Mining Frequent Itemsets from Transactional Data Streams with Probabilistic Error Bounds

Dissertation
zur
Erlangung des Doktorgrades (Dr. rer. nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt
von
Daniel Trabold
aus
Frankfurt am Main

Bonn, 2020

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

1. Gutachter: Prof. Dr. Stefan Wrobel
2. Gutachter: Prof. Dr. Christian Bauckhage
Tag der Promotion: 18.03.2020
Erscheinungsjahr: 2020

Daniel Trabold

Fraunhofer Institute for Intelligent Analysis
and Information Systems IAIS

Schloss Birlinghoven
53754 Sankt Augustin
Germany

Declaration

I, Daniel Trabold, confirm that this work is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g. ideas, equations, figures, text, tables, programs) are properly acknowledged at the point of their use. A full list of the references employed has been included.

Acknowledgements

This thesis was written at the department Knowledge Discovery of Fraunhofer IAIS in Sankt Augustin under my thesis supervisors Prof. Dr. Stefan Wrobel and Dr. Tamás Horváth. Without the continuous support of many, this thesis would not have been written. I am most thankful to Tamás who advised and guided me with great patience over the years. He is not only a wonderful advisor but also a lovely host as I had the pleasure to experience during several intense weeks of work in Nemesvita.
A warm thank you goes to all my co-authors and especially those who contributed to the articles that relate to this thesis. These are Tamás Horváth, Stefan Wrobel, Dr. Mario Boley, and PD Dr. Michael Mock. Everyone has contributed advice and own experience to the success of the articles.
I express my gratitude to Dr. Dirk Hecker and Dr. Stefan Rüping for their support and liberation from other projects for the hours that went into this thesis. An equally warm thank you goes to all the wonderful colleagues at the department Knowledge Discovery. Especially, to Dr. Claus-Peter Buszello who shared his inspiration and thoughts on an industrial problem. I would equally like to thank the colleagues at the MLAI group at the University of Bonn for their hospitality. I am grateful to all those who have read preliminary versions of this thesis and provided valuable feedback with their comments and questions.
I would also like to thank some former or present colleagues who became great friends along the way and supported me throughout this journey. Back from the days in Ulm these are: Nira & Martin Schierle, Elena & Mathias Bank, and Michael Krüger. In Bonn I would like to thank: Nathalja Friesen, Franziska Dörr, Sebastian Bothe, Sebastian Konietzny, and Sven Giesselbach.
Last but not least, I thank my two magnificent sisters and our beloved parents for their support. Thank you!

Abstract

Frequent itemset mining is a classical data mining task with a broad range of applications, including fraud discovery and product recommendation. The enumeration of frequent itemsets has two main benefits for such applications: First, frequent itemsets provide a human-understandable representation of knowledge. This is crucial as human experts are involved in designing systems for these applications. Second, many efficient algorithms are known for mining frequent itemsets. This is essential as many of today's real-world applications produce ever-growing *data streams*. Examples of these are online shopping, electronic payment or phone call transactions. With limited physical main memory, the analysis of data streams can, in general, be only approximate. State-of-the-art algorithms for frequent itemset mining from such streams bound their error by processing the transactions in blocks of fixed size, either each transaction individually or in mini-batches. In theory, single transaction-based updates provide the most up-to-date result after each transaction, but this enumeration is inefficient in practice as the number of frequent itemsets for a single transaction can be exponential in its cardinality. Mini-batch-based algorithms are faster but can only produce a new result at the end of each batch. In this thesis, the binary choice between up-to-date results and speed is eliminated. To provide more flexibility, we develop new algorithms with a probabilistic error bound that can process an arbitrary number of transactions in each batch.

State-of-the-art algorithms mining frequent itemsets from data streams with mini-batches derive the size of the mini-batch from a user-defined error parameter and hence couple their error bound to the size of the update. By introducing a *dynamic error bound* that adapts to the length of the data stream the error is decoupled from the size of the update. The benefits of this approach are twofold: First, the dynamic error bound is independent of the size of the update. Hence, an arbitrary number of transactions can be processed without losing the error bound. Second, the bound becomes tighter as more transactions arrive and thus the tolerated error decreases, in contrast to algorithms with static thresholds. Our approach is extensively compared to the state-of-the-art in an empirical evaluation. The results confirm that the dynamic approach is not only more flexible but also outperforms the state-of-the-art in terms of F-score for a large number of data streams.

