned to the closest p-micro-cluster with respect to the distance from the center, and the new radius will be tested:

$$r = \sqrt{\frac{WSS}{W} - \left(\frac{WLS}{W}\right)^2} \leq \epsilon$$

If the new radius is smaller than a threshold ϵ, the candidate sequence will be added to the p-micro-cluster. Otherwise, an o-micro-cluster will be updated or created with the single candidate sequence. As a consequence, each micro-cluster contains sequences that are similar to each other, and the center can be seen as the temporary pattern. Each micro-cluster will be scanned and pruned periodically based on the estimated support.

It is worth to note that all elements above could be maintained incrementally. For example, let t_{i0} be the appearing time of sequence S_i in a micro-cluster c and t_c be the appearing time of the last sequence. Total weight at time t_c is

$$W_{t_c} = \sum_{i=1}^{n} w_i(t_c) = \sum_{i=1}^{n} 2^{-\alpha(t_c - t_{i0})}$$

Given the current time t, if no new sequence is merged,

$$\begin{aligned} W_t &= \sum_{i=1}^{n} w_i(t) = \sum_{i=1}^{n} 2^{-\alpha(t-t_{i0})} = \sum_{i=1}^{n} 2^{-\alpha(t-t_c)-\alpha(tc-t_{i0})} \\ &= 2^{-\alpha(t-t_c)} \cdot \sum_{i=1}^{n} 2^{-\alpha(t_c-t_{i0})} = 2^{-\alpha(t-t_c)} \cdot W_{t_c} \end{aligned}$$

If a new sequence is merged,

$$W_t = 2^{-\alpha(t-t_c)} \cdot W_{t_c} + 1$$

Thus, attributes of micro-cluster can be updated incrementally by decaying factor $2^{\alpha(t-tc)}$.

When the user requires clustering results of the stream, all potential core micro-clusters are clustered using a modified version of DBSCAN [35], temporary patterns represented by different micro-clusters might be joined and support the same final pattern. Based on the clustering results of micro-clusters, temporal pattern P is computed as average sequence:

$$P = \frac{\sum c_i.WLS}{\sum c_i.W}$$

where all micro-clusters c_i are in the same cluster.

9.2 Support Estimation

Some criteria are necessary for pruning during stream mining process to keep the size of information stored small. DenStream prunes micro-clusters based

on the corresponding weight. SEQ [47] prunes candidate patterns if the count value is low. As discussed in Section 7.2, these criteria are not suitable for a stream with varying speed.

Support is a better criterion in the stream scenario. However, conventional definition of support of a pattern in a static database is the percentage value of sequences which contain the pattern. It is not applicable to the mining task discussed here since no static database is created and the total number of sequences is unknown.

We adopted the support definition from [53]:

$$supp(P) = \frac{C(P)}{N} \leq minSup$$

where $C(P)$ is the count of pattern P, and N is the total number of events evolved in the stream. $minSup$ is the minimum support threshold given by the user.

Taken decaying weights into consideration, the support definition above can also be written as:

$$supp(P) = supp(MC) = \frac{MC.W}{W} \tag{9.2}$$

where MC is the micro-cluster that represents the pattern P, $MC.W$ is the corresponding weight. W is the overall weight of the data stream which depends on the speed of stream v: $W = \frac{v}{1-2^{-\alpha}}$ [19]. The corresponding support value $supp(P)$ divides micro-clusters into two categories, p-MC and o-MC. The pruning of micro-clusters is also determined by the value of support.

A p-micro-clusters will be pruned if its support is smaller than the minimum support $minSup$. Pruning is performed every T_p unit of time periodically. The value of T_p is estimated based on the minimum time span for a p-micro-cluster fading into an outlier [19]. Let the current support of such p-micro-cluster c_p is $supp_{old}(c_p) = \frac{c_p.W_{old}}{W_{old}} = minSup$, the new support after

T_p units of time should be: $supp_{new}(c_p) = \frac{c_p.W_{new}}{W_{new}} = minSup$,

$$\frac{2^{-\alpha T_p} MC.W_{old} + 1}{W_{new}} = minSup$$

$$\begin{aligned} T_p &= \frac{1}{\alpha} \log \frac{c_p.W_{old}}{minSup \cdot W_{new} - 1} \\ &= \frac{1}{\alpha} \log \frac{minSup \cdot W_{old}}{minSup \cdot W_{new} - 1} \end{aligned}$$

However, the future speed of stream is unknown, i.e., W_{new} is unknown. We assume that the speed will not change within one pruning period, i.e., $W_{new} = W_{old}$. Thus, T_p can be estimated based on the current total weight of stream W_{old}:

$$T_p = \frac{1}{\alpha} \log \frac{minSup \cdot W_{old}}{minSup \cdot W_{old} - 1}.$$

T_p and W_{old} will then be updated until the next pruning step.

The pruning threshold for o-micro-cluster is determined by the function $\xi(\Delta t)$, where Δt is the existing time of the o-micro-cluster up to now. If an o-micro-cluster c_o tends to become a p-micro-cluster, then:

$$\xi(\Delta t) + 2^{-\alpha \Delta t}(minSup \cdot W_{old} - 1) = minSup \cdot W_{new}$$

Again, let $W_{new} = W_{old}$, we have:

$$\xi(\Delta t) = \frac{2^{-\alpha(\Delta t + T_p)} - 1}{2^{-\alpha T_p} - 1}$$

Detailed proofs of the correctness of these components above are the same as in [19]. Having all settings defined above, the `Merging` and `Pruning` processes are described in Algorithms 9.2 and 9.3.

Algorithm 9.2: Merging

Input: Candidate sequence S'

1 Assign S' to the nearest potential core micro-cluster c_p
2 **if** *new radius of* c_p $r_p \leq \epsilon$ **then**
3 | Merge S' to c_p
4 | Update attributes of c_p
5 **else**
6 | Assign S' to the nearest outlier micro-cluster c_o
7 | **if** *new radius of* c_o $r_o \leq \epsilon$ **then**
8 | | Merge S' to c_o
9 | | Update attributes of c_p
10 | | **if** *new support of* c_o $supp(c_o) > minSup$ **then**
11 | | | Convert c_o to a potential core micro-cluster *p-MC*
12 | | **end**
13 | **else**
14 | | Create a new outlier micro-cluster by S'
15 | **end**
16 **end**

Algorithm 9.3: Pruning

1 **foreach** *potential core micro-cluster* c_p **do**
2 | **if** $supp(c_p) < minSup$ **then**
3 | | Delete c_p
4 | **end**
5 **end**
6 **foreach** *outlier micro-cluster* c_o **do**
7 | $\xi(\Delta t) = \frac{2^{-\alpha(\Delta t + T_p)} - 1}{2^{-\alpha T_p} - 1}$
8 | **if** $supp(c_o) < \xi(\Delta t)$ **then**
9 | | Delete c_o;
10 | **end**
11 **end**
`/* Update the total weight of stream */`
12 $W \leftarrow$Current stream total weight
`/* Compute the next pruning period */`
13 $T_p = \frac{1}{\alpha} \log \frac{minSup \cdot W}{minSup \cdot W - 1}$

9.3 Fast Micro-cluster Updating

For each candidate sequence, searching through all existing micro-clusters to find the nearest one for insertion (Algorithm 9.2, step 1,6) is inefficient. Since the distance between sequences with different strings is infinity, all sequences in the same micro-cluster must share the same string. A prefix tree is introduced to index micro-clusters based on their strings for fast candidate sequence insertion and micro-cluster retrieving, as shown in Figure 9.2.

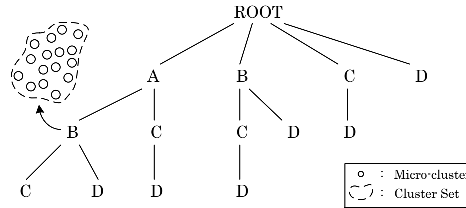


Figure 9.2: Prefix tree for micro-cluster update.

Each node of the prefix tree is marked with one event type while the root node is null. One concatenation of labels from the root to any node forms a sequence string. Thus, each node corresponds to a set of micro-clusters $CS = \{c_1, c_2, \ldots, c_n\}$ in which $\forall i \in \{1, \ldots, n\} : c_i.str = CS.str$. Micro-clusters with the same string of a given candidate sequence can be retrieved efficiently by traversing through the prefix tree.

Moreover, given sliding window size w, the total number of candidate sequences for insertion is 2^{w-1}, as the latest event must be included. Traversing all candidate sequences in Algorithm 9.1 (step 4) is time-consuming. In order to reduce the search space, the Apriori property of sequences is employed, i.e., if sequence S' can not be merged to any existing micro-clusters, then its super-sequence $S \supset S'$ has a high degree of outlierness. In this work, such kind of super-sequence will be dropped to reduce the search space of pattern updating.

We first identify the high degree outlierness candidate sequence S' by categorizing as follows:

1. The string of S' matches an existing path in the prefix tree, i.e., $\exists micro-clusterc: S'.str = MC.str$.

2. The string of S' matches a path in the prefix tree by adding one extra node, i.e., $\exists micro-clusterc$: $c.str \subset S'.str \wedge |S'.str| = |c.str| + 1$.

3. The string of S' matches a path in the prefix tree by adding more than one extra node, i.e., $\exists micro-clusterc$: $c.str \subset S'.str \wedge |S'.str| \leq |c.str| + 2$.

A candidate sequence belonging to the first category will be merged into the corresponding micro-cluster stored in the prefix tree. Candidate sequences in the second type have no corresponding node in current prefix tree. A new node will be created, and sequences in this category will be considered as supporting an unusual pattern at the moment. Candidate sequences in the third group will be considered as abnormal sequences since additional nodes need to be created and attached to new nodes set up by sequences in the second type. These candidate sequences will be directly dropped, and the search space is reduced. For example, given newly evolved event D and current sequence $S = \langle D, E, F, G \rangle$, subsequences such as $\langle D, E \rangle$, $\langle D, F \rangle$ and $\langle D, G \rangle$ will be considered and new micro-cluster will be added. However, subsequence such as $\langle D, E, F \rangle$ is not eligible for insertion since $|\langle D, E, F \rangle| = |\langle D \rangle| + 2$.

Furthermore, the monotonicity property of distance function could also be used to reduce the search space. If a sequence in the first category above is far away from all existing micro-clusters, then its super-sequence will be dropped. However, it should be noted that the similarity measure presented in Section 8.2 violates the monotonicity property, i.e., given $S_1.str = S_2.str, S'_1.str = S'_2.str, S'_1 \subset S_1, S'_2 \subset S_2$, $d(S'_1, S'_2) > \epsilon \nRightarrow d(S_1, S_2) > \epsilon$, as the distance function is normalized with respect to the length of the sequence. Nevertheless, this pruning strategy still works well in practice.

In practice, we will first detect the string of all valid candidate sequences in the current prefix tree based on the discussion above. Candidate sequences will then be merged into corresponding micro-clusters stored in the prefix tree. Given current sequence covered by the sliding window S_c and the latest event E_c, new sequence instances are added (Algorithm 9.1, step 4) more efficiently as shown in Algorithm 9.4. If a candidate sequence brings a new

outlier micro-cluster, it is considered as an outlier, and its super-sequences will not be merged due to the monotonicity.

The subroutine `TraverseNode` detects all valid string with respect to the current sequence S_c from the prefix tree recursively. Events are traversed based on the order given in the definition of sequence in Section 8. Parameter `Skipped Events` is a list which tracks all events been skipped due to the order of traversal. These skipped events will be used to create new node only if the latest event is covered by current prefix tree (Step 5-7). Otherwise, the new node can only be set up by the latest event.

Algorithm 9.4: MicroClusterUpdate

```
   Input: Current Sequence S_c, Latest Event E_c, Prefix Tree T
1  Current Node n ← T.root
   /* Detect valid strings without changing prefix tree    */
2  Candidate string list LIST ← TraverseNode(S_c, E_c, n, ∅)
   /* Add the subsequence of each valid string             */
3  OUTLIER ← ∅
4  foreach str ∈ LIST do
5      if ∃s ∈ OUTLIER : s ⊂ str then
6          Continue
7      end
8      if ∃node ∈ T : node.str = str then
9          Merge S_c(str) to node
10         if New outlier micro-cluster created then
11             OUTLIER ← str ∪ OUTLIER
12         end
13     else
14         node' ← T.newNode(str)
15         Merge S_c(str) to node'
16     end
17 end
```

Algorithm 9.5: TraverseNode

```
   Input: Sequence S, Latest Event E_c, Current Node n, Skipped
          Events SE
   Output: Valid String List LIST
 1 LIST ← ∅
 2 if E_c ≠ null then
 3 |   LIST ← ⟨n.str ∪ {E_c.l}⟩ ∪ LIST
 4 else
 5 |   foreach Event E_j ∈ SE do
 6 |   |   LIST ← ⟨n.str ∪ {E_j.l}⟩ ∪ LIST
 7 |   end
 8 end
 9 foreach Event E_i ∈ S do
10 |   if ∃n' ∈ n.children : n'.label = E_i.l then
   |   |   /* Select events after E_i                          */
11 |   |   Sequence S' ← {E' ∈ S|E' > E_i}
12 |   |   if E_i = E_c then
   |   |   |   /* The latest event is covered and set to null
   |   |   |       */
13 |   |   |   LIST ← TraverseNode(S', null, n', SE) ∪ LIST
14 |   |   else
15 |   |   |   LIST ← TraverseNode(S', E_c, n', SE) ∪ LIST
16 |   |   end
17 |   else
18 |   |   SE ← {E_i} ∪ SE
19 |   end
20 end
21 return LIST
```

Chapter 10

Mining Interval-based Event Streams

Many events in the real world persist for some short or long period, so it is quite natural and more appropriate to reflect the temporal extension in the model and to represent events in intervals, as shown in Figure 10.1. Despite the demand, to the best of our knowledge, in stream environment, the interval-based temporal pattern mining problem has not been investigated.

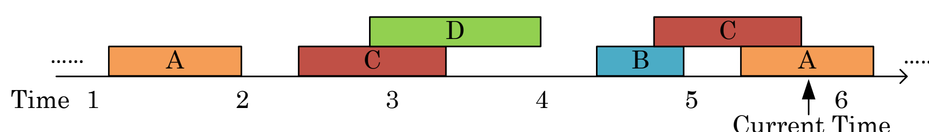


Figure 10.1: Stream of interval-based events, each event is represented by a rectangle.

In the static database environment, existing point-based approaches utilizing the gap information, such as [39], can be extended to handle interval-based data. Each interval-based event will be processed as two individual events. Additional processes are necessary to guarantee that interval-based patterns generated are complete, i.e., each starting point has a corresponding ending point. Some other approaches only focus on the interval-based data with novel representations. Our approach, on the other hand, can be extended to handle interval-based temporal data smoothly and efficiently by

just introducing one more numerical attribute to the event point.

In this section, we will first give a short introduction to the interval-based temporal pattern mining in the static database. Then, our approach proposed above will be extended to handle interval-based temporal event streams.

10.1 Interval-based Temporal Pattern Mining in Static Database

Relationships among interval-based events are more complicated than the point-based case. Different temporal operators, such as Allen's relations [5] which describe interval-based sequences using 13 different relationships, were proposed. Approaches such as [102] investigated the interval-based SPM. Most of them extended the point-based algorithm PrefixSpan with additional completeness checking and pruning steps.

Handling interval-based events with temporal information at the same time is nontrivial since not only the gap between events but also the duration of each event need to be considered. Approaches such as [25, 27] generate sequential patterns first. Temporal information was attached to each frequent pattern. To integrate the SPM strategy with the clustering techniques, novel representations were introduced in recent years. [45] proposed to represent an interval-based temporal sequence as a hypercube. Each interval event is treated as an edge of the cube. Temporal patterns were generated based on a breadth-first search and EM algorithm. [82] annotated the point-based representation with temporal information. Subspace clustering is conducted in high-dimensional space repeatedly in a PrefixSpan based search. [46] was proposed to mine item sets with attached interval-based temporal bounds. By introducing temporal information to SPM, the relationship between events can be expressed more in detail. All approaches mentioned above declare that a better mining quality could be achieved compare with point-based SPM.

In algorithms discussed above, intervals are ordered based on their tem-

poral information, such as the starting time or ending time. Thus, the same problem mentioned in the point-based case when considering noisy temporal database also exists in the interval-based case, as shown Figure 10.2.

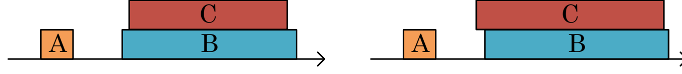


Figure 10.2: Two sequences with the same event types and similar temporal information, but support different patterns due to some natural noise.

10.2 Interval-based Temporal Event and Sequence

Given an event label set Σ, an interval-based *event* E has four attributes: l, s, e, d, where $l \in \Sigma$ is the label. $s, e \in \mathbb{R}$ are the starting and ending time respectively. Duration $d = e - s$ is a dependent variable. Each interval event can be uniquely defined by any two elements in s, e and d. Thus, an interval event can be represented as a labeled point on the 2D plane as proposed in [85]. In this work, starting time s and duration d are used as the X and Y axises. The relationship between interval-based events on such 2D representation can be converted to the widely used Allen's relations by comparing their relative positions, as shown in Figure 10.3.

Similar to the point-based case, a *interval-based sequence* S is defined as a list of interval-based events. Events are ordered based on the alphabet of their labels, i.e., $S = \{E_1, E_2, \ldots, E_n\}, \forall i \in \{1, \ldots, n-1\}, E_i.l \leq E_{i+1}.l$. Events with the same label will be ordered based on their starting time. For example, the sequence appeared in the snapshot of a stream in Figure 10.3 is:

$$\begin{aligned} S =&\{(A, 1.1, 2.0), (A, 5.3, 6.3), (B, 4.4, 4.9), (C, 2.3, 3.3), \\ &(C, 4.7, 5.8), (D, 2.8, 4.0)\} \end{aligned}$$

Other definitions such as subsequence and string of sequence are the same as in the point-based case.

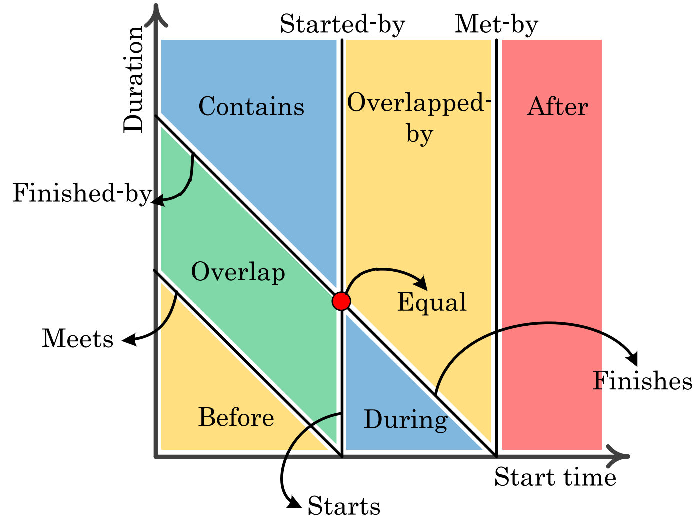


Figure 10.3: 13 Allen's relations and the 2D representation.

10.3 Similarity Measures

The distance between interval-based events is defined similarly as the point-based case. Events with different labels differ completely. As shown in Figure 10.3, relationships between interval-based events are complicated and affected by both starting time and duration. Euclidean distance is employed to describe the distance between events with the same label.

$$d(E_1, E_2) = \begin{cases} \infty & , l_1 \neq l_2 \\ \min_h \sqrt{(s_1 - (s_2 - h))^2 + (d_1 - d_2)^2} & , l_1 = l_2 \end{cases} \tag{10.1}$$

where s, d are the corresponding properties of each event. The parameter $h \in \mathbb{R}$ is also employed here but only applied to the starting time. When $h = s_2 - s_1$, equation 10.1 is minimized with value $|d_1 - d_2|$, as shown in Figure 10.4a. That is, the difference between interval-based events with the same label only depends on the difference in duration value. Absolute happening time, i.e., the starting time, did not affect the distance value.