As it is easier for experts to extract knowledge from a smaller collection, we consider mining a compact pattern set. Especially useful are parameterized pattern classes for which the expert can regulate the size of the output. An example of such a parameterized pattern class are *strongly closed itemsets*. Additionally, they are stable against small changes in the data stream. We present an algorithm mining strongly closed itemsets from data streams. It builds on reservoir sampling and is thus capable of producing a result after any number of transactions, once the initial sample is complete. The high approximation quality of the algorithm is empirically demonstrated and the potential of strongly closed patterns for two stream mining tasks is shown: concept drift detection and product configuration recommendation.

Zusammenfassung

Das *Finden häufiger Itemsets* ist eine klassische Aufgabe der Datenanalyse mit breitem Anwendungsspektrum, wie etwa der Betrugserkennung und der Produktempfehlung. Häufige Itemsets stellen eine verständliche Form der Wissensrepräsentation dar. Davon profitieren Experten, die mit ihnen arbeiten. Außerdem gibt es viele effiziente Algorithmen für ihre Aufzählung. Dies ist wesentlich, da viele reale Anwendungen lange *Datenströme* erzeugen. Beispiele hierfür sind Online-Einkäufe, Transaktionen aus elektronischen Zahlungen und der Telekommunikation. Durch den limitierten Hauptspeicher kann die Analyse von Datenströmen in der Regel nur approximativ erfolgen. Die Qualität der Approximation richtet sich nach einer Fehlerschranke. Existierende Algorithmen beschränken ihre Fehler durch die Verarbeitung von Transaktionen in Blöcken fester Größe. Entweder verarbeiten sie Transaktionen einzeln oder in Mini-Batches. Theoretisch liefern transaktionsbasierte Verfahren das aktuellste Ergebnis nach jeder Transaktion. In der Praxis ist diese Strategie jedoch langsam, da die Anzahl häufiger Muster in einer Transaktion exponentiell in ihrer Kardinalität sein kann. Mini-Batch-basierte Algorithmen sind schneller, können jedoch nur am Ende eines Batches ein Ergebnis liefern. Als Alternative zur binären Wahl zwischen Aktualität und Geschwindigkeit werden hier neue Algorithmen vorgestellt, die in der Lage sind Batches mit einer beliebigen Anzahl an Transaktionen zu verarbeiten. Für diese beweisen wir eine probabilistische Schranke ihres Fehlers.

Aktuelle Mini-Batch-basierte Algorithmen leiten ihre Batch-Größe aus einem vom Anwender vorgegebenen Fehlerparameter ab. Hierdurch koppeln die Algorithmen die Fehlerschranke an die Batch-Größe. In dieser Arbeit wird durch die Einführung einer *dynamischen Fehlerschranke*, die sich an die Länge des Datenstroms anpasst, eine größere Flexibilität erreicht, die zweierlei Vorteile bietet. Erstens beweisen wir, dass die dynamische Fehlerschranke unabhängig von der Größe des Mini-Batches ist und daher beliebig viele Transaktionen verarbeitet werden können. Zweitens wird die Schranke kleiner und der tolerierte Fehler geringer, je mehr Transaktionen vorliegen. Im Gegensatz hierzu verwenden viele existierenden Algorithmen statische Fehlerschranken. Der neue Ansatz wird umfassend empirisch mit dem Stand der Technik verglichen. Die Ergebnisse bestätigen die Vorteile unseres dynamischen Ansatzes. Das Verfahren ist nicht nur flexibler als der Stand der Technik, sondern es erreicht auch ein besseres F-Maß für die Mehrzahl getesteter Datensätze.