Distance between interval-based sequences follows the same definition used in point-based case:

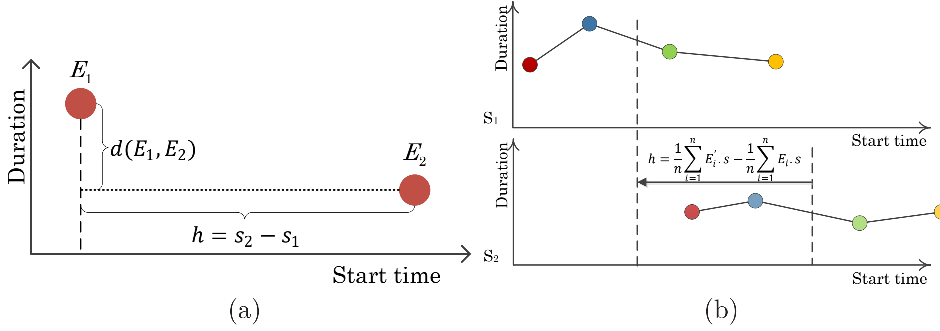


Figure 10.4: Similarity between interval events (a) and interval sequences (b).

$$d(S_1, S_2) = \begin{cases} \infty & , S_1.str \neq S_2.str \\ \frac{1}{n} \min\limits_{h, f(\cdot)} \sum_{i=1}^{n} d(E_{1i}, f(E_{1i})) & , S_1.str = S_2.str \end{cases} \tag{10.2}$$

where n is the sequence length, $h \in \mathbb{R}, E_{1i} \in S_1, E_{2i} \in S_2$. $f : E_{1i} \rightarrow E_{2i}$ is the bijective mapping of events in one sequence to that in another sequence. It can be shown that the optimized value of h is independent from the mapping function $f(\cdot)$, the same as the point-based case. Assuming $f(\cdot)$ is given, equation 10.2 can be written as:

$$d(S_1, S_2) = \frac{1}{n} \min_h \sum_{i=1}^{n} \sqrt{(E_{1i}.s - (f(E_{1i}).s - h))^2 + (E_{1i}.d - f(E_{1i}).d)^2}$$

Let $\frac{\partial d}{\partial h} = 0$, we have:

$$\begin{aligned} h_{opt} &= \frac{1}{n} \sum_{i=1}^{n} (f(E_{1i}).s - E_{1i}.s) = \frac{1}{n} \sum_{i=1}^{n} f(E_{1i}).s - \frac{1}{n} \sum_{i=1}^{n} E_{1i}.s \\ &= \frac{1}{n} \sum_{i=1}^{n} E_{2i}.s - \frac{1}{n} \sum_{i=1}^{n} E_{1i}.s \end{aligned}$$

which means that the value of distance is computed by moving sequences horizontally until they have a common average value of starting time, as

shown in Figure 10.4b.

The mapping function $f(\cdot)$ keeps the same as before. When multiple events with the same label exist, the mapping between those events is given in the order of their starting time.

The interval-based temporal pattern is also computed as the average sequence. Sequences support the same pattern are aligned to a common average starting time. The average value of starting time and duration corresponding to each event is computed, as shown in Figure 10.5.

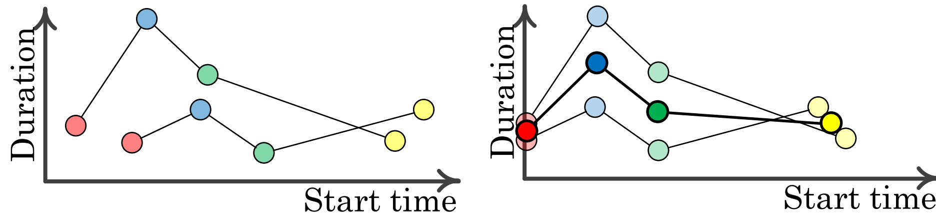


Figure 10.5: Visualization of interval-based temporal pattern as an average sequence.

Other similarity measures could also be applied to fulfill different application requirements. For example, the similarity measure IBSM [64] proposed recently performs full sequence matching and the sequence scale invariant could be achieved.

10.4 Interval-based Event Stream

As shown in Figure 10.6, an interval-based temporal stream can be represented as a stream of starting and ending points. The starting point will be stored in a buffer first until the corresponding ending point appears. A new interval event is then created based on the pair of starting and ending points. For a new event just arrived, the relevant information of the starting point (l, s) is added to the starting point buffer. When an ending point is coming, the corresponding entry will be retrieved from the starting point buffer and a new event (l, d, e) is generated. Old information will be removed from the buffer to keep the size small.

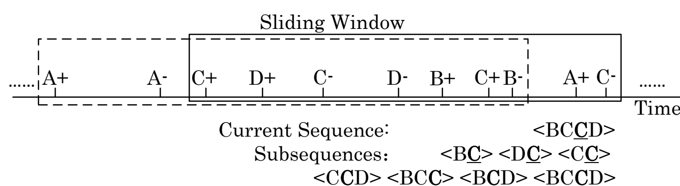


Figure 10.6: Sliding window on the interval-based event stream shown in Figure 10.1.

As illustrated above, our model can be easily extended from point-based data to interval-based data. The incremental mining process on streams can be applied without any modification. This advantage is given by the fact that each event is considered as a 2D point with a label and numerical attributes.

On the other hand, alternative representation which makes use of the gap information presents each interval-based event as two points, one starting point, and one ending point. Point-based approaches utilizing those representations need to be revised.

10.5 Experiment Results

In this section, we present the experimental results conducted on our temporal pattern mining algorithm on the stream data. To the best of our knowledge, there is no pre-existing approach solve the same task as we do. Thus, the purpose of this comparison is to show that our approach, with its richer information, can still provide a comparable performance. Quality and performance of our approach are compared with that of SS-BE [81], a batch based method, and SEQ [47], which employs sliding window on a single stream without considering temporal information. Furthermore, we also studied the property of our approach when extending to interval-based event streams. All algorithms presented here are implemented in JAVA.

[4] proposed a well-known synthetic data generator for point-based sequential pattern mining. Our synthetic stream is generated in the same strategy with necessary modifications, such as timing information and noise for temporal events. Sequences are connected end to end to generate the stream. Parameter settings for stream clustering follow the suggestion in

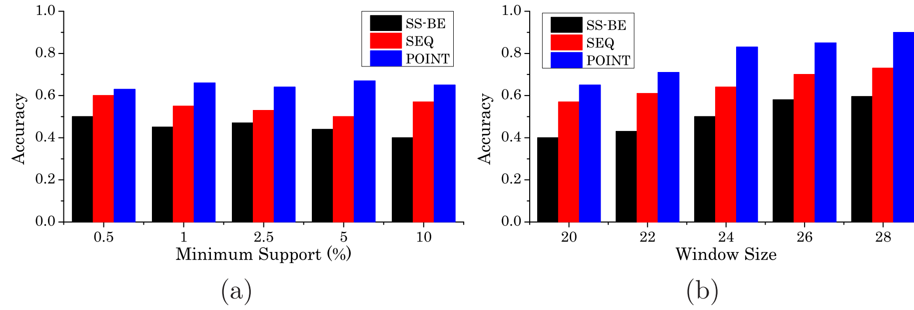


Figure 10.7: Accuracy on synthetic stream with respect to the sliding window size and the minimum support.

[19]. Unless otherwise mentioned, the default values for other parameters are: Stream length $L = 200000$, Minimum support $minSup = 10\%$ and Window Size $w = 20$. To measure the SS-BE algorithm, we cut the stream into sequences with length equal to the window size. Every 200 sequences form a batch. The same minimum support is also used for SS-BE approach. The corresponding cardinality threshold used by SEQ is roughly computed as $minSup \cdot w$. POINT and INTERVAL refer to our point-based approach and interval-based approach respectively.

We first compared the quality of patterns generated by those algorithms on synthetic streams. The accuracy value is defined as the rate of correctly identified patterns with respect to the total number of patterns. In synthetic streams, the ground truth, i.e., the total number of patterns, is known. Patterns found by SS-BE and SEQ will be considered as correct if the order of events is correct. As shown in Figure 10.7, our approach can achieve a better accuracy value under different minimum support and window size settings. Two major reasons lead to the low accuracy value of the other two approaches. Firstly, patterns such as $\{(A, 0), (B, 1)\}$ and $\{(A, 0)(B, 20)\}$ are considered as the same. Secondly, patterns such as A *equal* B will not be detected due to the noise. The accuracy value of the batch based method, SS-BE, is even lower since many patterns across sequences and batches are lost.

Figure 10.8 illustrate the comparison of processing time on the synthetic dataset with respect to the stream length, the minimum support, and the

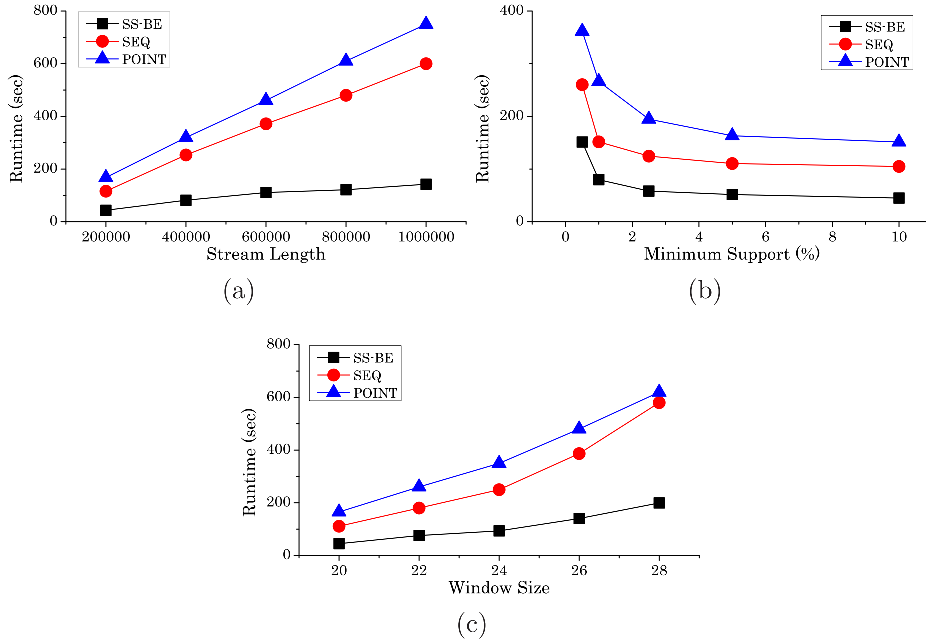


Figure 10.8: Processing time on synthetic stream with respect to the stream length, the minimum support and the sliding window size.

sliding window size. The fastest approach is obviously the SS-BE. By cutting streams into batches and applying the PrefixSpan algorithm, SS-BE is very efficient. Thanks to the efficient micro-cluster updating schema and the prefix tree, our approach can achieve a similar runtime as SEQ despite the effort taken to consider temporal information. Thus, with richer information, our algorithm can still provide a comparable performance.

The test on a real dataset, REDD [63], also gives the same conclusion. REDD dataset is a sequential stream which tracks the ON/OFF status of electrical devices in several smart homes. The original REDD data records the voltage value of device at every second. We took the time where the voltage value changed as the events. A small noise is added to each timestamp. There are only around 10000 events after the preprocessing.

Figure 10.9 illustrated the accuracy value on the real stream. The ground truth of the real data is the temporal patterns extracted by applying a long sliding batch over a stream. Frequent strings in each batch are clustered based on their temporal information. This naive method is slow but can

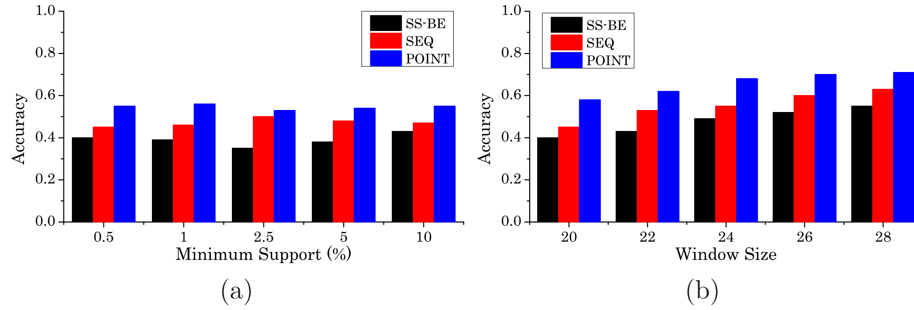


Figure 10.9: Accuracy on real stream with respect to the sliding window size and the minimum support.

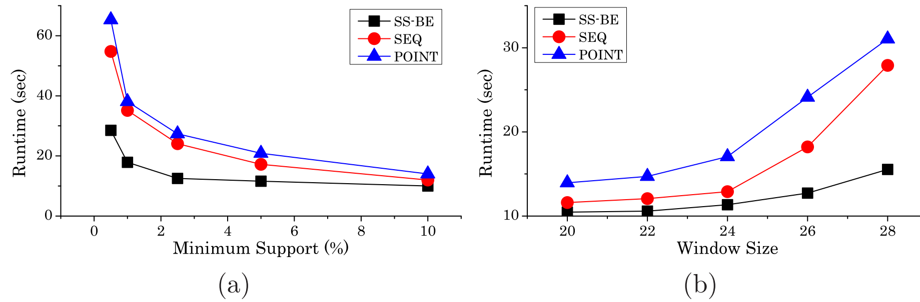


Figure 10.10: Processing time on real stream with respect to the minimum support and the sliding window size.

obtain temporal patterns as many as possible. As shown in the figure, the accuracy value is increasing with a longer window size. Overall, a better accuracy is achieved by our approach. A comparison of runtime on the real stream is investigated in Figure 10.10.

In conclusion, sliding window based algorithms are slower than batch based methods. However, many patterns coming across different sequences and batches are lost. Our approach, gives a similar performance as other sliding window based methods and can provide richer information.

We also investigate the property of our approach when extends to interval-based streams. The synthetic stream is generated in the same way as the point-based stream while each event has a corresponding ending event. In the case of real dataset, we also transferred the REDD dataset to an interval-based stream by taking the increasing and decreasing of voltage as the start-

ing and ending time of an event respectively. Our point-based approach will treat each starting and ending point as an independent event. Figure 10.11 illustrates the runtime performance on the synthetic stream with various stream length, minimum support and sliding window size.

Figure 10.12 depicts the runtime performance on the real stream. It is

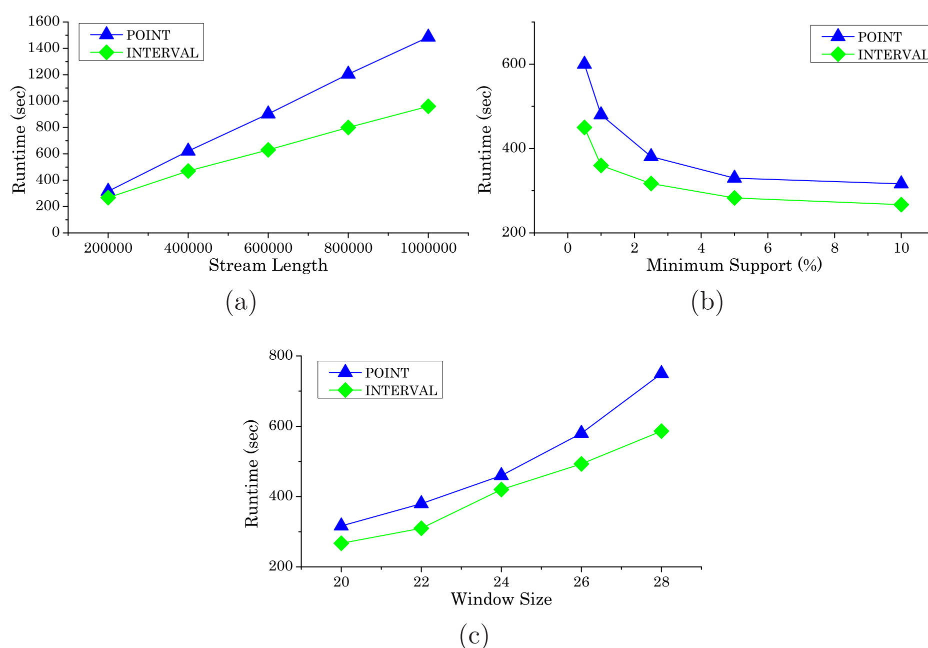
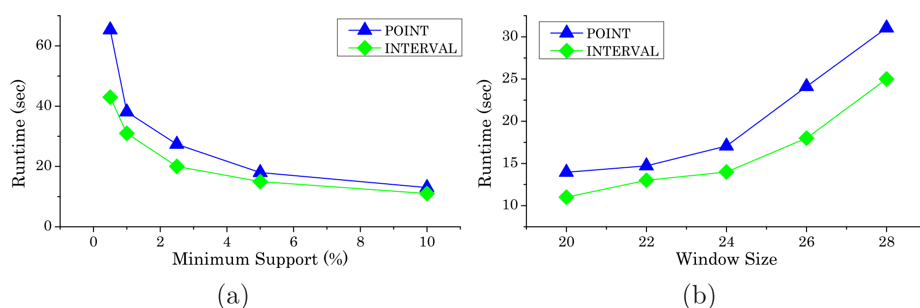


Figure 10.11: Processing time on interval-based synthetic stream with respect to the stream length, the minimum support and the sliding window size..



(a) (b)

Figure 10.12: Processing time on interval-based real stream with respect to the minimum support and the sliding window size.

clear that our interval-based approach is more efficient than the point-based one. The stream length that the point-based approach processes is actually doubled the size of the original stream since each interval-based event is treated as two separate events. Furthermore, incomplete patterns in which some ending points are missing, such as "*A+ before B+*", will be generated by a point-based approach. Those patterns can be identified by post-processing or completeness checking and pruning during the mining process, as shown in [102], but with additional runtime cost. Our interval-based approach treats each interval-based event as a 2D point. Only one extra dimension is introduced in the clustering step, and the completeness of pattern is guaranteed implicitly in the model. In conclusion, our approach can be extended to handle interval-based data easily and efficiently.

10.6 Conclusion and Future Perspective

In this part, we presented our novel efficient algorithm for temporal pattern mining over streams. Noise in the temporal information will not affect the order of events in a sequence. A similarity measure is introduced and the stream clustering approach is applied to generate temporal patterns. The extension to interval-based data is smooth using our approach. Experimental studies indicate that our approach, with richer information, can still provide comparable performance. One drawback of frequent pattern mining is the large number patterns been extracted. A lot of research has been made on condensed representations such as maximum patterns and closed patterns. Such condensed representation in the context of incremental mining with temporal information will be investigated in the future.

Part III

kNN based Clustering

Chapter 11

Introduction

Clustering is a vital pattern mining task that aims to group similar objects into the same group while separating dissimilar objects into different groups. Clustering algorithms are traditionally divided into one of several categories, which include partitioning, hierarchical, model, density, and grid-based approaches. In this part, novel clustering algorithms are proposed to address two new challenges that appeared in application scenarios, such as scientific data analysis. In those application scenarios, the ability to handle clusters in arbitrary shapes is necessary. Therefore, clustering algorithms developed in this part are more or less density-based clustering algorithms.

In general, density-based clustering algorithms need more than one parameter to define the density threshold, which makes parameter tuning difficult. k nearest neighbor density-based clustering algorithms are proposed to address the problem by using the concepts of k nearest neighbor and reverse k nearest neighbor to define density. Algorithms proposed in this part are all k nearest neighbor based. More importantly, only one parameter is required for clustering.

11.1 Density-based Clustering

In this section, the most famous density-based clustering, DBSCAN [34], is recalled. The general structure of density-based clustering algorithms is

described. Usually, density-based clustering algorithms require more than one parameter, the density threshold definition. To address the problem, the idea of k nearest neighbor (kNN) is employed. Therefore, kNN-based clustering algorithms are also introduced afterward.

DBSCAN

DBSCAN (Density Based Spatial Clustering of Applications with Noise) [34] is one of the most famous density-based clustering algorithms. The key idea of density-based clustering is that for each observation in a cluster, the local density must be higher than a given density threshold. In DBSCAN, such local density is described as the number of observations within a given radius ϵ. The density threshold is the minimum number of observations within the radius ϵ, denoted as $minPts$.

Given the data set of observations $\mathcal{D}$, the distance function between observations in $\mathcal{D}$ is denoted as $dist$. For a given $\epsilon \in \mathbb{R}^+$, the set of ϵ-neighborhood of an observation $p \in \mathcal{D}$ is denoted as $N_\epsilon(p)$:

$$N_\epsilon(p) = \{o \in \mathcal{D} | dist(p, o) \leq \epsilon\}.$$

Definition 11.1.1 (Core points, border points and local density)
Given $minPts \in \mathbb{N}^+, minPts \leq |\mathcal{D}|$, a data point $p \in \mathcal{D}$ is called a *core point* if $|N_\epsilon(p)| \geq minPts$, *border point* if otherwise. $|N_\epsilon(p)|$ is the *local density* of the data point p.

Usually, clusters in density-based clustering are formed by connecting consecutive core points and their neighbors together. To determine how data points are connected, the concepts of *direct density-reachability*, *density-reachability* and *density-connectivity* are introduced in DBSCAN.

Definition 11.1.2 (direct density-reachability)
Given $\mathcal{D}, \epsilon$ and $minPts$. An observation $p \in \mathcal{D}$ is *directly density-reachable* from an observation $q \in \mathcal{D}$ w.r.t. ϵ and $minPts$ if $|N_\epsilon(q)| \geq \wedge p \in N_\epsilon(q)$.

Definition 11.1.3 (density-reachability)
Given $\mathcal{D}, \epsilon$ and $minPts$. An observation $p \in \mathcal{D}$ is *density-reachable* from an

observation $q \in \mathcal{D}$ w.r.t. ϵ and $minPts$ if there is a sequence of observations $p_1, \ldots, p_n$ where $p_1 = q, p_n = p$ such that p_{i+1} is directly density-reachable from p_i for $1 \leq i \leq n-1$.

Definition 11.1.4 (density-connectivity)
Given $\mathcal{D}, \epsilon$ and $minPts$. An observation $p \in \mathcal{D}$ is *density-connected* to an observation $q \in \mathcal{D}$ w.r.t. ϵ and $minPts$ if there is an observation $o \in \mathcal{D}$ such that both p and q are density reachable from o.

Algorithm 11.1: DBSCAN

```
   Input: Dataset D, ε, minPts
1  foreach Unvisited data point p ∈ D do
2      Mark p as visited;
3      if |N_ε(p)| < MinPts then
           // Border point
4          Mark p as noise;
5      else
6          C ← New empty cluster;
7          Add p to C;
8          N_ε ← N_ε(p);
9          foreach q ∈ N_ε do
10             if q is not visited then
11                 Mark q as visited;
12                 if |N_ε(q)| ≥ minPts then
                       // Core point
13                     N_ε ← N_ε(q) ∪ N_ε;
14                 end
15             if q is not yet a member of any cluster then
16                 Add q to C;
17             end
18         end
19     end
20 end
```

A density-based cluster is defined to be a set of density-connected observations. Given a cluster C, all data points that are density-reachable from a data point $q \in C$ are also in cluster C. According to the definition above,

density-based clusters are mainly formed by connecting core points together. Border points that are directly density-connected to a core point are also assigned to a cluster. Border points that are not directly density-connected to any core points are defined as noise.

The algorithm DBSCAN uses the previously described concepts and computes flat density-based clusters w.r.t. the user-defined parameters ϵ and $minPts$. DBSCAN can detect clusters in arbitrary shapes without requiring prior knowledge of cluster numbers. To identify clusters, DBSCAN starts from a core point p and recursively traversing directly density-connected data points w.r.t. to core points in the cluster, until all directly density-connected data points are accessed. The detailed process of DBSCAN is illustrated in Algorithm 11.1.

The recursive cluster expanding process of DBSCAN can also be found in many other density-based clustering algorithms. Different local density definitions are used to address weaknesses in DBSCAN. For example, DBSCAN requires two parameters, which makes parameter tuning difficult. Novel density-based clustering algorithms, such as k nearest neighbor (kNN) based approaches, are proposed to tackle this problem.

kNN-based Clustering

As mentioned above, one of the main drawbacks of DBSCAN is the difficulty of parameter tuning. To address the problem, novel k nearest neighbor density-based clustering algorithms are proposed to reduce the complexity of the problem to the use of a single parameter.

In this section, a novel density-based clustering algorithm, RNNDBSCAN [15] is introduced, which employed the concepts of kNN and RkNN (Reverse k nearest neighbor) to define local density. Therefore, this algorithm only requires one parameter k.

Definition 11.1.5 (k nearest neighbor)
Given dataset $\mathcal{D}$ and $k \in \mathbb{N}^+$. The set of k nearest neighbors of data point $p \in \mathcal{D}$ is $N_k(p)$ where $|N_k(p)| = k$ and $\forall o \in \mathcal{D}, o \notin N_k(p) : dist(p, q) \leq dist(p, o)$.

Definition 11.1.6 (reverse k nearest neighbor)
Given dataset $\mathcal{D}$ and $k \in \mathbb{N}^+$. The set of reverse k nearest neighbors of data point $p \in \mathcal{D}$ is $R_k(p)$ where $\forall q \in R_k(p) : p \in N_k(q)$.

Based the concepts of kNN and RkNN described above, a data point p is a core point iff $|R_k(p)| \geq k$ where k is the user defined parameter. After core points being identified, the concept of directly density-reachability can also be defined for RNNDBSCAN:

Definition 11.1.7 (directly density-reachability)
Given dataset $\mathcal{D}$ and $k \in \mathbb{N}^+$. An observation $p \in \mathcal{D}$ is directly density reachable from an observation $q \in \mathcal{D}$ if $p \in N_k(q) \wedge |R_k(q)| \geq k$.

Density-reachability and density-connectivity are then defined similarly to DBSCAN. In the last step of the RNNDBSCAN algorithm, the recursive cluster expanding process of DBSCAN (Algorithm 11.1) is used to generate clusters. Other kNN based clustering algorithms such as ISDBSCAN [20] and ISBDBSCAN [76] are also designed similarly. Some kNN based clustering algorithms are not following the DBSCAN-alike paradigm. For example, the KNNCLUST [94] algorithm does not define core and border points. Instead, the algorithm starts by assigning each data point with a separate cluster label. Then, in each iteration, the cluster label of a data point is updated to the majority cluster label in its k nearest neighbors.

11.2 kNN-based Clustering - New Challenges

Two kNN based clustering algorithms are introduced in this part to tackle two new challenges in clustering: shape alternation adaptability (Chapter 12) and extremely noisy dataset (Chapter 13).

Shape Alternation Adaptability

As shown above, most density-based clustering algorithms follow a DBSCAN-alike paradigm. Clusters are formed by connecting consecutive dense regions together. A region is dense if its local density exceeds a given threshold,

which is defined using kNNs in our case. In other words, the connectivity is purely determined by local density. The shape of data points distributed in the dataset on a global scale is not considered. However, in some applications such as scientific research, a series of datasets are generated at once, waiting for clustering, since an experiment has to be repeated for many times. Points distribution shape in those datasets may alter, for instance, from a strip shape to a dumbbell shape, which implies cluster changing. Figure 11.1 gives an example of RNNDBSCAN on a series of datasets with shape alternation.

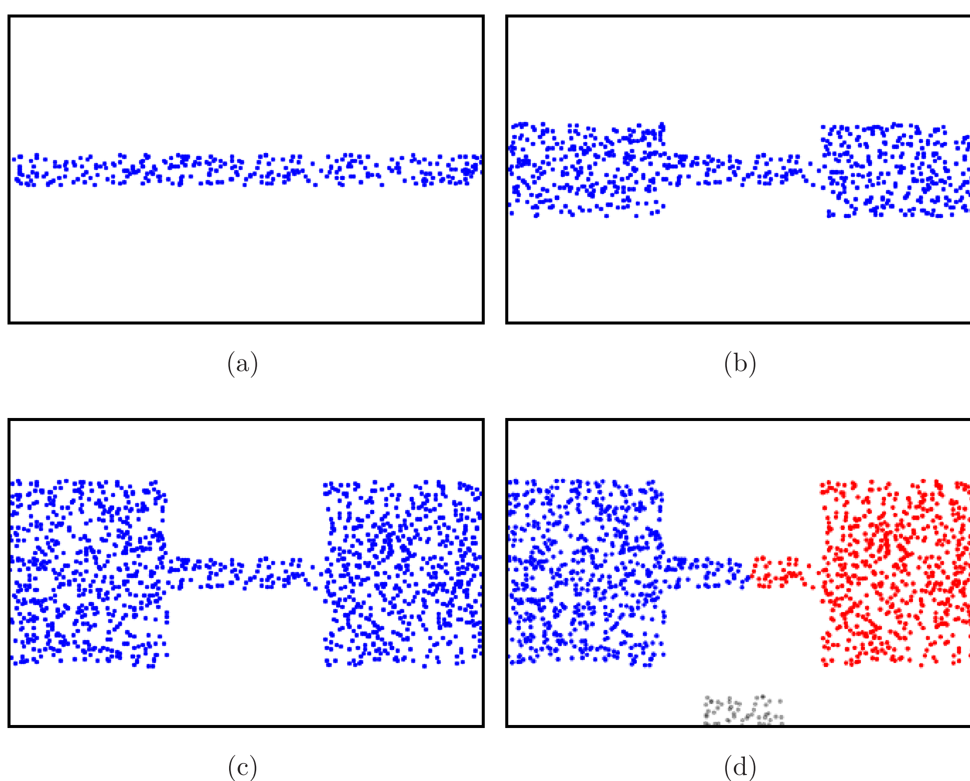


Figure 11.1: RNNDBSCAN correctly detects one cluster in (a) but failed to find two clusters properly in (b) and (c), when parameters are fixed. Even if we set parameters to find two clusters, the small cluster in (d) is incorrectly labeled as noise.

Figure 11.1(a-c) illustrate three datasets with data points distributed in different shapes but similar densities. Clusters are identified using a DBSCAN-alike approach (RNNDBSCAN [15]) under the same parameter.

In the first dataset (a), it is safe to say that the result is correct as all points are in the same strip-shaped cluster. When the number of points increased on the two ends (b,c), it is likely that there are two clusters. However, RNNDBSCAN can not adapt automatically and still return a single cluster. Of course, we can also choose the parameters to return two clusters in Figure 11.1(b,c), but then the single strip-shaped cluster in (a) will be separated. Therefore, users always need to manually adjust the parameter on each dataset to generate the correct result, which is infeasible for applications where hundreds of datasets are waiting for clustering. We call this problem the lacking of shape alternation adaptability. Obviously, this problem is caused by the use of the DBSCAN-alike paradigm, where a single threshold is used to determine whether a region is dense or not. There are also some other problems, even in the case of a single dataset. For example, in Figure 11.1d, the small cluster (gray) is identified as noise if we set parameters to detect the two big clusters properly since the density threshold is too large.

Existing DBSCAN-alike approaches can not adapt automatically when distribution shapes alter across different datasets. A painful parameter tuning process is necessary for each dataset. Clustering algorithms with shape alternation adaptability can adapt to cluster shapes across different datasets, without extra parameter tuning. In Chapter 12, a novel kNN based clustering algorithm with shape alternation adaptability is proposed.

Extremely Noisy Dataset

In some applications such as scientific data analysis, the noise level in datasets can be extremely high, i.e., the quantity of noises is considerably larger than the number of "in-cluster" data points, and the density in noise regions are also similar to the density in the region of clusters. Figure 11.2 is an example from scientific research. The dataset contains the position of atoms detected using microscopy technology. Material scientists only interest in areas that atoms concentrate. Other atoms, which are the majorities, are considered as "noise". A clustering algorithm that can extract concentrated atoms from the extremely noisy dataset is essential for scientists. However, most existing

clustering algorithms only consider noises up to a moderate level. They can not extract clusters correctly from such kinds of datasets, or a painful parameter tuning process is necessary.

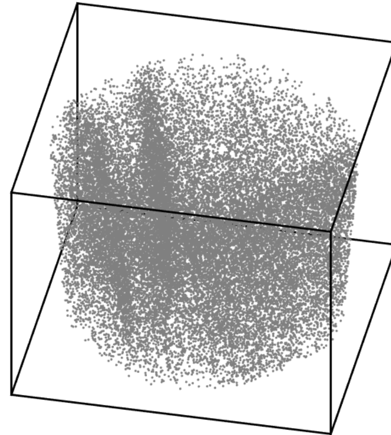


Figure 11.2: A real extremely noisy dataset example.

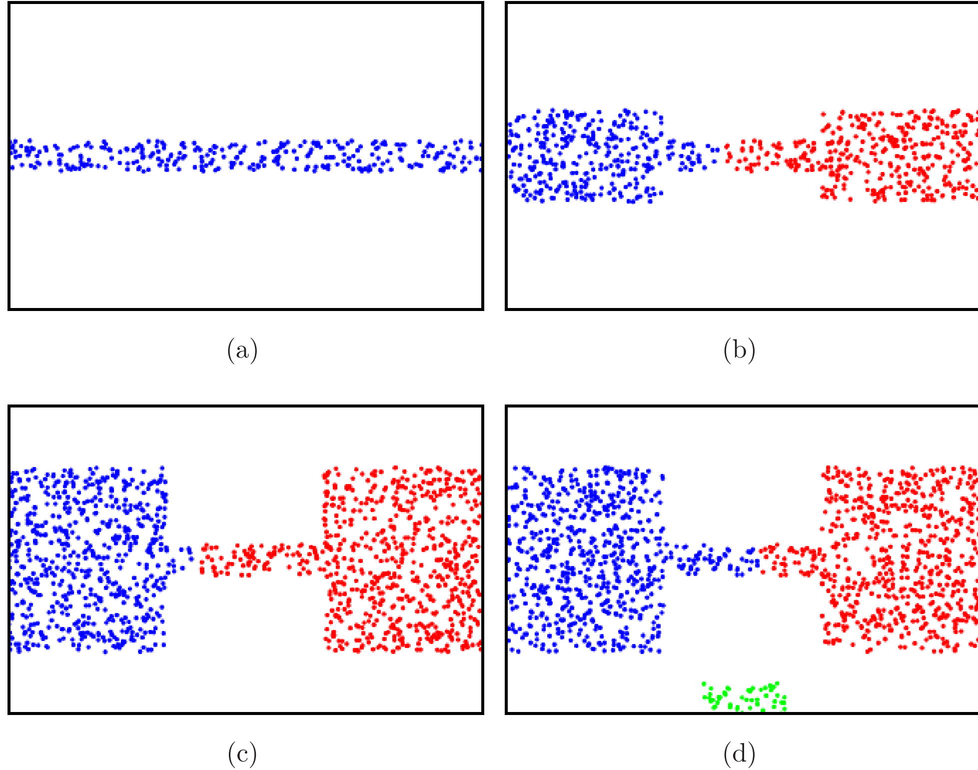
To address the issue, Chapter 13 studies the property of extremely noisy datasets and introduces a two-step kNN based clustering algorithm. It can identify arbitrarily shaped clusters under high noise settings. Moreover, it requires only a single parameter.

Chapter 12

Shape Alternation Adaptable Clustering

The problem of shape alternation in a series of datasets is described in the previous chapter. To tackle this problem, a new kNN-based clustering algorithm is proposed here. The DBSCAN-alike structure is avoided. The new approach can handle arbitrary shaped clusters using a single parameter k. With shape alternation adaptability, it is also aware of the overall data distribution so that clusters can be identified and separated even if it is densely connected to other clusters, without parameter tuning. We estimate similarities using the concept of kNN to reflect the probability that two points belonging to the same cluster and the probability that a point is a cluster center. The Affinity Propagation [38] algorithm is applied to extract cluster centers.

Figure 12.1 illustrates the clustering result of our approach on toy examples given in Figure 11.1. The parameter k is selected so that three clusters are detected in Figure 12.1d. Without manual parameter tuning for the rest of the datasets, our approach can adapt itself to identify clusters properly.



(a) (b) (c) (d)

Figure 12.1: Clustering results of our approach. The parameter k is set to work properly on (d). No further parameter tuning is applied to the rest of datasets.

12.1 Related Works

The concept of k-nearest neighbor is widely used in density-based clustering approaches. For example, the relationship between the connectivity of a mutual k-nearest neighbor graph and the clustering structure is studied in [14]. The SNN [33] algorithm uses the kNN to handle clusters with various densities. LDBSCAN [32] employs the local outlier factor as the metric, which is also defined using kNN. However, extra parameters are required in those approaches.

Our work is a single parameter kNN clustering algorithm. Most reported single parameter kNN clustering algorithms borrowed the idea of DBSCAN [34]. The RECORD [98] algorithm makes use of kNN graph and reverse kNN

graph to define core points. Clusters are extracted from the subgraph of core observations. HDBSCAN [18] builds a minimum spanning tree on a mutual reachability graph. Edges are iteratively removed to generate optimized clusters. The IS-DBSCAN [20] approach introduced the concept of influence space of a data point, which is defined as the intersect between its reverse and k nearest neighbor sets. The influence space concept is then used to describe local density and reachability of data points. ISB-DBSCAN [76] goes a step further by using an undirected influence space graph. In RNNDBSCAN [15], density reachability is defined by only using the concept of k-nearest neighbor and reverse k-nearest neighbor. A data point is a core point if the number of its reverse k-nearest neighbor is larger than k. KNNCLUST [94] does not follow a DBSCAN clustering style. Instead, it starts clustering by assigning different cluster labels to each data point. The cluster label is then updated recursively by computing a posterior probability concerning labels in kNN. Unfortunately, all methods above can not adapt automatically to the drift in cluster shapes. The parameter k needs to be determined separately for a series of datasets.

Of course, there are algorithms such as spectral clustering [89] that can handle cluster shape alteration. However, they need to know the cluster number in advance.

12.2 Preliminaries

A good clustering should have a high intra-cluster similarity. To identify cluster properly, we need a good estimation of similarities, and an algorithm that maximizes the intra-cluster similarity. Affinity Propagation (AP) [38] is designed to identify clusters that maximizing the intra-cluster similarity based on a similarity matrix and a preference vector. As suggested in AP, the similarity matrix and the preference vector should reflect the probability that two points belong to the same cluster and the probability that a point is a cluster center. Thus, the major challenge of this work is how to estimate both probabilities properly.

In this work, we use the kNN distance to measure those two probabilities.

The heuristic of cluster centers proposed is DPC [86] is also employed in computing the preference vector. Those two probabilities are utilized as the similarity matrix and the preference vector for AP to generate cluster centers, which eventually produces clusters. Only one parameter k is employed in the whole process.

Affinity Propagation

Affinity Propagation does not require the user to estimate the number of clusters or the density of points. Instead, the user has to provide a similarity matrix S in which s_{ij} is the similarity between point i and j. The diagonal of S is the preference vector. s_{ii} is the *preference* of point i, denoted as pref_i. The similarity measures how likely that two points belong to the same cluster. The preference value reflects the likelihood that a point being a cluster center. Cluster centers and cluster assignments are determined by maximizing the overall intra-cluster similarity:

$$\mathcal{S} = \sum_{i=1}^{N} S_{i,e_i} \tag{12.1}$$

where e_i is the cluster center of point i. Obviously, the overall distribution of data points influences the final clustering result.