Ergänzend wird das Finden einer kompakten Mustermenge betrachtet, da die Wissensextraktion aus kleineren Mengen für Experten einfacher ist. Besonders nützlich sind parametrisierbare Mengen, für die sich die Zahl der Muster steuern lässt. Eine dieser Mengen sind *Strongly Closed Itemsets*. Wir stellen einen Algorithmus für das Finden von Strongly Closed Itemsets aus Datenströmen vor. Er basiert auf dem Reservoir-Sampling-Verfahren und ist daher in der Lage, nach einer beliebigen Anzahl von Transaktionen, ein Ergebnis zu liefern. Die hohe Approximationsgüte des Ansatzes wird empirisch belegt und das Potenzial von Strongly Closed Itemsets bei der Datenstromanalyse beim Erkennen von Konzeptverschiebungen und dem Empfehlen von Produktkonfigurationen aufgezeigt.

Contents

1. Introduction 1
1.1. Motivation . . . 1
1.2. Background . . . 4
1.3. Contributions . . . 6
1.4. Previously Published Work . . . 9
1.5. Outline . . . 10

2. Notions and Problem Definitions 11
2.1. Itemset Mining . . . 11
2.1.1. Closed and Crucial Itemsets . . . 15
2.1.2. Strongly Closed and Maximal Frequent Itemsets . . . 16
2.2. Data Streams . . . 20
2.2.1. Properties of Data Streams . . . 20
2.2.2. Data Stream Models . . . 22
2.2.3. Transactional Data Streams . . . 23
2.3. Problem Definitions . . . 24

3. Related Work 29
3.1. Frequent Itemset Mining from Data Streams . . . 29
3.2. Landmark Algorithms . . . 30
3.3. Sliding Window Algorithms . . . 37
3.4. Time Fading Window Algorithms . . . 40
3.5. Discussion . . . 41
3.6. Summary . . . 42

4. Frequent Itemset Mining from Transactional Data Streams 45
4.1. Contribution . . . 45
4.2. The Partial Counting Algorithm . . . 46
4.2.1. Support Approximation Strategies . . . 48
4.2.2. Implementation Issues . . . 51
4.3. The Dynamic Threshold Miner . . . 53
4.4. Empirical Evaluation . . . 57
4.4.1. Data sets . . . 57
4.4.2. Design of Experiment . . . 59
4.4.3. Experimental Comparison . . . 60
4.5. Discussion . . . 70
4.6. Summary . . . 71

5. Strongly Closed Itemset Mining from Transactional Data Streams 73
5.1. Motivation . . . 73
5.2. The Strongly Closed Stream Mining Algorithm . . . 74
5.2.1. Sampling . . . 75
5.2.2. Incremental Update . . . 76
5.2.3. Implementation Details . . . 83
5.3. Empirical Evaluation . . . 84
5.3.1. Relative Closure Strength . . . 86
5.3.2. Error . . . 87
5.3.3. Confidence . . . 88
5.3.4. Buffer Size . . . 89
5.3.5. Mining Quality . . . 90
5.3.6. Speed-up . . . 91
5.4. Potential Applications . . . 92
5.4.1. Concept Drift Detection . . . 94
5.4.2. Product Configuration Recommendation . . . 104
5.5. Discussion . . . 111
5.6. Summary . . . 114

6. Conclusion 115

Bibliography 119

Appendix 129
A. Parameter Tuning . . . 129

List of Acronyms

AP_{Stream}	–	Approximate Partition for Stream
CPM	–	Crucial Pattern Mining
CPS-Tree	–	Compact Pattern Stream Tree
DCIM	–	Dynamic Confidence Interval Miner
DFP	–	Dynamical Frequent Pattern
DIU	–	Direct Update
DSCA	–	Data Stream Combinatorial Approximation
DSM-FI	–	Data Stream Mining for Frequent Itemsets
FCI	–	Frequent Closed Itemset
FDPM	–	Frequent Data Stream Pattern Mining
FP-Growth	–	Frequent Pattern Growth
FP-Tree	–	Frequent Pattern Tree
IFI	–	Inverted FCI Index
LC	–	Lossy Counting
MFCI	–	Maintenance of Frequent Closed Itemsets
MFI-TransSW	–	Mining Frequent Itemsets with a Transaction-sensitive Sliding Window
S_Apriori	–	Stream Apriori
SA-Miner	–	Support-Approximation-based Data Stream Frequent-Pattern Miner
SCSM	–	Strongly Closed Stream Miner
SWIM	–	Sliding Window Incremental Miner
TID	–	Transaction Identifier
WSW	–	Weighted Sliding Window