AP is reported to work well on certain tasks such as computational biology, where the similarity between observations is well defined. In a more general case, the similarity between points is set to the *negative* Euclidean distance. Preferences of all points are initialized to the same value, such as the minimum or the median of similarities. However, it is difficult to achieve the desired results with the default setting of AP. Figure 12.2 illustrates the clustering results on two datasets. Although the data only contains 2/3 simple clusters, AP does not identify these clusters successfully due to shapes and cluster proximity.

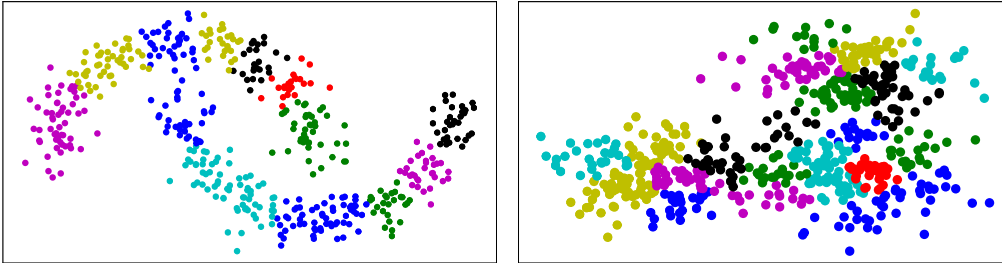


Figure 12.2: Clustering results of AP under default settings.

Density Peaks Clustering

Density Peaks Clustering (DPC) is a semi-automated clustering approach, which makes use of two heuristics to highlight cluster centers: 1) *cluster centers are surrounded by neighbors with lower density*; 2) *cluster centers are far away from other points with a higher density.* A decision graph is derived by computing the local density ρ_i and the distance to the nearest point with a higher density δ_i (*delta distance*) for each point. Then the user identifies cluster centers by selecting points manually with both large ρ and large δ.

The DPC algorithm shows outstanding performance in a variety of clustering tasks by providing a great visualization tool for identifying cluster centers manually. There are some algorithms proposed in recent years to fully automate the cluster center selection process by analyzing the values of ρ and δ. Indeed, the original DPC algorithm also proposes to examine the product of ρ and δ for each data point. However, those approaches introduce additional parameters, which need to be precisely selected [104].

Nonetheless, the heuristic introduced by DPC is useful for our approach, which provides a good description of cluster centers. We apply it to compute the probability of a point being a cluster center, i.e., the preference of points.

12.3 Ada$k$NN Clustering Algorithm

In this section, the kNN based clustering algorithm with shape alternation adaptability, Ada$k$NN, is presented. There are four major steps in our approach: 1) estimating the similarity matrix; 2) estimating the preference

vector; 3) using affinity propagation to identify cluster centers; 4) assigning labels to the rest of data points.

Similarity Estimation

As mentioned above, similarities used by AP should reflect how likely that two points belong to the same cluster. To achieve the goal, we adopt the idea of minimax distance on graph. Minimax distance can model the underlying structures and the transitive relations nonparametrically [23]. Moreover, minimax distance on a graph is equivalent to longest edge on the path of the corresponding minimum spanning tree (MST). In this work, we built MST on our dataset. Instead of using edge length, the *minimum edge density* on the path from point i to j is used to measure similarity. Edge density of an edge e is represented using the kNN distance of the center point of e in the dataset.

For simplicity, the MST is built based on the Euclidean distance between points, known as the Euclidean Minimum Spanning Tree (EMST). Other distance function can also be used but are beyond the scope of this work.

Let T be the EMST built on our dataset, T_{ij} be the path from point i to point j, which is formed by a list of adjacent edges $e_{ii_1}, e_{i_1 i_2}, \ldots, e_{i_m j}$ of T. The distance between two points i, j is:

$$dist_{ij} = \max_{e \in T_{ij}} d_k(e) \tag{12.2}$$

where $d_k(e)$ of edge e is the kNN distance of the center point of e with respect to points in the dataset.

Intuitively, we measure how sparse it is on the path from point i to j. If data points stay densely around all edges on the path, then $dist_{i,j}$ is small, and i, j tend to come from the same cluster. Figure 12.3a shows a toy example. Dashed lines (red and green) represent the kNN distance of corresponding edges, i.e., the edge density. Thus, the point B is closer to A than C. As AP asks for negative similarity values, we normalize our distance

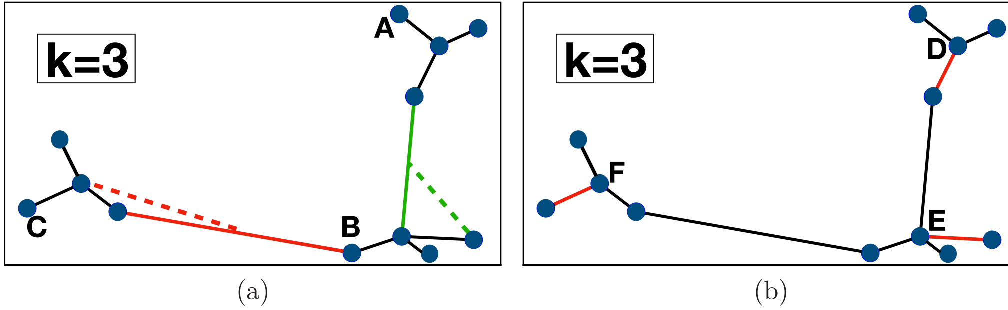


Figure 12.3: Similarity and Preference on toy MST example.

and take the negative as the similarity value:

$$s_{ij} = -\frac{dist_{ij} - dist_{min}}{dist_{max} - dist_{min}} \tag{12.3}$$

where $dist_{min} = \min_{i \neq j}(dist_{ij})$, $dist_{max} = \max_{i \neq j}(dist_{ij})$

Algorithm 10 illustrates procedure for calculating the similarity matrix.

Algorithm 12.1: Similarity Matrix

Input: Dataset X, Parameter k
Output: Similarity Matrix S

1 $T \leftarrow$ `MinimumSpanningTree(X)` ;
2 **foreach** $x_i \in X$ **do**
3   **foreach** $x_j \in X, j \neq i$ **do**
4         $dist_{ij} = \max_{e \in T_{ij}} d_k(e)$;
5   **end**
6 **end**
7 **foreach** $s_{ij} \in S, i \neq j$ **do**
8     $s_{ij} \leftarrow -\frac{dist_{ij} - dist_{min}}{dist_{max} - dist_{min}}$;
9 **end**
10 **return** S;

Preference Estimation

The preference value of a data point implies the likelihood of being a cluster center. According to the heuristic of DPC, the preference value of a data

point i is positively correlated with two properties:

1. ρ_i, the local density of point i,

2. δ_i, the delta distance of point i (distance to a point with larger ρ).

As DPC suggests to analyzing the production of ρ and δ for automatic cluster center detection, we can assume $\text{pref}_i \propto \delta_i \cdot \rho_i$.

AP requires both similarity and preference values to be negative values. A cluster center point with large ρ_i and δ_i should have a negative preference value close to 0. Preference values of off-center points must be much smaller than 0. In consequence, we define the preference of a point i as:

$$\text{pref}_i = (\delta_i - dist_{max}) \cdot \rho_i \tag{12.4}$$

where the local density ρ_i is defined as the kNN distance of point i, which is similar to the edge density $d_k(e)$ defined above.

In Figure 12.3b, the length of the red line is the local density of points D, E, and F. δ_i and $dist_{max}$ are defined using distance values in equation 12.2. Under such definition, boundary points may also have high preference values (close to 0) when $\delta_i - dist_{max} \approx 0$. AP might identify those points as cluster centers of small clusters. However, their cluster size are smaller than k so that we can identify and correct those "outlier clusters" easily.

Furthermore, pref_j is not defined in equation 12.4 if $\rho_j = \max(\rho)$, since δ_j is not defined. Thus, we let $\text{pref}_j = \max_{i \neq j}(\text{pref}_i)$ as j is very likely to be a cluster center. Moreover, we need to normalize the preference value by dividing by the maximum nonzero preference value:

$$\text{pref}_i = -\frac{\text{pref}_i}{\max_{\text{pref} \neq 0} \text{pref}} \tag{12.5}$$

Such normalization reflects how likely a point i is to be a cluster center, when compared with the most possible one. Algorithm 12.2 presents the procedure for calculating the preference vector.

Algorithm 12.2: Preference(X, k, $dist$)

Input: Dataset X, Parameter k, Distance Matrix $dist$

Output: Preference Vector pref

1 $\rho_i \leftarrow$ local density of $x_i \in X$;

2 $j \leftarrow \operatorname{argmax}_i \rho_i$;

3 **foreach** $x_i \in X, i \neq j$ **do**

4     $\delta_i \leftarrow$delta distance of x_i;

5     $\text{pref}_i \leftarrow (\delta_i - dist_{max}) \cdot \rho_i$;

6 **end**

7 $\text{pref}_j \leftarrow \max_{i \neq j}(\text{pref}_i)$;

8 $\text{pref} \leftarrow -\frac{\text{pref}}{\max_{\text{pref} \neq 0} \text{pref}}$;

9 **return** pref;

Generating Clusters

After estimating similarities and preferences, we add the preference vector **pref** into the diagonal of the similarity matrix S and let AP determine cluster centers. As AP tends to maximize the overall intra-cluster similarity, the global distribution of data points is also taken into consideration when identifying cluster centers.

Cluster labels of the rest of the points are assigned in a similar way to DPC. We traverse unlabeled points in descending order according to the local densities. The label of the nearest cluster is attached. A refinement step is introduced by comparing the label of each point to its k nearest neighbors. Labels will be updated to the majority label among kNN. Such a refinement step is useful since AP may generate "outlier clusters", as mentioned above. Algorithm 12.3 illustrates the overall procedure of our approach.

Computational Complexity

Our approach contains four primary steps: 1) EMST generation; 2) Similarity and Preference estimation; 3) AP for cluster centers generation and 4) Cluster labels assignment. The complexity of the first step, generating EMST, can achieve $O(N \log N)$ by utilizing index structure and the Prim's algorithm [11]. Similarity and preference computation involves the EMST traversing

Algorithm 12.3: Generating Clusters

```
Input: Dataset X, Parameter k
Output: Cluster labels l = [l_1, ..., l_N]^T
1  N × N similarity matrix: s ← Similarity(X, k);
2  N × 1 preference vector: pref ← Preference(X, k, −s);
3  diag(s) ← pref;
4  ∀x_i ∈ X, l_i ← −1 ;                    // initial cluster label to -1
5  Exemplars E ← AffinityPropagation(s);
6  foreach x_i ∈ E do
7  |  l_i ← A unique cluster label;
8  end
9  Sort X in descending order of ρ;
10 foreach all x_i ∈ X, l_i = −1 do
11 |  l_i = argmax_{(l_j ≠ −1)}(s_{i,j});
12 end
13 foreach all x_i ∈ X do
14 |  l_i ← majority label among kNN of x_i ;        // refinement
15 end
16 return c;
```

and the similarity matrix filling, which takes $O(N^2)$. The time complexity of Affinity Propagation makes $O(N^2T)$, where N is the number of data points, T is the number of iterations. The last cluster assignment step is mainly about nearest neighbor search, which takes $O(N \log N)$ in total. In summary, our k-nearest neighbor density-based clustering approach has a complexity of $O(N^2T)$.

12.4 Experimental Results

We investigate our approach on both synthetic and real-world datasets from the UCI Machine Learning Repository [31]. Additionally, several artificial datasets of varying sizes, densities, and shapes were generated to highlight the effectivity of our approach. A summary of each dataset is provided in Table12.1.

Affinity Propagation algorithm under default settings (as described in

Table 12.1: Dataset Statistics

(a) Synthetic Datasets

Data	N	N_c	d
spiral[31]	312	3	2
aggregation[31]	788	7	2
flame[31]	240	2	2
d31[31]	3100	31	2
moon (Fig 12.4)	600	2	2
gaussian (Fig 12.5)	800	2	2
blobs (Fig 12.6)	1K, 5K, 10K	2	2
strip (Fig 12.7)	1K, 5K, 10K	2	2

(b) Real-world Datasets

Data	N	N_c	d
iris[31]	150	3	4
digits[31]	1797	10	64
seeds[31]	210	3	7
segments[31]	2310	7	19
seismic-bumps[31]	210	3	8
satimage[31]	6430	6	37
banknote[31]	1372	2	4

Section 12.2) is conducted. RNNDBSCAN algorithm is included to represent recent DBSCAN-alike kNN clustering algorithms. KNNCLUST stands for the performance of kNN clustering approaches that are not DBSCAN-alike. Furthermore, two density-based algorithms, DBSCAN and OPTICS, and a manifold-based algorithm, Spectral Clustering, are also conducted as a baseline.

Adjusted Rand Index (ARI) and Normalized Mutual Information (NMI) are reported. For each kNN based algorithm, we vary k from 1 to 100 and report the result of k with the best ARI score. Clusters generated by KNNCLUST are inconsistent across multiple runs on the same dataset due to its random access to data points. Thus, the average score of several runs is reported while the best run is used for visualization. For DBSCAN, min_{pts}

is selected from the set $\{2, 5, 10, 20\}$, and eps is selected over a set of values equal to the min_{pts} nearest neighbor distance of each observation. OPTICS has the same parameter pool as DBSCAN, while the schema mentioned in [6] is used for cluster generation. The affinity matrix we used in Spectral Clustering (SC) is constructed using the kNN method, where k also varies from 1 to 100. Table 12.2 and 12.3 show the ARI and NMI score on synthetic and real datasets. The best values in each row are marked in bold.

Our approach has the best ARI and NMI scores in almost all synthetic datasets. It is only defeated by the Spectral Clustering on the `gaus` dataset with a narrow margin. On real-world datasets, our method is also very competitive. It has the best ARI score in 4 out of 7 datasets. It also ranks second or third place in the rest of the datasets. In terms of NMI score, our approach also outperforms competitors in real datasets. It has the highest NMI score in 5 real datasets and takes the second place in `sati` dataset with a marginal gap. `bank` dataset is the only one that our approach is not among top-2, but still better than AP, KNNCLUST, and OPTICS. In summary, our approach provides solid clustering quality compared with traditional methods, as well as novel kNN based methods.

Figure 12.4 12.5 12.6 and 12.7 visualize the clustering result of our approach and other kNN clustering algorithms on a series synthetic datasets. Our approach provides the best clustering result, especially in Figure. 12.5 and 12.6, where two or three clusters are mutually overlapped. Our approach can identify clusters from densely connected data, while other kNN methods fail.

Each experiment above also compares the clustering quality (ARI score) of our approach with other algorithms. The available range of k with a high ARI score is much more extensive than competitors, which means that the user can estimate the parameter roughly. Indeed, accepting roughly estimated parameters is an essential feature of kNN clustering algorithms since users do not need to search for the best parameter value accurately as $k$means or spectral clustering required, which is time-saving.

More importantly, the more extensive available range of k also means that our approach can handle a series of datasets without parameter tuning on

Table 12.2: ARI & NMI Score on Synthetic Datasets

Dataset		**Our**	**AP**	**RNN**	**KNN**	**DBS**	**OPT**	**SC**
spir	ari	**1**	0.101	0.179	**1**	**1**	0.162	0.388
	nmi	**1**	0.538	0.454	**1**	**1**	0.395	0.466
	n_c	3	34	8	3	3	4	3
aggr	ari	**0.996**	0.177	0.991	0.809	0.992	0.984	0.809
	nmi	**0.988**	0.681	0.987	0.895	0.98	0.977	0.895
	n_c	7	38	7	5	8	8	5
flame	ari	**0.97**	0.086	0.949	0.208	**0.97**	0.967	0.650
	nmi	**0.93**	0.483	0.891	0.517	**0.93**	0.927	0.741
	n_c	2	19	3	16	3	2	3
d31	ari	**0.954**	0.529	0.855	0.457	0.884	0.641	0.943
	nmi	**0.97**	0.854	0.917	0.081	0.927	0.879	0.962
	n_c	31	86	31	43	32	29	31
moon	ari	**0.984**	0.076	0.976	0.485	0.980	0.976	0.802
	nmi	**0.97**	0.458	0.946	0.379	0.970	0.951	0.754
	n_c	2	27	2	5	3	3	2
gaus	ari	0.962	0.042	0.183	0.544	0.526	0.034	**0.98**
	nmi	0.931	0.409	0.156	0.554	0.501	0.195	**0.97**
	n_c	2	47	10	5	3	12	2
blobs	ari	**0.941**	0.093	0.511	0.245	0.765	0.469	0.921
	nmi	**0.908**	0.508	0.569	0.493	0.717	0.619	0.887
	n_c	3	40	5	37	4	2	3
strip	ari	**0.783**	0.097	0.530	0.355	0.750	0.621	0.709
	nmi	**0.780**	0.487	0.544	0.477	0.723	0.626	0.697
	n_c	3	44	4	14	3	3	3
Average	ari	**0.949**	0.150	0.647	0.443	0.791	0.607	0.672
	nmi	**0.935**	0.552	0.640	0.485	0.781	0.696	0.711

each dataset. In this example, our method can identify all clusters properly by fixing k at around 30. In contrast, KNNCLUST is able to find clusters correctly only if the parameter is accurately selected. Spectral clustering works only if the number of clusters is chosen appropriately. RNNDBSCAN has a wider parameter range, but parameter tuning is still necessary across

Table 12.3: ARI & NMI Score on Real-World Datasets

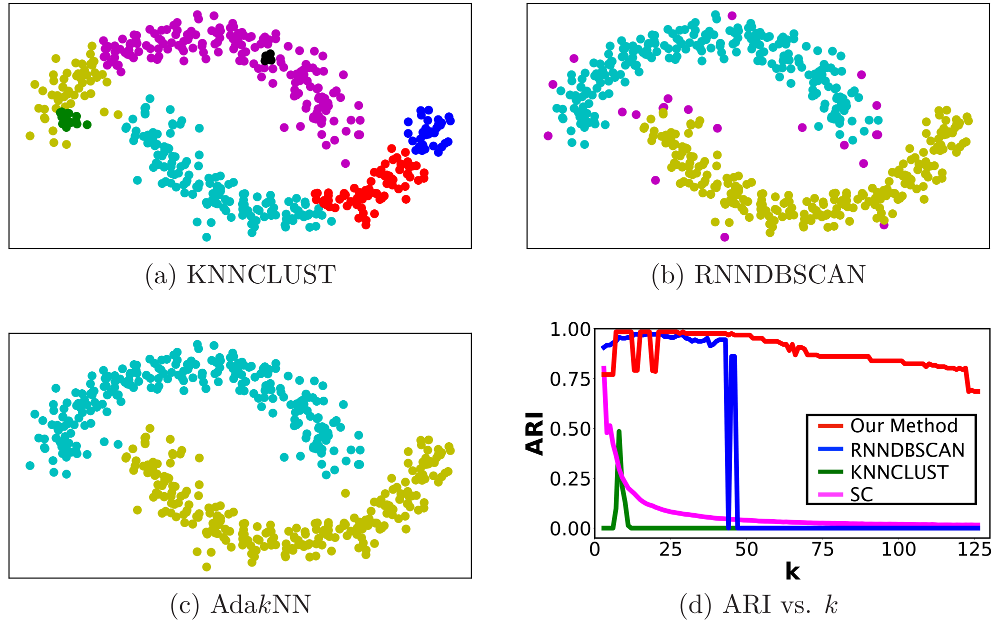
Dataset		**Our**	**AP**	**RNN**	**KNN**	**DBS**	**OPT**	**SC**
iris	ari	**0.882**	0.344	0.548	0.562	0.739	0.578	0.767
	nmi	**0.867**	0.633	0.688	0.664	0.722	0.731	0.811
	n_c	4	12	3	5	3	3	3
digi	ari	**0.799**	0.126	0.574	0.783	0.677	0.116	0.756
	nmi	**0.854**	0.667	0.742	0.848	0.805	0.495	**0.854**
	n_c	10	142	34	19	17	15	10
seed	ari	**0.753**	0.177	0.534	0.434	0.582	0.470	0.602
	nmi	**0.712**	0.510	0.562	0.543	0.554	0.550	0.629
	n_c	4	19	5	11	3	3	3
segm	ari	0.428	0.080	**0.500**	0.116	0.401	0.175	0.473
	nmi	**0.699**	0.544	0.622	0.570	0.610	0.531	0.651
	n_c	4	128	14	214	93	3	8
seis	ari	**0.738**	0.186	0.436	0.477	0.499	0.558	0.616
	nmi	**0.711**	0.526	0.555	0.565	0.531	0.618	0.644
	n_c	4	18	7	9	4	3	3
sati	ari	0.529	0.169	0.432	0.389	0.389	0.090	**0.550**
	nmi	0.628	0.540	0.575	0.537	0.551	0.364	**0.656**
	n_c	4	51	7	19	5	4	4
bank	ari	0.516	0.027	**0.763**	0.029	0.668	0.342	0.531
	nmi	0.569	0.394	**0.707**	0.391	0.638	0.439	0.451
	n_c	4	85	4	122	11	10	3
Average	ari	**0.664**	0.158	0.541	0.399	0.565	0.333	0.614
	nmi	**0.720**	0.545	0.636	0.588	0.630	0.533	0.671

different datasets.

12.5 Further Properties and discussions

Automatic cluster centers detection for DPC

Many approaches are proposed to detect cluster centers for DPC automatically by proposing new density functions [43] or analyzing the scalar value,



(a) KNNCLUST

(b) RNNDBSCAN

(c) Ada$k$NN

(d) ARI vs. k

Figure 12.4: Visualization of kNN based clustering approaches on synthetic datasets and the ARI score under different k values.

such as $\rho_i \cdot \delta_i$, of each point. Points with properties larger than a threshold are returned as cluster centers.

Our approach is different since we keep the information of both ρ and δ, and use AP to find cluster centers that maximize intra-cluster similarity. Thus, cluster centers are determined not only by the properties of each point, but also by the relationship between points. Figure 12.8 illustrates cluster centers found by our approach and their corresponding preference values. We can see that three centers are not those points with the highest preference value.

Various Dataset Size

Despite less complexity in parameter selection, another essential property of kNN clustering over DBSCAN is the dataset size invariant [15]. Figure 12.9 shows the clustering qualities of our approach on two datasets with different dataset sizes. We can see that, although the DBSCAN structure is not used,

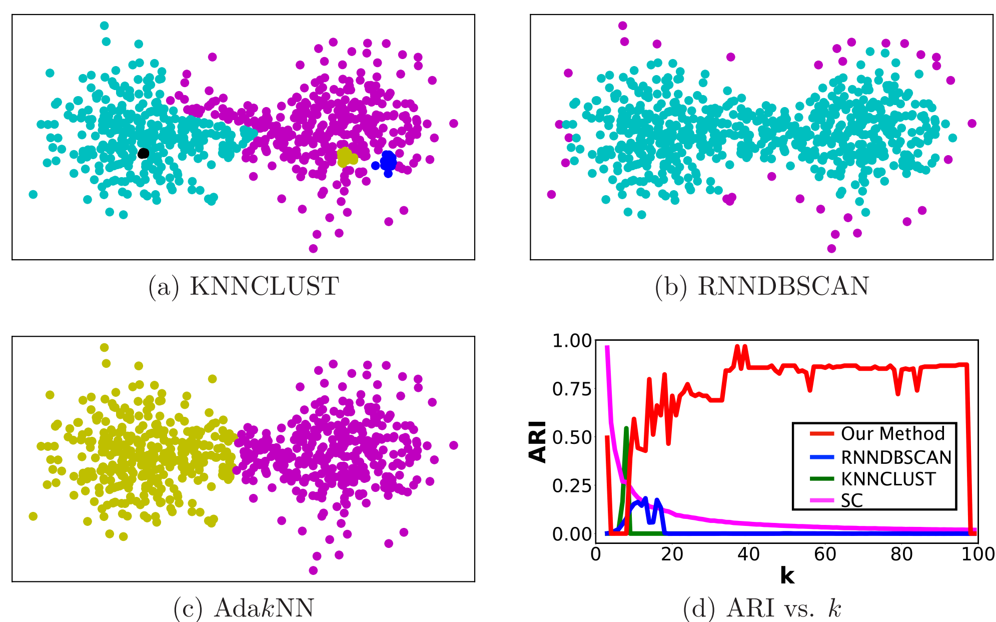


Figure 12.5: Visualization of kNN based clustering approaches on synthetic datasets and the ARI score under different k values.

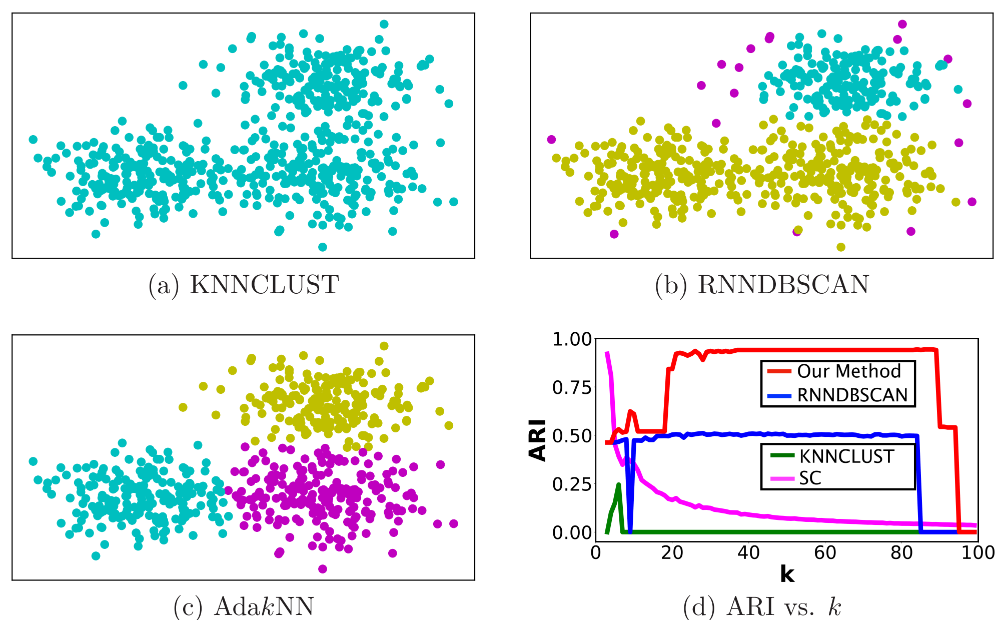
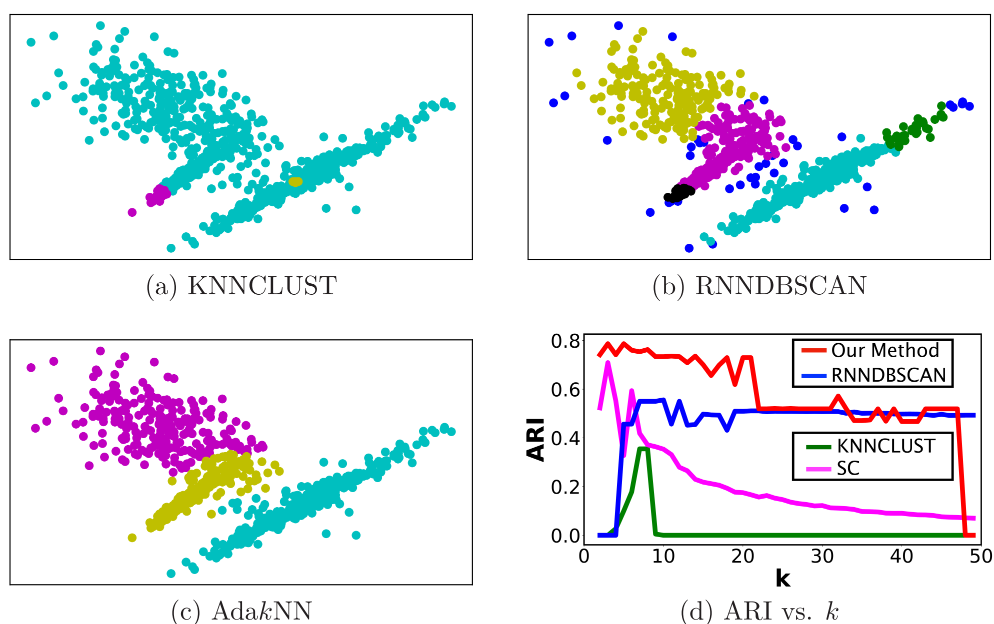


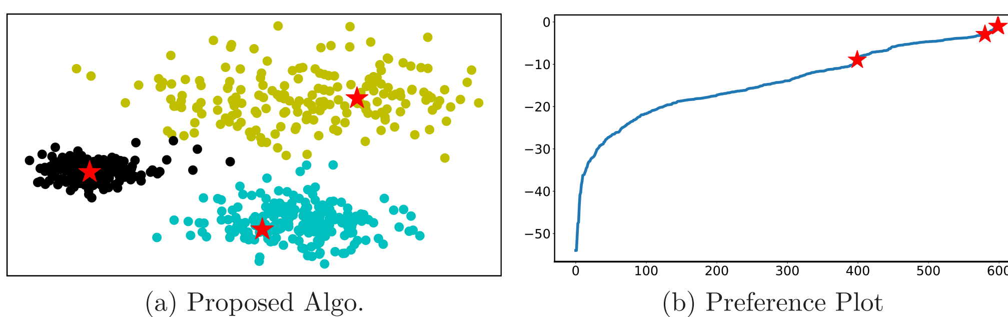
Figure 12.6: Visualization of kNN based clustering approaches on synthetic datasets and the ARI score under different k values.



(a) KNNCLUST (b) RNNDBSCAN

(c) Ada$k$NN (d) ARI vs. k

Figure 12.7: Visualization of kNN based clustering approaches on synthetic datasets and the ARI score under different k values.



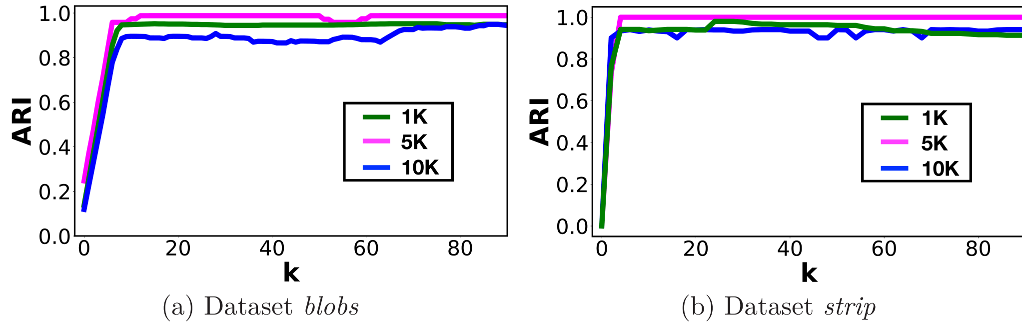
(a) Proposed Algo. (b) Preference Plot

Figure 12.8: Cluster centers (red stars) and their preference values in the dataset.

the quality does not differ a lot under different sizes.

Cluster Number and Parameter k Estimation

There is no ground truth in the real-word for unsupervised learning. Thus, even if we can try different settings, we do not know which one is the best. Some unsupervised cluster quality measures, such as the silhouette coefficient



(a) Dataset *blobs* (b) Dataset *strip*

Figure 12.9: ARI Performance vs k on *blobs* and *strip* dataset with different sizes.

[87], are introduced to address the problem. However, those methods can only be applied to model-based algorithms such as $k$means, and it is challenging to estimate the qualities of arbitrary shaped clusters.

Note that the quality of our kNN clustering approaches is relatively high on a wide range of parameters. Such property can be used to estimate cluster numbers and the best parameter setting in the real-world [15]. Our approach also keeps such benefits. Figure 12.10 illustrates the histogram of cluster number been identified with respect to the value of parameter k for $k \in [0, 100]$. Red and blue lines are the corresponding best and worst ARI scores in each cluster number. As shown in Figure 12.10, for $k \in [1, 100]$, the number of clusters concentrates to a few values. Thus, the real cluster number can be estimated as the most frequent cluster number, and the value of k that leads to the most frequent cluster number can be considered as a good setting.

12.6 Conclusion

In this work, we present a novel kNN based clustering algorithm. Although we are not DBSCAN-alike, we still keep the ability of arbitrary shape clustering using the only parameter k. Similarities and preferences of points are measured as probabilities based on heuristics from DPC and kNN distance. Affinity Propagation algorithm is employed to maximize the overall intra-cluster similarity. In consequence, our kNN based clustering approach

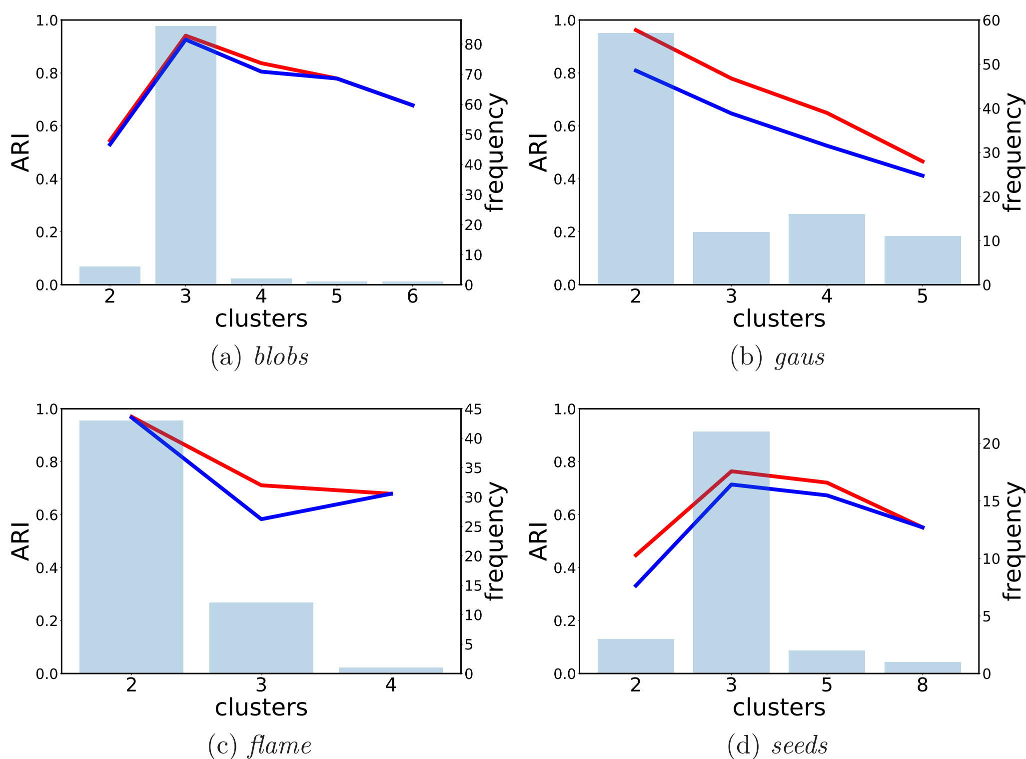
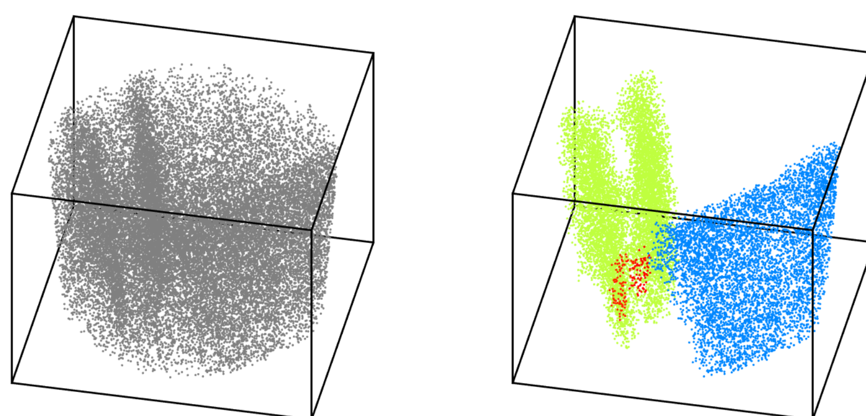


Figure 12.10: Histogram of cluster number identified by our approach, $k \in [1, 100]$. The best (red) and the worst (blue) ARI score are illustrated for each case.

is also aware of the overall distribution in the dataset. In cases where we need to cluster a series of datasets with similar density but different shapes, our approach is particularly useful since it can adapt to changes in shapes automatically. Furthermore, with all those functional benefits, our approach still provides a reliable clustering quality.

Chapter 13

Extremely Noisy Datasets Clustering



(a) Raw data (b) Extracted concentrated atoms

Figure 13.1: The clustering result of our KENClus algorithm a real-world example.

In this chapter, a novel kNN-based clustering algorithm is proposed for an extremely noisy datasets. In-cluster and noise data points are identified by analyzing the k nearest neighbor (kNN) statistics of each data point. A DBSCAN-alike clustering algorithm based on kNN and reverse kNN (RkNN)

is then applied to generate final clusters. Practically, k is the only parameter involved in the whole clustering process. Other parameters are fixed at a default value. For instance, the statistical significant level of unimodality test α is fixed at 0.05. Intensive experiments on both synthetic and real-world datasets with different noise quantities and noise densities show that the approach can provide better clustering quality under strong noise settings. As shown in Figure 13.1, our method works well in real world applications.

13.1 Related Works

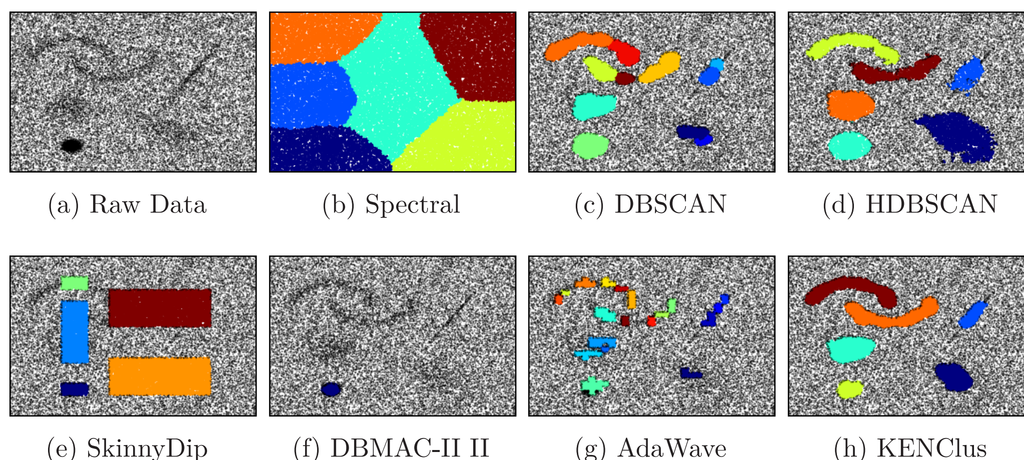


Figure 13.2: Clustering results on extremely noisy datasets. Here, the results of DBSCAN and HDBSCAN are promising because we put huge efforts to obtain the correct parameters.