List of Figures

1.1. Example payment transactions . . . 2
1.2. Frequent Itemsets . . . 3

2.1. Illustration of itemset encodings . . . 13
2.2. Itemset lattice . . . 13
2.3. FP-Tree data structure . . . 15
2.4. Pattern classes . . . 17
2.5. Relationships between frequent pattern classes . . . 19
2.6. Relationship between crucial and strongly closed itemsets . . . 19
2.7. Window models . . . 22

3.1. Example transactional data stream . . . 30
3.2. DSM-FIs data structure . . . 34
3.3. Decision tree for algorithms mining frequent itemsets from data streams . 42

4.1. Partial Counting working principle . . . 47
4.2. DTM itemset states . . . 54
4.3. DTMs development of the itemset states over time . . . 54
4.4. Effect of varying the frequency threshold for the UCI data streams . . . 62
4.5. Effect of varying the frequency threshold for the QUEST data streams . . 62
4.6. Effect of increasing data stream length for the UCI data streams . . . 63
4.7. Worst case effect of increasing data stream length for the QUEST data streams . . . 64
4.8. Effect of varying the number of items . . . 65
4.9. Effect of varying the average number of items per transaction . . . 66
4.10. Effect of varying the average length of maximal patterns . . . 67
4.11. Effect of varying the number of patterns . . . 68
4.12. Effect of varying the correlation between patterns . . . 69

5.1. Strongly closed itemsets example . . . 75
5.2. SCSM call stack for the update of Figure 5.1 . . . 79
5.3. SCSM worst case effect of varying $\tilde{\Delta}$ . . . 86
5.4. SCSM worst case effect of varying error . . . 87
5.5. SCSM worst case effect of varying confidence . . . 88
5.6. SCSM worst case effect of varying buffer size . . . 89
5.7. SCSM runtime as fraction of Batch . . . 92
5.8. Runtime results for SCSM on T10I4D100k . . . 92
5.9. Illustration of drift paces . . . 95

5.10. Effect of the drift type on concept drift detection 100
5.11. Effect of the drift length on concept drift detection 100
5.12. Effect of the probability of intersection on concept drift detection 101
5.13. Effect of $\tilde{\Delta}$ on concept drift detection 101
5.14. Effect of the delay between miners on concept drift detection 102
5.15. Effect of the buffer size on concept drift detection 103
5.16. Effect of the detection delay on concept drift detection 103
5.17. Cumulative item frequency distribution: real-world vs. benchmark 107
5.18. Transaction histograms: real-world vs. benchmark 108
5.19. Co-ocurrences of items: real-world vs. benchmark 109
5.20. Product configuration results 111
5.21. Product configuration strongly closed sets 112

A.1. DTM worst-case effect of varying δ on UCI data 130
A.2. DTM average effect of varying δ on UCI data 130
A.3. DTM worst-case effect of varying δ on QUEST data 131
A.4. DTM average effect of varying δ on QUEST data 131
A.5. EStream average effect of error on UCI data 132
A.6. EStream average effect of error on QUEST data 132
A.7. FDPM average effect of reliability on UCI data 133
A.8. FDPM average effect of reliability on QUEST data 133
A.9. FDPM average effect of k on UCI data 134
A.10.FDPM average effect of k on QUEST data 134
A.11.Lossy Counting effect of error on UCI data 135
A.12.Lossy Counting effect of error on QUEST data 135
A.13.Partial Counting effect of strategy on UCI data 136
A.14.Partial Counting effect of strategy on QUEST data 136
A.15.SApriori effect of confidence on UCI data 138
A.16.SApriori effect of confidence on QUEST data 138
A.17.SApriori effect of error on UCI data 139
A.18.SApriori effect of error on QUEST data 139

List of Tables

3.1. Characteristics of algorithms mining frequent itemsets 43

4.1. Illustration of PARTIAL COUNTINGS estimation strategies 51
4.2. Benchmark data sets used for the empirical evaluation 58
4.3. Parameters used for synthetic data stream generation 59
4.4. Optimal parameters for frequent itemset mining algorithms 61
4.5. Average F-score over all frequent itemset mining experiments 69

5.1. Benchmark data sets used for the empirical evaluation of SCSM 85
5.2. Sample sizes for various error and confidence values of SCSM 85
5.3. Number of $\tilde{\Delta}$-closed sets, precision and recall for benchmark data sets . . 91
5.4. Number of $\tilde{\Delta}$-closed sets, precision and recall for QUEST data sets 1/2 . 91
5.5. Number of $\tilde{\Delta}$-closed sets, precision and recall for QUEST data sets 2/2 . 92
5.6. SCSMs speedup for changing a single transaction 93
5.7. Real-world product configuration data set characteristics 110

A.1. Optimal parameters for frequent itemset mining algorithms 140

1. Introduction

Frequent itemset mining (Agrawal et al., 1993) is considered to be one of the most important data mining problems (Zhu et al., 2007). It is motivated by a huge number of practical applications including, for example, association rule discovery, fraud discovery, recommendation systems, advertising strategies, catalog design, plagiarism detection, and biomarker identification (Leskovec et al., 2014). While the algorithmic aspects of pattern discovery from *static* databases are well-understood, many real-world settings produce continuous and potentially unbounded *data streams* (Gama, 2010). Their analysis is a computationally challenging task because they provide a constant flow of new information, while the computational resources are limited (Babcock et al., 2002). State-of-the-art algorithms mining frequent itemsets from transactional data streams guarantee *error bounds* only if the transactions are either processed *transaction by transaction* or in fixed-size *mini-batches*. The first strategy is slow. In particular, frequency-based pruning is less efficient for single transaction updates. This limits such algorithms to transactions of small size[1]. Mini-batch-based updates lack flexibility: They only produce new results at the end of each mini-batch. Working with a fixed batch size is impractical if the transaction rate changes over time as is the case, e.g., for electronic payment transactions. If such algorithms are queried for an updated result while the mini-batch is incomplete, they lose their guarantees. In this thesis, we address the problem of the binary choice between slow or fast but static *fixed-size* update intervals. Our goal is to *decouple* the *error bound* from the *update interval*. We present algorithms with guaranteed error bounds that combine the flexibility of the transaction-based scheme and the speed of mini-batch-based updates. In other words, the goal is to present new algorithms with guarantees that are independent of the number of processed transactions, without the need to produce a new result for each individual transaction.

1.1. Motivation

Data streams are ubiquitous. Any series of events constitutes a data stream. The speed of the wind or the temperature at a given location are two data streams of natural origin, while traffic flow, payment transactions, telecommunication, or network packages are human-generated data streams. These are just a few of the omnipresent examples of sources of data streams. Such streams can be analyzed with various goals. As a prominent example, in case of payment and telecommunication transactions, one of the goals of the analysis is the identification of fraud, see, e.g., Delamaire et al. (2009); Rosset

[1] For any minimum frequency threshold θ, transaction by transaction enumeration would generate *all* subsets (i.e., exponentially many frequent itemsets) for each of the first $1/\theta$ transactions.

Country	MCC	Amount in €	Fraud
DEU	5411	54.23	false
DEU	5661	198.99	true
DEU	5411	27.31	false
FRA	8021	102.95	false
FRA	5940	1799.00	false
ITA	5411	54.23	false

Figure 1.1.: Example payment transactions, showing the country, merchant category code (mcc), amount in € and fraud flag. See, e.g., Bhattacharyya et al. (2011) for further typical data fields.