Figure 13.2 illustrates the clustering result of some related works on an example dataset. Partition-based clustering approaches, such as $k$means [69], EM with Gaussian mixture model[30] and spectral clustering [110], are not suitable for this task. They partition the dataset so that each data point is assigned to one cluster. The existence of noise is completely ignored [10].

The famous density-based clustering algorithm, DBSCAN [34], can be a good solution, as it is designed to identify arbitrarily shaped clusters and noise-robust. However, tuning the two parameters of DBSCAN to achieve

its best performance is not easy under a strong noise setting. HDBSCAN [18] tackles this problem by reducing the number of parameters to one. It is a hierarchical clustering algorithm that only requires a single parameter, the minimum cluster size m_{clSize}. The non-hierarchical clustering result is generated by maximizing the total link area in the dendrogram. However, its performance is not very promising when the quantity of noise points is enormous.

Other clustering techniques such as OPTICS [7], CURE [42], SYNC [12], FOSSCLU [40] and RNNDBSCAN [15] are also only effective on low noise level datasets. A serious performance degradation, or a difficult parameter tuning process, is inevitable. Use an existing outlier detection algorithm, such as LOF [13], to remove noise and then perform clustering is also not possible since noises in an extremely noisy dataset are not unusual objects.

Recently, three algorithms, SkinnyDip [79], DBMAC-II [111], and Adawave [28], are proposed to tackle the clustering problem on an extremely noisy dataset with less parameter tuning difficulties. SkinnyDip and AdaWave are practically parameter-free. DBMAC-II removes noise first and then use another clustering algorithm, such as DBSCAN, to generate final clusters, which is easier than directly cluster the original dataset. However, as shown in Figure 13.2, their performances are not promising. Their clustering results are even worse than the fine-tuned DBSCAN.

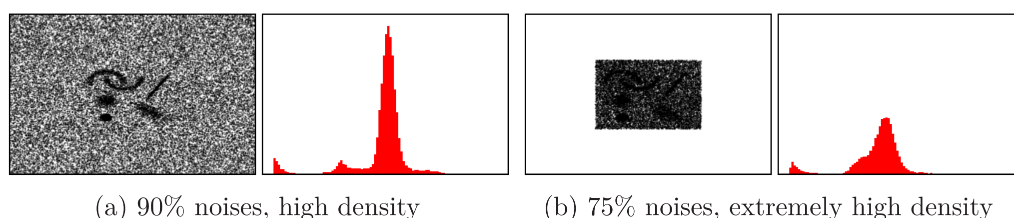
13.2 KENClus Clustering Algorithm

Here we propose a novel ***k*** Nearest Neighbor based **E**xtremely **N**oisy Dataset **Clus**tering algorithm, named as KENClus. It consists of two steps: noises identification and clusters identification. We assume that noises are uniformly distributed in the background with densities slightly smaller than clusters. For noise identification, the kNN distance (k-dist) is used to define the local-density of each data point. Noises are then identified by analyzing the overall local-density distribution of all data points in the dataset. For cluster identification, we design a novel algorithm based on kNN and RkNN. Practically, the whole process only requires one parameter k. Though all dis-

tance functions (metric or non-metric) can be used, the Euclidean distance is employed for all results produced in this work.

Noise Identification

Noise identification is the main challenge in extremely noisy dataset clustering. The large quantity and the high density of noises make noise identification difficult. Figure 13.3 shows two datasets under different noise settings. Their k-dist histograms are also illustrated. Regions with similar densities form a peak in the histogram. Noises form the right-most one. A large number of noises lead to a high peak, which makes other peaks formed by clusters invisible. In this case, clustering algorithms that optimize an objective function, such as HDBSCAN, encounter difficulties. Furthermore, a large number of noises might also form dense areas due to randomness. Parameters of conventional clustering algorithms such as DBSCAN have to be selected carefully to filter out those areas. In the case of high noise density, the peak of low-density clusters is covered by the peak of noises. This situation is also difficult for algorithms such as DBSCAN to determine the correct density threshold. Our KENClus is robust in both cases.



(a) 90% noises, high density (b) 75% noises, extremely high density

Figure 13.3: Datasets under different noise settings and their k-dist histogram.

Local-Density using Neighbors' k-dist

Local-density is employed to identify noises. k-dist is commonly used to represent the local-density of data points in density-based clustering algorithms. The larger k-dist value, the lower the local-density of a data point. Ideally,

the k-dist of an in-cluster point is always smaller than a noise point. However, the local-density contrast between in-cluster and noise data points is small under strong noise settings. Using k-dist alone is less effective.

In KENClus, we assume that most neighbors of an in-cluster data point should also be in-cluster data points. Therefore, kNNs' information of a data point should also be involved when computing its local-density. We first compute the *self-density* of a data point using the distances to its all kNNs. Then the local-density of the data point is then computed using its own and its neighbors' self-densities.

Formally speaking, let X be the dataset with n data points, $\mathcal{N}_k(x_i)$ be the set of kNNs of data point $x_i \in X$. A data point is also a neighbor of itself, i.e., $x_i \in \mathcal{N}_k(x_i)$. The local-density of x_i, denoted as $den[i]$, is computed using the self-density of all data points in $\mathcal{N}_k(x_i)$, as shown in Algorithm 13.1.

Algorithm 13.1: LocalDensity

Input: Dataset X, Parameter k

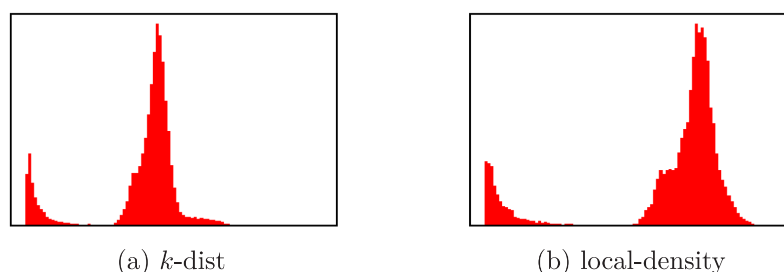
Output: Local Density List den

1 $D_1 \leftarrow k$NN-Query(X, k) ; // D_1: Distances, $n \times k$ matrix
2 $D_2 \leftarrow$ SVD$(D_1, dim = 1)$; // D_2: Self-density, $n \times 1$ vector
3 $D_3 \leftarrow D_2[\mathcal{N}_k]$; // D_3: Neighbors' density, $n \times k$ matrix
4 $den \leftarrow$ SVD$(D_3, dim = 1)$; // den: Local density, $n \times 1$ vector
5 **return** den

The first step is to run kNN query, which returns a $n \times k$ matrix D_1. Row i of the matrix contains distances between x_i and $\forall x_j \in \mathcal{N}_k(x_i)$, in ascending order. Then we compute the vector D_2 where $D_2[i]$, is the self-density of x_i, by reducing the dimensionality of D_1 to 1 using SVD (step 2). As a result, the self-density of x_i includes distances to all neighbors in $\mathcal{N}_k(x_i)$. We do not use k-dist alone since it is not robust. For example, if the selected k is too large, all data points may share a similar k-dist value. Finally, we store all neighbors' self-density in D_3 and using SVD again to compute the local-density vector den.

Figure 13.4 demonstrates the difference between k-dist and our local-

density. In the histogram of k-dist, the peak formed by low-density clusters is fully covered by noises. However, the peak in the histogram of local-density is clear.



(a) k-dist (b) local-density

Figure 13.4: Comparison of histograms k-dist and our local-density with respect to the dataset in Figure 13.2a.

Note that our local-density is defined based on k-dist, which means the larger absolute value, the lower density. To avoid confusion, the term "*left*" and "*right*" on the histogram are used to represent higher and lower local-density respectively.

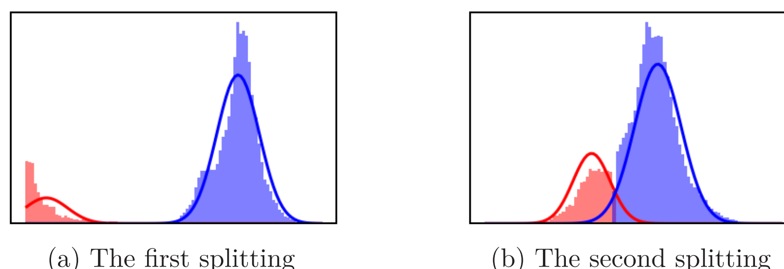
Noise Identification by Recursive Splitting

Noises can be identified by finding the right-most peak of the local-density histogram. As each peak is a cluster in the 1-dimensional local-density space, clustering algorithms such as $k$means and EM can be applied to find peaks. Since we do not know the number of peaks, a parameter-free strategy is necessary.

Here we develop a recursive splitting method. The idea is to split the local-density space at a valley in the histogram at each step. The left part has a higher density and is formed by in-cluster points. The right part contains both in-cluster data points and noises, and therefore must be split further. Figure 13.5 demonstrates two splitting steps.

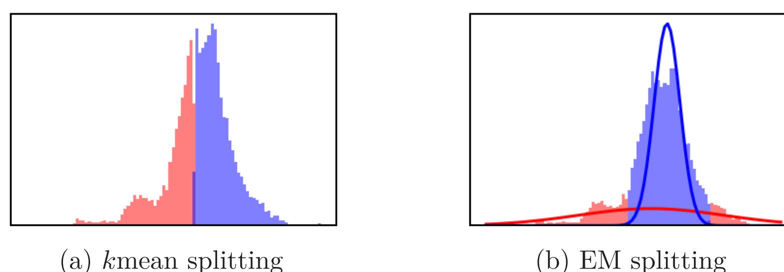
In designing the recursive splitting algorithm, two problems have to be solved: 1) where to split and 2) when to stop.

Previous works such as DBMAC-II [111] also contains a recursive splitting step where $k$means is employed to find the splitting position in each step.



(a) The first splitting (b) The second splitting

Figure 13.5: Each splitting step divides the local-density histogram into two parts. The right part (blue) is used for the next splitting.



(a) $k$mean splitting (b) EM splitting

Figure 13.6: Splitting local-density histogram using $k$means and EM algorithms.

However, directly using $k$means for splitting is not ideal. As shown in Figure 13.6, $k$means may split local-densities at the center of a cluster since it is bad at handling clusters of various sizes and densities. Clusters found by the EM algorithm are not consecutive so that there is no splitting position. We address the problem by adapting the EM algorithm. The initial splitting position is set to the kth smallest local-density value (Algorithm 13.2, step 17). In each iteration, the intersect position of two Gaussians is computed (step 14-20). Objects on each side are fully assigned to each Gaussian model. The process is repeated until the splitting position is not changed or a maximum iteration limit is reached. As shown in Figure 13.5 and 13.6, our approach works better in finding the splitting position.

The recursive splitting process is stopped when there is only one peak left. The Dip-test[52], which is a parameter-free statistical test of unimodality, is employed to test if there is only one peak. If the p-value of the Dip-test is less

than the given significant level α, the null hypothesis of unimodal is rejected, which implies that there are more than one peaks left and further splittings are still necessary (step 2). The parameter α is fixed at 0.05. When the Dip-test detects only one peak, we still need to try one more splitting (step 5-8). The reason is that the peak formed by low-density clusters and covered by the noise peak might note be recognized by the Dip-test. Therefore, we execute one more splitting and check if the splitting position is valid (step 7).

Algorithm 13.2: NoiseIdentification

Input: Local-density List den, Significant level α, Maximum Iteration $maxIter$, Stop Threshold ϵ, Parameter k

Output: Noise Set $\mathcal{R}$

1. $den' \leftarrow den$; `// Make a copy`
2. **while** `Dip-test`$(den') < \alpha$ **do**
3. $\quad splitPos \leftarrow$ `Split`$(den', maxIter, \epsilon, k)$;
4. $\quad den' \leftarrow den[den \geq splitPos]$; `// The right part`
5. **end**

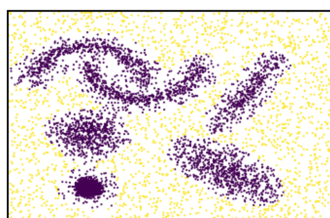
   `/* Test if one more splitting is necessary */`
6. $splitPos' \leftarrow$ `Split`$(den, maxIter, \epsilon, k)$;
7. $N(\mu_l, \sigma_l), N(\mu_r, \sigma_r), w_l, w_r \leftarrow$Gaussian Mixture Model wrt. $splitPos'$;
8. **if** $f_{N(\mu_l,\sigma_l)}(\mu_l) < f_{N(\mu_r,\sigma_r)}(\mu_r)$ **then**
9. $\quad splitPos \leftarrow splitPos'$; `// Accept the splitting`
10. **end**
11. **foreach** $den[i] \in den, den[i] < splitPos$ **do**
12. $\quad \mathcal{R} \leftarrow \mathcal{R} \cup x_i$;
13. **end**
14. **return** $\mathcal{R}$

Finally, the noise set $\mathcal{R}$ is returned (step 8-10). Figure 13.7 shows the noise identification results on our running example. The result is promising but some errors still exist. Therefore, a refinement strategy is necessary during the cluster generation step.

Algorithm 13.3: Local-Density Splitting

```
1  Function Split(den, maxIter, ε, k)
2      splitPos ← kth smallest value in den′ ;        // Initial split
        position
3      while iter < maxIter do
4          den_l ← den[den < splitPos] ;
5          den_r ← den[den ≥ splitPos] ;
6          N(μ_l, σ_l) ←Gaussian fit on den_l ;
7          N(μ_r, σ_r) ←Gaussian fit on den_r ;
8          w_l, w_r ←Normalized weight of two Gaussians ;
           /* Compute the intersect position                  */
9          a ← 1/(2σ_l^2) − 1/(2σ_r^2), b ← μ_r/σ_r^2 − μ_l/σ_l^2, c ← μ_l^2/(2σ_l^2) − μ_r^2/(2σ_r^2) − ln(w_lσ_r/(w_rσ_l)) ;
10         splitPos′ ←The root of ax^2 + bx + c closer to (μ_l + μ_r)/2 ;
11         if |splitPos′ − splitPos| < ε then
12             break;
13         else
14             splitPos ← splitPos′;
15         end
16     end
17     return splitPos
18 end
```

Line 9: $a \leftarrow \frac{1}{2\sigma_l^2} - \frac{1}{2\sigma_r^2},\ b \leftarrow \frac{\mu_r}{\sigma_r^2} - \frac{\mu_l}{\sigma_l^2},\ c \leftarrow \frac{\mu_l^2}{2\sigma_l^2} - \frac{\mu_r^2}{2\sigma_r^2} - \ln(\frac{w_l\sigma_r}{w_r\sigma_l})$;



(a) High noise

(b) Extremely high noise

Figure 13.7: Noise identification results on high and extremely high noise datasets. Some noises that are close to clusters or locate in small but dense areas due to randomness are identified as in-cluster points. Further refinement is necessary.

Noise Identification Complexity

Local density computation involves dimensionality reduction (SVD) on a $n \times k$ matrix, which has a complexity of $O(k^2 n)$. Each split iteration takes $O(n)$

distance computations. Let m be the number of peaks, the split operation is expected to be called for $O(\log m)$ times. Therefore, the overall complexity of noise identification is $O(k^2 n + In \log m)$, where I is the maximum number of iterations of splitting.

Clusters Identification

We follow a general density-based clustering algorithm paradigm to identify clusters: 1) identify core and border points and 2) connect consecutive core points and their neighboring border points together to form clusters.

Core and Border Points

The concept of core and border points are widely used in density-based clustering algorithms. A core point and its border points form the smallest unit of a cluster. In this work, we use reverse k nearest neighbor (RkNN) to define core points.

Definition 13.2.1 (Reverse k Nearest Neighbor)
Given the dataset X, let $\mathcal{N}_k(x_i)$ be the set of kNN of $x_i \in X$. The set of reverse k nearest neighbors (RkNN) of x_i, denoted as $\mathcal{N}_{Rk}(x_i)$, is the set of point $x_j \in X$ such that $x_i \in \mathcal{N}_k(x_j)$.

The size of RkNN set varies across different data points. Given x_i, $|\mathcal{N}_{Rk}(x_i)|$ measures how many data points are "similar" to x_i. If many other in-cluster points are similar to x_i, we say x_i is a core point.

Definition 13.2.2 (Core Points)
Given the dataset X, a data point $x_i \in X$ is a core point if $x_i \notin \mathcal{R}$ and the number of in-cluster points in its RkNN set is larger than k: $|\{x_j | x_j \in \mathcal{N}_{Rk}(x_i) \wedge x_j \notin \mathcal{R}\}| \geq k$.

Definition 13.2.3 (Border Points)
Given the dataset X, a data point $x_i \in X$ is a border point if $x_i \notin \mathcal{R}$ and the number of in-cluster points in its RkNN set is less than k: $|\{x_j | x_j \in \mathcal{N}_{Rk}(x_i) \wedge x_j \notin \mathcal{R}\}| < k$.

The definition of core and border points is important for noise refinement. A noise been mistakenly identified as in-cluster points will be labeled as a border point and re-classified as noise if it is not assigned to any clusters later.

Clusters

The concept of *directly density-reachable*, *density-reachable* and *density-connected* are defined to describe clusters. Let $\mathcal{Q}$ be the set of core points:

Definition 13.2.4 (Directly Density-reachable)
A point $x_j \in X$ is directly density-reachable from another point $x_i \in X$, denoted as $x_i \to x_j$, if $x_j \notin \mathcal{R}$ and:

1. $x_j \in \mathcal{N}_k(x_i)$, where $x_i \in \mathcal{Q}$ or
2. $x_j \in \mathcal{N}_{Rk}(x_i)$, where $x_i \in \mathcal{Q} \wedge x_j \in \mathcal{Q}$

Note that the kNN relationship is not symmetric, i.e., $x_j \in \mathcal{N}_k(x_i) \not\Leftrightarrow x_i \in \mathcal{N}_k(x_j)$. The second condition includes core points in RkNNs, which guarantees the symmetry of directly density-reachable between core points. However, when a border point is involved, directly density-reachable is not symmetric.

Definition 13.2.5 (Density-reachable)
A point $x_j \in X$ is density-reachable from another point $x_i \in X$, denoted as $x_i \rightrightarrows x_j$, if there exists a chain of points $x_1, x_2, \ldots, x_{n-1}, x_n$, where $x_1 = x_i$, $x_n = x_j$, $\forall m \in [1, n-1]$, $x_m \to x_{m+1}$.

Density-reachable extends the definition of directly density-reachable by making it transitive. It is symmetric for core points.

Definition 13.2.6 (Density-connected)
A point $x_j \in X$ is density-connected to another point $x_i \in X$, denoted as $x_i \rightleftarrows x_j$, if there is a point $x_n \in X$ such that $x_n \rightrightarrows x_i, x_n \rightrightarrows x_j$.

Obviously, density-connected is symmetric.

Definition 13.2.7 (Cluster)
A cluster C in the dataset X is a non-empty subset of X such that:

1. $\forall x_i, x_j \in X$, if $x_i \in C$ and $x_i \rightrightarrows x_j$, then $x_j \in C$.

2. $\forall x_i, x_j \in C$, $x_i \rightleftarrows x_j$.

3. for a border point $x_j \in X$ that is not density-reachable from any core point, we have $x_j \in C$ if $\exists x_i \in C$ such that:

   (a) x_i is a core point and $dist(x_i, x_j) \leq \max_{x_n, x_m \in \{C \cap \mathcal{Q}\} \wedge x_n \rightarrow x_m} dist(x_n, x_m)$.

   (b) $\nexists x'_i \in \mathcal{N}_k(x_j)$ such that $x'_i \in C' \neq C$ and the condition (a) holds, while $dist(x'_i, x_j) < dist(x_i, x_j)$

The first two criteria in the definition above describe the maximality and the connectivity property of a cluster. However, there are border points that are very close to a cluster but not directly density-reachable from any core points, since the parameter k might be too small for some core points in a highly dense region. The first two criteria do not cover such kind of border points. We address the problem by introducing the last criterion, which adds a border point to cluster C if its distance to a core point in C is smaller than the maximum distance between any two directly density-reachable core points in C.

Algorithm 13.4 illustrates the cluster generation process. L is the list of cluster labels. $L[i] = -1$ indicates that x_i is a noise. $L[i] = 0$ means that x_i has not been processed yet. A cluster is generated starting from a core point (step 8-10) and expanding to neighbors recursively (Step 15-28). According to the definition of directly density-reachable, neighbors of a core point x_i include the set of its kNN and core points in the set of its RkNN (step 17, 23). A border point that is not density-reachable from any core points in any cluster is labeled as noise (Step 12). Finally, all clusters are expanded by including nearby border points that are close enough (step 13). Algorithm 13.6 is the cluster expanding process that assigns border points that fulfill the 3rd criterion in Definition 13.2.7 to the corresponding clusters.

Algorithm 13.4: GenerateClusters

```
   Input: Dataset X, Core set Q, Noise set R
   Output: Cluster Label List L
 1 foreach x_i ∈ X do
 2 |  if x_i ∈ R then
 3 |  |  L[i] ← −1 ;                                   // Noise
 4 |  else
 5 |  |  L[i] ← 0 ;                               // UNPROCESSED
 6 |  end
 7 end
 8 label ← 1 ;                         // Cluster label starts from 1
 9 foreach x_i ∈ X ∧ L[i] = 0 do
10 |  if x_i ∈ Q then
11 |  |  GenerateCluster(x_i, label, L, Q, R) ;
12 |  |  label ← label + 1;
13 |  else
14 |  |  L[i] ← −1 ;                         // Border become noise
15 |  end
16 end
17 ExpandCluster(X, L) ;
18 return L ;
```

Cluster Identification Complexity

Cluster identification is a kNN based clustering algorithm with a DBSCAN-alike structure. A kNN query is performed on each in-cluster data points. If the complexity of kNN query is $O(\log n)$ with an indexing structure, the average complexity of cluster identification is $O(n \log n + |\mathcal{Q}|^2)$, where $|\mathcal{Q}|^2$ is the additional component for cluster expanding, which is quadratic to the number of core points.

13.3 Experiments

Experiments are conducted on both synthetic and real datasets with different manually controlled noise levels. Details about the setting of noise are shown below. We use the Adjusted Mutual Information (AMI) score [99], which is

Algorithm 13.5: Generate Cluster

```
1  Function GenerateCluster(x_i, label, L, Q, R)
2      L[i] ← label;
3      q ← {Neighbors of x_i} ;
4      foreach x_j ∈ q do
5          L[j] ← label;
6      end
7      while Q is not empty do
8          x_j ← q.pop() ;
9          if x_j ∈ Q then
10             foreach x_n ∈ {Neighbors of x_j} do
11                 if L[n] = 0 then
12                     q.append(x_n) ;
13                     L[n] ← label ;
14                 else if L[n] = −1 ∧ x_n ∉ R then
15                     L[n] ← label;
16                 end
17             end
18         end
19     end
20 end
```

a standard metric ranging from 0 at worst and 1 at best, to evaluate the performance. Noises are included in computing the AMI score.

Seven different algorithms are selected as baselines for comparison. We begin with DBSCAN and HDBSCAN, which are the most commonly used conventional density-based clustering algorithms. SkinnyDip, DBMAC-II and AdaWave are the most recent approaches aimed to be robust on extremely noisy datasets. *k*means and EM are widely known centroid and model based clustering algorithms. We use the implementation of DBSCAN, *k*means and EM in the scikit-learn library [83]. The HDBSCAN implementation can be found in [80]. SkinnyDip [79] and AdaWave [28] provide the source code in their original works. We implement DBMAC-II and our KEN-Clus using Python. The final clustering result of DBMAC-II is generated using DBSCAN. The implementation can be found in our supplementary material.

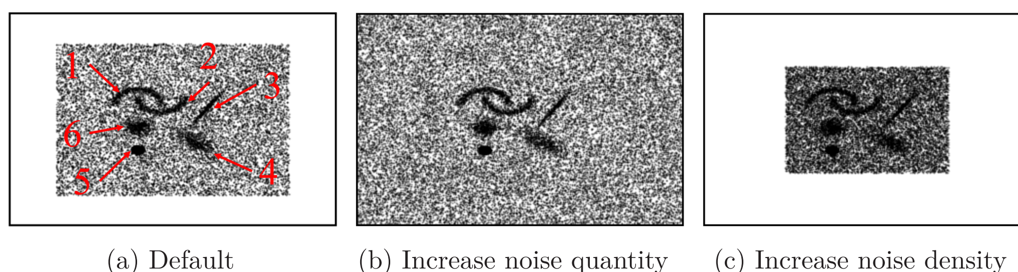
Algorithm 13.6: ExpandCluster

```
    Input: Dataset X, Core set Q, Noise set R,
    Cluster Label List L
    Output: Cluster Label List L
    /* Maximum distance between directly density-reachable
       core points in each cluster                              */
 1  foreach Unique label ∈ L do
 2  |   C ← {x_i ∈ X | L[i] = label} ;
 3  |   d_max[label] ← max_{x_n,x_m ∈ C ∧ x_n,x_m ∈ Q ∧ x_n → x_m} dist(x_n, x_m) ;
 4  end
    /* For each border point not in a cluster                   */
 5  foreach x_i ∈ X ∧ x_i ∉ R ∧ L[i] = −1 do
 6  |   mindist ← ∞;
 7  |   foreach x_j ∈ N_k(x_i), x_j is a core point do
 8  |   |   label ← L[j];
 9  |   |   if dist(x_i, x_j) ≤ min(d_max[label], mindist) then
10  |   |   |   mindist ← dist(x_i, x_j);
11  |   |   |   L[i] ← label;
12  |   |   end
13  |   end
14  end
```

We put great effort on parameter tuning for DBSCAN. We select *minpts* from the set $\{5, 10, 15, \ldots, 100\}$. The value of ϵ is tested over the *minpts* nearest neighbor distance of every data point in the dataset. The parameter of HDBSCAN, minimum cluster size m_{clSize}, is selected from the set of $\{5, 10, 15, \ldots, 500\}$. SkinnyDip is practically parameter-free, with significant level α fixed at 0.05. We manually adjust the 4 parameters (starting/ending radius, step size and significant level) of DBMAC-II for noise identification, and then reuse the best parameters of DBSCAN to generate clusters. AdaWave has one parameter *scale* to control the resolution of the grid. We follow the suggestion in the original work [28] and set *scale* $= 128$. For $k$means and EM, the correct cluster number is given.

Synthetic Dataset

In the synthetic dataset, we try to mimic the general situation of extremely noisy dataset clustering and investigate how the quantity and the density of noise affect clustering performance. As shown in Figure 13.8, the synthetic dataset has 2 dimensions with 6 clusters. Clusters 1, 2 are two interleaving half circles with 1000 data points and Gaussian noise ($\sigma = 0.1$). Clusters 3, 4, 5, 6 are Gaussian distributed with 300, 500, 2000 and 500 data points respectively. Noises are uniformly distributed in the background. Noises that locate inside a cluster are considered as in-cluster points. In the default synthetic dataset, 75% data points are noises. The noise density is represented as the average local-density ratio between the most sparse cluster and noises, which is set to 2.0 by default. The size of the background is adjusted accordingly (Figure 13.8b and 13.8c) so that we can change the noise quantity without affecting density, and vice versa. As synthetic datasets are randomly generated, each experiment is repeated for 3 times and the average AMI score is reported.

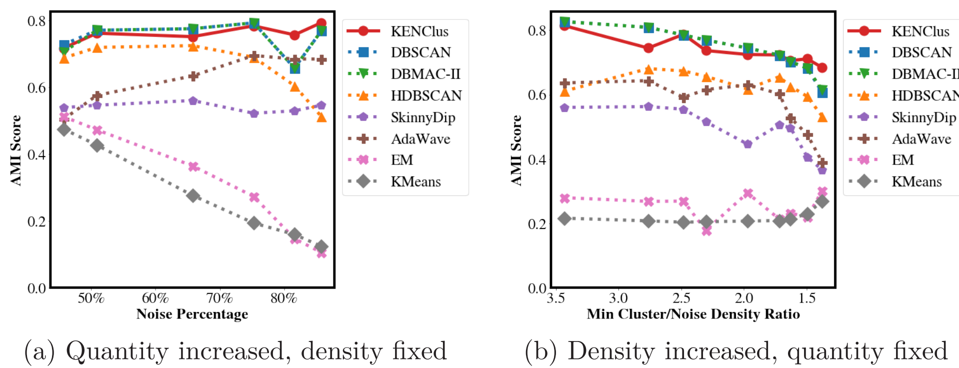


(a) Default (b) Increase noise quantity (c) Increase noise density

Figure 13.8: Synthetic dataset examples. The noise density and the noise quantity are controlled independently.

The experimental results on varies noise quantities and noise densities are shown in Figure 13.9. $k$means and EM are the worst since they do not consider noise at all. SkinnyDip and AdaWave are better as they are designed for high noise dataset. Since SkinnyDip only generates rectangular shaped clusters, it is worse than AdaWave. The conventional method, DBSCAN, outperforms SkinnyDip and AdaWave. However, the time we spend on parameter tuning for DBSCAN is also much more than that of the oth-

ers. DBMAC-II is a noise removing algorithm. Its performance here is the same as DBSCAN since DBSCAN is used to generate clusters after noises been removed. Although DBMAC-II aims to ease the parameter tuning difficulty under strong noise settings, the 4 parameters for noise removing are not easy to set. HDBSCAN is slightly worse than DBSCAN in most cases. Our KENClus algorithm, with only one parameter, is among the best under all circumstances.



(a) Quantity increased, density fixed (b) Density increased, quantity fixed

Figure 13.9: Experimental results on synthatic datasets.

It is worth to note that noise quantity and noise density have different impacts on the performance of clustering algorithms, as the asymptotic trends of clustering quality are different between two experiments series. The AMI score of KENClus, DBSCAN, DBMAC-II, SkinnyDip and AdaWave is stable with large noise quantity but decreasing with high noise density. The reason is that those algorithms detect noises based on the density of data points. High quantity of noise does not affect the density contrast between in-cluster and noise data points. However, high noise density leads to a small density contrast, which makes it more difficult to distinguish between in-cluster and noise data points. The performance of $k$means and EM are stable with respect to noise densities since all noises are wrongly classified. As long as the quantity of noise is not increased, the AMI score is not affected. HDBSCAN is a hierarchical clustering algorithm. When the noise level is increasing (quantity or density), it tends to classify noises as low-density clusters. In general, our KENClus is the most stable one with respect to changes in both

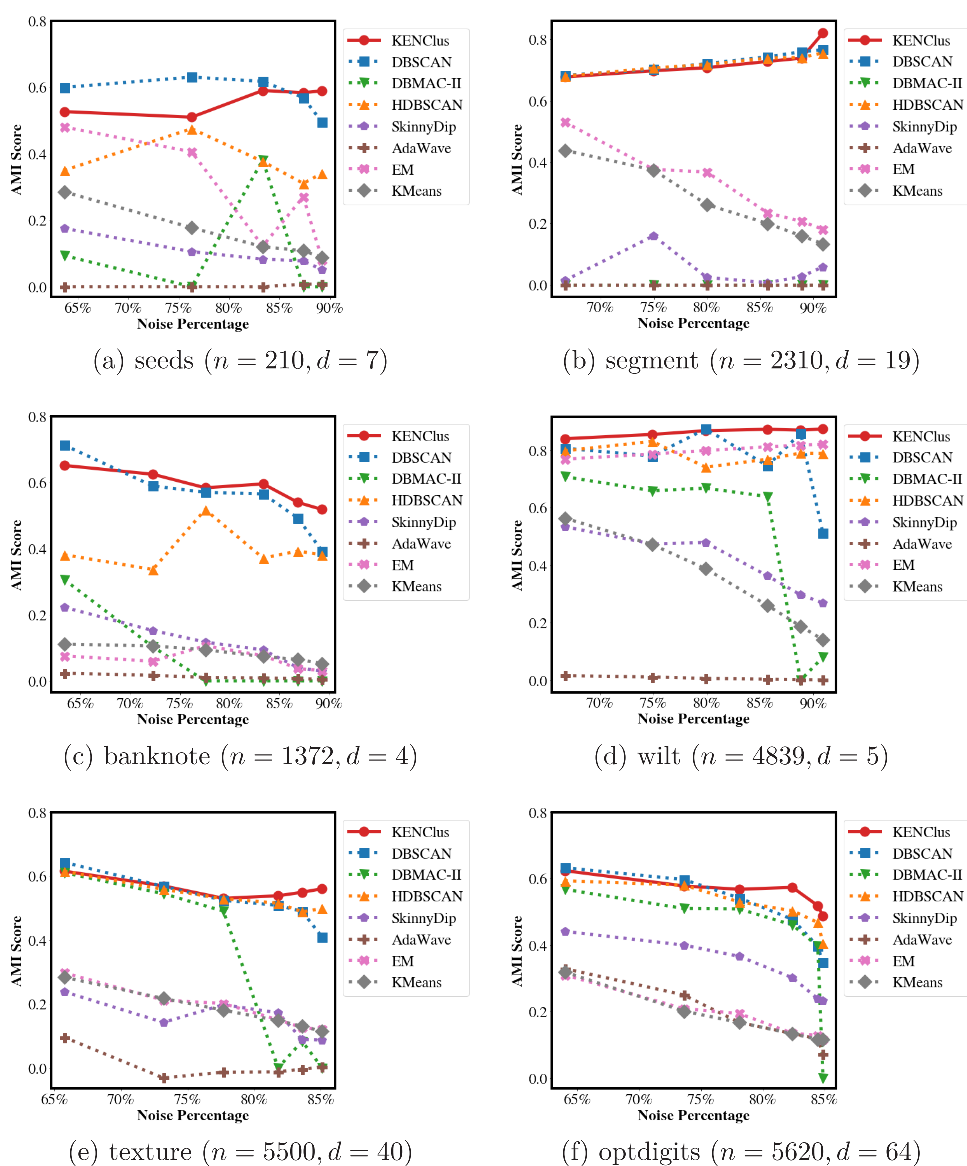
noise quantity and noise density.

Real-world Dataset

Six real-world datasets obtained from the UCI repository are tested, as shown in Figure 13.10. Note that in these classification-style datasets, every point is assigned to a semantic class label, i.e., none of those datasets have noise. For this reason, we manually add uniformly distributed noises for testing. Firstly, all dimensions of each dataset are normalized to $[0, 1]$. Then, noises that are uniformly distributed in the space $[0, 1]^d$ are added to the dataset. We investigate the performance on different noise levels by adjusting the number of noises been added. As the background area size is fixed at 1, both noise quantity and noise density are changed accordingly. Noises that locate inside a cluster is treated as in-cluster data points. As noises are generated randomly, each experiment is repeated for 3 times and the average AMI score is reported. SkinnyDip contains a dimensionality reduction step. For fair comparison, we use t-SNE [77] to reduce the dimensionality of the two high dimensional datasets (`texture, optdigits`) to 3 for all other algorithms. Experimental results are illustrated in Figure 13.10, where n is the number of data points without noise and d is the dimensionality of the data.

In general, the results of our KENClus approach are promising on all real-world datasets and all noise levels. In particular, when the noise level is high (noise percentage $> 80\%$), it outperforms all other baselines. The conventional method, DBSCAN, is also very competitive. It outperforms more recent noise robust clustering algorithms. However, the parameter tuning step is time-consuming. Different from the results on synthetic dataset, the performance of DBMAC-II on real-world datasets falls behind DBSCAN. This is because its noise removing step becomes unstable under high dimensional settings. SkinnyDip and AdaWave are parameter-free clustering algorithms designed for high noise dataset. However, they show unsatisfactory results on real-world datasets with noise. In summary, our KENClus is a very competitive clustering algorithm for extremely noisy dataset clustering. Its clustering quality is better than the fine tuned DBSCAN, and most

importantly, only one parameter is required.



(a) seeds ($n = 210, d = 7$)

(b) segment ($n = 2310, d = 19$)

(c) banknote ($n = 1372, d = 4$)

(d) wilt ($n = 4839, d = 5$)

(e) texture ($n = 5500, d = 40$)

(f) optdigits ($n = 5620, d = 64$)

Figure 13.10: Experimental results on real datasets with varies background noise levels .

13.4 Conclusion

In this work, we investigate the extremely noisy dataset clustering problem. The huge quantity and the high density of noises in such datasets are challenging for existing clustering algorithms. Through our intensive experiments, surprisingly, we found the conventional method DBSCAN outperforms algorithms that are proposed recently for extremely noisy dataset clustering. However, to achieve such promising results under strong noise settings, a timing consuming parameter tuning step is necessary for DBSCAN. In applications such as scientific research, where there may be hundreds of datasets waiting for processing, such kind of parameter tuning is impractical. To address the problem, we propose our KENClus algorithm, which reduces the complexity of parameter tuning as only one parameter is required. At the same time, it has a promising clustering performance under strong noise settings.

Chapter 14

Summary and Outlook

In this chapter, the major contributions of this thesis are summarized, and possible directions for future research are pointed out.

14.1 Summary

In this thesis, novel algorithms are proposed to solve new challenges under different conditions in different pattern mining tasks, including itemset mining, sequential pattern mining, and clustering. The preliminaries of pattern mining involved in this thesis are introduced at the very beginning. Then, the new challenge of extracting rare patterns in itemset mining is investigated in part I. A novel data structure called the negative itemset tree is designed for top-down rare itemset mining. An efficient rare itemset mining algorithm using the negative itemset tree and the idea of diff-set, called NIIMiner, is proposed in chapter 4. Besides extracting the full list of rare itemsets, the problem of closed rare itemset mining is also conducted, which produces a small condensed representation. In chapter 5, a bi-directional closed rare itemset mining framework is proposed to accelerate the mining process. In chapter 6, an efficient closed rare itemset mining algorithm, called LSCMiner, is proposed.

In part II, existing sequential pattern mining and temporal pattern mining are extended to handle data streams and interval-based events. A general

introduction and preliminaries of sequential and temporal pattern mining are given in chapter 7 and 8. Then, an incremental temporal pattern mining algorithm for data streams is described in chapter 9. Chapter 10 further extends the algorithm to handle interval-based events in data streams.

Part III studies two new challenges, shape alternation adaptability and extremely noisy dataset, in kNN based clustering. Chapter 9 presents a novel kNN based clustering algorithm with shape alternation adaptability. Given a series of datasets, users do not need to tune the parameter manually for each dataset. Chapter 10 proposes a novel kNN based clustering algorithm for extremely noisy datasets.

14.2 Outlook

The last section points out some directions for future research. In this thesis, various novel algorithms are proposed to address new challenges under different conditions in the area of itemset mining, sequential pattern mining, and clustering. One big common challenge in those pattern mining areas is the performance on a big dataset. For example, the rare itemset mining algorithms we proposed in this thesis are orders of magnitude faster conventional methods. However, when the transaction dataset becomes large and dense, it still takes a significant amount of time to generate rare itemsets. One solution to address the performance issue is to design anytime algorithms for pattern mining. An anytime algorithm can return a valid solution to a problem even if it is interrupted before it ends. The longer it keeps running, the better solution it can provide. In recent years, the challenge of developing anytime pattern mining algorithms has caught much attention. There are various anytime frequent itemset mining and sequential pattern mining algorithms [41][57]. Anytime density-based clustering algorithms such as AnyDBC [78] has also been proposed recently. However, those anytime algorithms are designed for normal application scenarios such as frequent itemset mining. New challenges studies in this thesis, such as rare itemset mining and kNN based clustering, are not covered. It is worth developing new antytime pattern mining algorithms.