et al. (1999). Fraud discovery is important due to the high financial losses caused by fraudulent behavior. The European Central Bank, for example, reported a total value of €1.8 billion of fraud losses in the year 2016 for the cards issued within the Single Euro Payments Area (SEPA) (ECB, 2018).
To reduce these losses, fraud analysts often craft manual rules and feed them into commercial systems such as Online watcher, BV Detect, or SAS Fraud Manager. The interpretability of the fraud patterns is crucial in this context for the fraud analysts. First, they try to understand the fraud patterns to reason about new means to prevent fraud and second, they want to be able to explain a customer who's transaction got accidentally blocked why it was blocked. For these analysts, frequent itemsets and rules derived from are a valuable input. Once a rule is implemented to prevent a certain fraud pattern, fraudsters adapt their strategy and new fraud patterns emerge. This creates a constant demand for the continuous analysis of the stream of payment transactions and the identification of new frequent fraud patterns. If an analyst detects an active attack, i.e., a sudden peak in the number of reported fraud incidents, she is interested in an immediate updated result of the frequent patterns to implement new rules to stop this new kind of attack. Waiting for any mini-batch to complete in this setting is a considerable obstacle.
As an example, we consider the following set of payment transactions given in Figure 1.1[2]. The attributes of the transactions are the country in which the transaction took place, the code for the category of the merchant (mcc), the amount of money spent in euro, and a fraud flag. We call each "value" of a transaction an item and any combination of items an itemset. In the above example, we have a total of 14 items. For a universe of n items, there are $2^n - 1$ different non-empty itemsets. It is infeasible to list them all if n is not very small. A common strategy is to list only the *frequent* itemsets, cf. Agrawal et al. (1993). An itemset is frequent if it occurs at least as often as a user-defined minimum support threshold. We ignore the fraud flag for now and list all the frequent itemsets for a minimum support threshold of 2 in Figure 1.2.

[2] For the ease of understanding, we do not list the full set of data fields available for each transaction. Payment transactions include data about the card, the merchant, the payment, and its processing.

Itemset	**Support**
DEU	3
5411	3
DEU & 5411	2
FRA	2
54.23	2
5411 & 54.23	2

Figure 1.2.: Frequent itemsets for the example transactions of Figure 1.1 with a minimum support threshold of 2.

While an *absolute* minimum support threshold is reasonable for static databases, it is less suitable for growing data streams: As the stream length increases, with high probability more and more itemsets will become frequent for a given minimum support threshold. It is therefore common to use a *relative* frequency threshold θ instead. An itemset is frequent given the relative threshold θ if it occurs in at least a fraction θ of transactions in the stream. For any fixed-size data set, the minimum support threshold can be converted to a minimum frequency threshold and vice versa. For the above example, the minimum support threshold of 2 corresponds to the frequency threshold $\theta = 1/3$.

For the task of fraud discovery from transactional data streams, frequent itemsets can be employed as follows: Generate one data stream from all fraudulent transactions and add a transaction to it as soon as it is reported as fraudulent. The patterns discovered from this stream are potentially characteristic for fraud. They could, however, be characteristic both for fraud and genuine transactions. It is hence necessary and common to compute a reference model. This model would ideally be mined only from all genuine transactions. However, the true labels of transactions are not known for up to three months. During this period a customer can report any transaction as fraudulent. To stop fraud early in this setting an online update of the fraud cases is required as soon as they are reported. If a transaction is reported as fraud, it is labeled as a fraud transaction and immediately added to the stream of fraudulent transactions. All other transactions are considered to be genuine. Overall, fraud cases are very rare events. It is, therefore, reasonable to assume that the reference model is not affected by the very few transactions that turn out to be fraud cases later. The reference model can hence be computed from the stream of all transactions without delay, providing an up-to-date result needed for fast reaction. Given the set of frequent itemsets mined from both streams, these sets are compared to each other. Patterns that occur with high frequency in the fraudulent transactions and are infrequent in all transactions are characteristic for fraud.

We have applied our approach to real-world payment transactions and mined frequent itemsets with a minimum frequency threshold of 0.1, i.e, 10% of the transactions. From our results, we report two patterns: 68% of the fraudulent transactions have security type = 5 and response code = 2 whereas this combination occurs for only 17% of all

transactions of one bank. For another bank we obtained a pattern identifying a single merchant. It occurred in 61% of the fraudulent transactions and less than 10% in the stream of all transactions for two months. The ECB does not reveal its methods of analysis but reports a pattern that would be found with our method: "Cross-border transactions within SEPA made up for 8% of all transactions, but 43% of fraudulent transactions." (ECB, 2018).
We have motivated our work with an *exemplary* use case from finance. Similar use cases exists in other industrial segments, for example, mobile phone fraud discovery in telecommunications (Etzion et al., 2016). We note that our approach sketched above is not limited to fraud detection; in a more general perspective it can be applied to any transactional data stream classification task even with strong class imbalance and delayed label generation.