Besides the common performance issue, each pattern mining tasks studied in this thesis also has its own research problems to be solved. For example, this thesis developed efficient rare itemset mining algorithms. Efficient closed rare itemset mining algorithms are also proposed to achieve better performance and reduce the redundancy of the full rare itemset list. However, existing condensed itemset representations such as the closed pattern are designed for frequent patterns, which will cause problems. For example, as shown in this thesis, an extra frequent border set is necessary to keep the closed pattern representation lossless. Developing condensed representations, either lossless or approximated, for rare itemset mining is necessary.

Furthermore, the interval-based temporal pattern mining algorithm proposed in this thesis is still based on the fundamental sequential pattern mining algorithm, PrefixSpan. Subsequences that support the same temporal pattern must share the same event type order. However, in real-world applications, the log of event labels might not be as robust as temporal information. Mining temporal patterns without categorical label information could be very useful in this case.

Bibliography

[1] M. Adda, L. Wu, and Y. Feng. Rare itemset mining. In *Sixth International Conference on Machine Learning and Applications, 2007. ICMLA 2007.*, pages 73–80. IEEE, 2007.

[2] R. C. Agarwal, C. C. Aggarwal, and V. Prasad. A tree projection algorithm for generation of frequent item sets. *Journal of parallel and Distributed Computing*, 61(3):350–371, 2001.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[4] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering*, pages 3–14. IEEE, 1995.

[5] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[6] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. Optics: Ordering points to identify the clustering structure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–60. ACM, 1999.

[7] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. Optics: ordering points to identify the clustering structure. In *ACM Sigmod record*, volume 28, pages 49–60. ACM, 1999.

[8] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu. Sequential pattern mining using a bitmap representation. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 429–435, 2002.

[9] R. J. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining in large, dense databases. *Data mining and knowledge discovery*, 4(2):217–240, 2000.

[10] S. Ben-David and N. Haghtalab. Clustering in the presence of background noise. In *Proceedings of the 31st International Conference on Machine Learning*, volume 32, pages 280–288, 2014.

[11] Bentley and Friedman. Fast algorithms for constructing minimal spanning trees in coordinate spaces. *IEEE Transactions on Computers*, C-27(2):97–105, Feb 1978.

[12] C. Böhm, C. Plant, J. Shao, and Q. Yang. Clustering by synchronization. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 583–592. ACM, 2010.

[13] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: identifying density-based local outliers. In *ACM sigmod record*, volume 29, pages 93–104. ACM, 2000.

[14] M. Brito, E. Chavez, A. Quiroz, and J. Yukich. Connectivity of the mutual k-nearest-neighbor graph in clustering and outlier detection. *Statistics & Probability Letters*, 35(1):33–42, 1997.

[15] A. Bryant and K. Cios. Rnn-dbscan: A density-based clustering algorithm using reverse nearest neighbor density estimates. *IEEE Transactions on Knowledge and Data Engineering*, 30(6):1109–1121, June 2018.

[16] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *Proceedings. 17th*

International Conference on Data Engineering, 2001., pages 443–452. IEEE, 2001.

[17] L. Cagliero and P. Garza. Infrequent weighted itemset mining using frequent pattern growth. *IEEE Transactions on Knowledge and Data Engineering*, 26(4):903–915, 2014.

[18] R. J. Campello, D. Moulavi, and J. Sander. Density-based clustering based on hierarchical density estimates. In *Pacific-Asia conference on knowledge discovery and data mining*, pages 160–172. Springer, 2013.

[19] F. Cao, M. Estert, W. Qian, and A. Zhou. Density-based clustering over an evolving data stream with noise. In *Proceedings of the 2006 SIAM international conference on data mining*, pages 328–339. SIAM, 2006.

[20] C. Cassisi, A. Ferro, R. Giugno, G. Pigola, and A. Pulvirenti. Enhancing density-based clustering: Parameter reduction and outlier detection. *Information Systems*, 38(3):317–330, 2013.

[21] C.-I. Chang, H.-E. Chueh, and N. P. Lin. Sequential patterns mining with fuzzy time-intervals. In *2009 Sixth International Conference on Fuzzy Systems and Knowledge Discovery*, volume 3, pages 165–169. IEEE, 2009.

[22] L. Chang, T. Wang, D. Yang, and H. Luan. Seqstream: Mining closed sequential patterns over stream sliding windows. In *2008 Eighth IEEE International Conference on Data Mining*, pages 83–92, 2008.

[23] M. H. Chehreghani. Efficient computation of pairwise minimax distance measures. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 799–804. IEEE, 2017.

[24] G. Chen, X. Wu, and X. Zhu. Sequential pattern mining in multiple streams. In *Fifth IEEE International Conference on Data Mining (ICDM'05)*, pages 585–588. IEEE, 2005.

[25] K.-Y. Chen, B. P. Jaysawal, J.-W. Huang, and Y.-B. Wu. Mining frequent Time Interval-based Event with duration patterns from temporal database. In *2014 International Conference on Data Science and Advanced Analytics (DSAA)*, pages 548–554. IEEE, oct 2014.

[26] Y. Chen and T. C. Huang. Discovering fuzzy time-interval sequential patterns in sequence databases. *IEEE Trans. Systems, Man, and Cybernetics, Part B*, 35(5):959–972, 2005.

[27] Y.-C. Chen, W.-C. Peng, and S.-Y. Lee. Mining temporal patterns in time interval-based data. *Transactions on Knowledge and Data Engineering*, 27(12):3318–3331, 2015.

[28] Z. Chen, J. Liu, Y. Deng, K. He, and J. E. Hopcroft. Adaptive wavelet clustering for highly noisy data. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 328–337. IEEE, 2019.

[29] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. D. Ullman, and C. Yang. Finding interesting associations without support pruning. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):64–78, 2001.

[30] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977.

[31] D. Dua and C. Graff. UCI machine learning repository, 2017.

[32] L. Duan, L. Xu, F. Guo, J. Lee, and B. Yan. A local-density based spatial clustering algorithm with noise. *Information systems*, 32(7):978–986, 2007.

[33] L. Ertoz, M. Steinbach, and V. Kumar. A new shared nearest neighbor clustering algorithm and its applications. In *Workshop on clustering high dimensional data and its applications at 2nd SDM*, pages 105–115, 2002.

[34] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press, 1996.

[35] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

[36] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. Knowledge discovery and data mining: Towards a unifying framework. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 82–88. AAAI Press, 1996.

[37] P. Fournier-Viger, J. C.-W. Lin, A. Gomariz, T. Gueniche, A. Soltani, Z. Deng, and H. T. Lam. The spmf open-source data mining library version 2. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 36–40. Springer, 2016.

[38] B. J. Frey and D. Dueck. Clustering by passing messages between data points. *science*, 315(5814):972–976, 2007.

[39] F. Giannotti, M. Nanni, and D. Pedreschi. Efficient mining of temporally annotated sequences. In *Proceedings of the 2006 SIAM international conference on data mining*, pages 348–359. SIAM, 2006.

[40] S. Goebl, X. He, C. Plant, and C. Böhm. Finding the optimal subspace for clustering. In *2014 IEEE International Conference on Data Mining*, pages 130–139. IEEE, 2014.

[41] P. Goyal, J. S. Challa, S. Shrivastava, and N. Goyal. Anyfi: an anytime frequent itemset mining algorithm for data streams. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 942–947. IEEE, 2017.

[42] S. Guha, R. Rastogi, and K. Shim. Cure: An efficient clustering algorithm for large databases. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, page 73–84, New York, NY, USA, 1998. Association for Computing Machinery.

[43] Z. Guo, T. Huang, Z. Cai, and W. Zhu. A new local density for density peak clustering. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 426–438. Springer, 2018.

[44] A. Gupta, A. Mittal, and A. Bhattacharya. Minimally infrequent itemset mining using pattern-growth paradigm and residual trees. In *Proceedings of the 17th International Conference on Management of Data*, page 13. Computer Society of India, 2011.

[45] T. Guyet and R. Quiniou. Mining temporal patterns with quantitative intervals. In *2008 IEEE International Conference on Data Mining Workshops*, pages 218–227. IEEE, 2008.

[46] T. Guyet and R. Quiniou. Extracting temporal patterns from interval-based sequences. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1306, 2011.

[47] T. Guyet and R. Quiniou. Incremental mining of frequent sequences from a window sliding over a stream of itemsets. *Atelier CIDN Classification Incrémentale et Détection de Nouveauté*, 2012.

[48] D. J. Haglin and A. M. Manning. On minimal infrequent itemset mining. In R. Stahlbock, S. F. Crone, and S. Lessmann, editors, *Proceedings of the 2007 International Conference on Data Mining, DMIN 2007, June 25-28, 2007, Las Vegas, Nevada, USA*, pages 141–147. CSREA Press, 2007.

[49] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu. Freespan: frequent pattern-projected sequential pattern mining. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 355–359, 2000.

[50] J. Han, J. Pei, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *proceedings of the 17th international conference on data engineering*, pages 215–224. Citeseer, 2001.

[51] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12, May 2000.

[52] J. A. Hartigan and P. M. Hartigan. The dip test of unimodality. *The annals of Statistics*, 13(1):70–84, 1985.

[53] M. Hassani and T. Seidl. Towards a mobile health context prediction: Sequential pattern mining in multiple streams. In *2011 IEEE 12th Conference on Mobile Data Management*, volume 2, pages 55–57. IEEE, 2011.

[54] Y. Hirate and H. Yamana. Generalized sequential pattern mining with item intervals. *Journal of Computers*, 1(3):51–60, 2006.

[55] C.-c. Ho, H.-f. Li, F.-f. Kuo, and S.-y. Lee. Incremental mining of sequential patterns over a stream sliding window. In *Sixth IEEE International Conference on Data Mining - Workshops (ICDMW'06)*, pages 677–681, 2006.

[56] N. Hoque, B. Nath, and D. Bhattacharyya. An efficient approach on rare association rule mining. In *Proceedings of Seventh International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA 2012)*, pages 193–203. Springer, 2013.

[57] Q. Hu and T. Imielinski. Alpine: Progressive itemset mining with definite guarantees. In *Proceedings of the 2017 SIAM International Conference on Data Mining*, pages 63–71. SIAM, 2017.

[58] T. Jain and M. Kamble. Exante method for mining infrequent itemsets in transactional database. *Archives of Computer Engineering*, 3:1–13, 2017.

[59] R. U. Kiran and P. K. Reddy. An efficient approach to mine rare association rules using maximum items' support constraints. In *British National Conference on Databases*, pages 84–95. Springer, 2010.

[60] Y. S. Koh and S. D. Ravana. Unsupervised rare pattern mining: a survey. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 10(4):45, 2016.

[61] Y. S. Koh and N. Rountree. Finding sporadic rules using apriori-inverse. In *Proceedings of the 9th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, PAKDD'05, page 97–106, Berlin, Heidelberg, 2005. Springer-Verlag.

[62] Y. S. Koh, N. Rountree, and R. A. O'Keefe. Mining interesting imperfectly sporadic rules. *Knowl. Inf. Syst.*, 14(2):179–196, 2008.

[63] J. Z. Kolter and M. J. Johnson. Redd: A public data set for energy disaggregation research. In *Workshop on data mining applications in sustainability (SIGKDD), San Diego, CA*, volume 25, pages 59–62, 2011.

[64] A. Kotsifakos, P. Papapetrou, and V. Athitsos. Ibsm: Interval-based sequence matching. In *2013 SIAM International Conference on Data Mining*, pages 596–604. SIAM, 2013.

[65] J. Lavergne, R. Benton, and V. V. Raghavan. Trarm-relsup: targeted rare association rule mining using itemset trees and the relative support measure. In *International Symposium on Methodologies for Intelligent Systems*, pages 61–70. Springer, 2012.

[66] J. Li, X. Zhang, G. Dong, K. Ramamohanarao, and Q. Sun. Efficient mining of high confidence association rules without support thresholds. In J. M. Zytkow and J. Rauch, editors, *Principles of Data Mining and Knowledge Discovery, Third European Conference, PKDD '99, Prague, Czech Republic, September 15-18, 1999, Proceedings*, volume 1704 of *Lecture Notes in Computer Science*, pages 406–411. Springer, 1999.

[67] B. Liu, W. Hsu, and Y. Ma. Mining association rules with multiple minimum supports. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 337–341. ACM, 1999.

[68] B. Liu, W. Hsu, and Y. Ma. Pruning and summarizing the discovered associations. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 125–134. ACM, 1999.

[69] S. P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28:129–137, 1982.

[70] Y. Lu, M. Hassani, and T. Seidl. Incremental temporal pattern mining using efficient batch-free stream clustering. In *Proceedings of the 29th international conference on scientific and statistical database management*, pages 1–12, 2017.

[71] Y. Lu, F. Richter, and T. Seidl. Efficient infrequent itemset mining using depth-first and top-down lattice traversal. In *Database Systems for Advanced Applications*, pages 908–915, Cham, 2018. Springer International Publishing.

[72] Y. Lu, F. Richter, and T. Seidl. Lscminer: Efficient low support closed itemsets mining. In *International Conference on Web Information Systems Engineering*, pages 293–309. Springer, 2019.

[73] Y. Lu, F. Richter, and T. Seidl. Efficient infrequent pattern mining using negative itemset tree. In *Complex Pattern Mining*, pages 1–16. Springer, 2020.

[74] Y. Lu and T. Seidl. Towards efficient closed infrequent itemset mining using bi-directional traversing. In *2018 IEEE 5th International Conference on Data Science and Advanced Analytics (DSAA)*, pages 140–149, 2018.

[75] Y. Lu, Y. Zhang, F. Richter, and T. Seidl. k-nearest neighbor based clustering with shape alternation adaptivity. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020.

[76] Y. Lv, T. Ma, M. Tang, J. Cao, Y. Tian, A. Al-Dhelaan, and M. Al-Rodhaan. An efficient and scalable density-based clustering algorithm for datasets with complex structures. *Neurocomputing*, 171:9–22, 2016.

[77] L. v. d. Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[78] S. T. Mai, I. Assent, and M. Storgaard. Anydbc: An efficient anytime density-based clustering algorithm for very large complex datasets. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 1025–1034, New York, NY, USA, 2016. Association for Computing Machinery.

[79] S. Maurus and C. Plant. Skinny-dip: Clustering in a sea of noise. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 1055–1064, New York, NY, USA, 2016. ACM.

[80] L. McInnes, J. Healy, and S. Astels. hdbscan: Hierarchical density based clustering. *The Journal of Open Source Software*, 2(11):205, 2017.

[81] L. F. Mendes, B. Ding, and J. Han. Stream sequential pattern mining with precise error bounds. In *2008 Eighth IEEE International Conference on Data Mining*, pages 941–946. IEEE, 2008.

[82] F. Nakagaito, T. Ozaki, and T. Ohkawa. Discovery of quantitative sequential patterns from event sequences. In *Proceedings of the 2009 IEEE International Conference on Data Mining Workshops*, pages 31–36. IEEE Computer Society, 2009.

[83] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Van-

derplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[84] I. Rahal, D. Ren, W. Wu, and W. Perrizo. Mining confident minimal rules with fixed-consequents. In *16th IEEE International Conference on Tools with Artificial Intelligence, 2004. ICTAI 2004.*, pages 6–13. IEEE, 2004.

[85] A. J. Reich. Intervals, points, and branching time. In *TIME*, pages 121–133, 1994.

[86] A. Rodriguez and A. Laio. Clustering by fast search and find of density peaks. *Science*, 344(6191):1492–1496, 2014.

[87] P. J. Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.

[88] M. Seno and G. Karypis. Lpminer: An algorithm for finding frequent itemsets using length-decreasing support constraint. In *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pages 505–512. IEEE, 2001.

[89] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[90] K. Smets and J. Vreeken. Slim: Directly mining descriptive patterns. In *Proceedings of SIAM International Conference on Data Mining*, pages 236–247. SIAM, 2012.

[91] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *International Conference on Extending Database Technology*, pages 1–17. Springer, 1996.

[92] L. Szathmary, A. Napoli, and P. Valtchev. Towards rare itemset mining. In *19th IEEE International Conference on Tools with Artificial Intelligence, 2007. ICTAI 2007.*, volume 1, pages 305–312. IEEE, 2007.

[93] F. Tao, F. Murtagh, and M. Farid. Weighted association rule mining using weighted support and significance framework. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–666. ACM, 2003.

[94] T. N. Tran, R. Wehrens, and L. M. Buydens. Knn-kernel density-based clustering for high-dimensional multivariate data. *Computational Statistics & Data Analysis*, 51(2):513–525, 2006.

[95] L. Troiano and G. Scibelli. A time-efficient breadth-first level-wise lattice-traversal algorithm to discover rare itemsets. *Data Mining and Knowledge Discovery*, 28(3):773–807, 2014.

[96] S. Tsang, Y. S. Koh, and G. Dobbie. Rp-tree: Rare pattern tree mining. In *Proceedings of the 13th International Conference on Data Warehousing and Knowledge Discovery*, DaWaK'11, pages 277–288, Berlin, Heidelberg, 2011. Springer-Verlag.

[97] T. Uno, M. Kiyomi, and H. Arimura. LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In R. J. B. 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.

[98] S. Vadapalli, S. R. Valluri, and K. Karlapalem. A simple yet effective data clustering algorithm. In *Sixth International Conference on Data Mining (ICDM'06)*, pages 1108–1112. IEEE, 2006.

[99] N. X. Vinh, J. Epps, and J. Bailey. Information theoretic measures for clusterings comparison: is a correction for chance necessary? In *Proceedings of the 26th annual international conference on machine learning*, pages 1073–1080, 2009.

[100] K. Wang, Y. He, and D. W. Cheung. Mining confident rules without support requirement. In *Proceedings of the tenth international conference on Information and knowledge management*, pages 89–96. ACM, 2001.

[101] K. Wang, Y. He, and J. Han. Pushing support constraints into association rules mining. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):642–658, 2003.

[102] S.-Y. Wu and Y.-L. Chen. Mining nonambiguous temporal patterns for interval-based events. *Knowledge and Data Engineering, IEEE Transactions on*, 19(6):742–758, 2007.

[103] H. Xiong, P.-N. Tan, and V. Kumar. Mining strong affinity association patterns in data sets with skewed support distribution. In *Third IEEE International Conference on Data Mining, 2003. ICDM 2003.*, pages 387–394. IEEE, 2003.

[104] H. Yan., Y. Lu., and H. Ma. Density-based clustering using automatic density peak detection. In *Proceedings of the 7th International Conference on Pattern Recognition Applications and Methods - ICPRAM,*, pages 95–102. INSTICC, SciTePress, 2018.

[105] M. Yoshida, T. Iizuka, H. Shiohara, and M. Ishiguro. Mining sequential patterns including time intervals. In *Data Mining and Knowledge Discovery: Theory, Tools, and Technology II*, volume 4057, pages 213–220. International Society for Optics and Photonics, 2000.

[106] H. Yun, D. Ha, B. Hwang, and K. H. Ryu. Mining association rules on significant rare data using relative support. *Journal of Systems and Software*, 67(3):181–191, 2003.

[107] M. J. Zaki. Scalable algorithms for association mining. *IEEE transactions on knowledge and data engineering*, 12(3):372–390, 2000.

[108] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 326–335. ACM, 2003.

[109] M. J. Zaki and C.-J. Hsiao. Charm: An efficient algorithm for closed itemset mining. In *Proceedings of the 2002 SIAM international conference on data mining*, pages 457–473. SIAM, 2002.

[110] L. Zelnik-Manor and P. Perona. Self-tuning spectral clustering. In *Advances in neural information processing systems*, pages 1601–1608, 2005.

[111] T.-T. Zhang and B. Yuan. Density-based multiscale analysis for clustering in strong noise settings with varying densities. *IEEE Access*, 6:25861–25873, 2018.