1.2. Background

Data mining is concerned with the extraction of knowledge (e.g., fraud patterns) from large data collections that are static (e.g., databases) or dynamic (e.g., data streams). It is an interdisciplinary field between computer science and statistics. After more than 20 years, the classical definition of Fayyad et al. (1996) is still up-to-date:

> "Knowledge discovery [...] is the nontrivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data".

Depending on the data analysis task at hand, the extracted knowledge can take diverse forms. The most prominent tasks include, among others, classification (Duda et al., 2000), clustering (Jain et al., 1999), outlier detection (Hodge and Austin, 2004), pattern discovery (Mannila, 2002), regression (Draper and Smith, 1998), and summarization (Gong and Liu, 2001). Each task has its objective and solves accordingly some particular class of problems. *Pattern discovery*, the subfield we are interested in, can itself take many forms, such as frequent itemset mining or association rule discovery (Agrawal et al., 1993), subgroup discovery (Klösgen, 1995), and others.
The book "Knowledge Discovery from Data Streams" by Gama (2010) provides a very good general introduction to the topic of data stream mining. Since the entire data of a potentially unbounded stream cannot be stored in limited memory, different window models for the analysis of data streams have been proposed. The *sliding window* model considers a constant number of the most recent transactions. This model allows accessing the data within the fixed-size window multiple times. The *time fading* model assigns weights to transactions. As transactions become older, their weight is reduced until they eventually no longer contribute to the result and can be removed. This model can be considered as a variation of the sliding window model. These models are, however, not well-suited for some use cases. Consider the example of fraud discovery once more. The number of fraudulent transactions fluctuates. Hence, the frequent itemsets mined from the stream containing the fraudulent and the one containing all transactions are hard to synchronize under these models. In contrast, the *landmark* model considers a growing

frame of the data stream from some landmark start transaction. The patterns extracted from the two data streams are always synchronized in this setting, i.e., in our example, all fraudulent transactions are contained in the stream of all transactions. Algorithms extracting patterns from data streams under this model have access to the data in *one pass only* (Babcock et al., 2002). This implies that all data from the past that are removed from the memory will be lost forever; this is an essential difference to the batch algorithms designed for static data sets and those for the other window models. In these cases, data can be accessed multiple times, which facilitates the development of algorithms for these settings. In fact, many data mining algorithms were first proposed for static data sets. In contrast, the analysis of data streams requires specialized algorithms. In the case of the landmark model, they often produce only *approximate* results with some explicit or implicit error bound. The focus of this thesis is on *frequent itemset mining* from transactional data streams with the *landmark* model. This problem is solvable only approximately for any single pass algorithm. Any exact solution requires two passes, cf. Hidber (1999). Most algorithms mining frequent itemsets from data streams with an explicit error guarantee can update the extracted patterns *not* upon request, but at some fixed time points only. This is a strong limitation, especially for the analysis of data streams with variable transaction rates, such as the payment transactions mentioned above.

Formally, each transaction is a non-empty subset of a finite ground set, called the set of items. Its subsets will be referred to as *itemsets*. The batch version of the frequent itemset mining task, i.e, when the *entire* data is available for the algorithm simultaneously, originates from the analysis of shopping baskets, where the items correspond to the products that can be purchased (Agrawal et al., 1993). The central problem for this classical task is to generate *all* itemsets that are bought together frequently. The notion of "frequent" is a binary property specified by a user-defined *frequency threshold*. In this sense, the frequency of an itemset denotes the popularity of the combination of the products in it. From the application point of view (e.g., store layout design), such frequent itemsets represent the essential "knowledge" about the shopping behavior of customers. The algorithmic difficulty of this problem lies in the fact that the number of frequent itemsets can be exponential in the cardinality of the set of items, implying the infeasibility of the generation of *all* frequent itemsets.

To overcome the above algorithmic challenge, one may consider the k *most-frequent* itemsets only for some user-defined integer k. Another common option to control the size of the output is to *increase* the frequency threshold. Clearly, the larger the threshold is, the smaller the cardinality of the output. These common pruning approaches introduce, however, a bias towards short patterns (see, e.g., Zhu et al. (2007)). A third possibility is to consider some *compact representation* of the family of frequent itemsets, such as *maximal* (Mannila and Toivonen, 1997), *crucial* (Das and Zaniolo, 2016), or *closed* (Pasquier et al., 1999b) frequent itemsets. While maximal frequent itemsets provide the most compact representation, they cannot be used to determine the support of the frequent itemsets without any (further) database access. Furthermore, it is computationally intractable to generate this family of itemsets (Boros et al., 2003). The family of crucial itemsets, a superset of maximal frequent itemsets, provides another

compact representation. While it is an open question whether they can be enumerated efficiently, this family has the advantage that the support of all frequent itemsets can be derived without any database access (Das and Zaniolo, 2016). Finally, the family of closed frequent itemsets, a superset of crucial itemsets, can be listed with polynomial delay (see, e.g., Gély (2005)) and allows to determine the support count of the frequent itemsets without any database access (Pasquier et al., 1999b). The disadvantage of this family is, however, that its cardinality can be exponential in that of the maximal frequent itemsets (Boros et al., 2003), making it infeasible for data stream applications with too many closed frequent patterns. Similarly to frequent itemsets, the size of the output can be controlled by a frequency threshold, facing, however, the bias mentioned above.

1.3. Contributions

In this thesis, we deal with generating frequent and other types of parameterized (e.g., strongly closed) itemsets from transactional data streams, i.e., from a sequence of transactions received one by one. We consider the problem of mining such itemsets from data streams under the landmark model with guaranteed error bounds. Such guarantees are important, for example, for the discovery of fraud. This section summarizes the most important contributions of this dissertation.

For the *batch* setting, algorithms mining frequent itemsets have been empirically compared by Hipp et al. (2000). Motivated by the vast amount of literature on the subject of frequent itemset mining from transactional data *streams* and the lack of a systematic experimental empirical comparison of the state-of-the-art algorithms for this problem, our first contribution is concerned with an overview of mining frequent itemsets from data streams, including a systematic empirical evaluation of the most prominent state-of-the-art algorithms.

C1 More precisely, in Chapter 3 we collect and discuss the most important state-of-the-art algorithms mining frequent itemsets from transactional data streams, including their main algorithmic properties. Since this chapter considers not only the algorithms developed for the landmark model, but also those working in the sliding window and the time fading models, the chapter serves as a short *survey* of the most prominent algorithms and as such, may be of some independent interest. For the empirical evaluation, we considered the algorithms Carma (Hidber, 1999), Lossy Counting (Manku and Motwani, 2002), FDPM (Yu et al., 2006), SApriori (Sun et al., 2006), and EStream (Dang et al., 2008). All other algorithms discussed in Chapter 3 have been excluded from our experiments, as they are tailored to specific data streams, e.g., streams with short transactions.

To obtain a fair comparison, we implemented all five algorithms listed above and evaluated them extensively in Chapter 4, where we compare the state-of-the-art to our new algorithms. For the FDPM algorithm, in particular, we have identified a bottleneck in its original pruning mechanism (Yu et al., 2004). This motivated us

to modify the pruning strategy of the algorithm, obtaining a much faster algorithm in this way, without affecting its accuracy. We omit the technical details of this reimplementation and denote the modified algorithm in the subsequent chapters by FDPM*. Finally, we systematically evaluated these state-of-the-art algorithms on real-world and synthetic data streams of different characteristics. All algorithms, except for FDPM*, obtained an average F-score below 0.9. Only FDPM*, our improved version of FDPM, was able to achieve an average F-score of 0.96. This is especially surprising, as the algorithms SApriori and EStream were published after FDPM.

Our analysis of the state-of-the-art algorithms revealed that they lack the flexibility to be queried after an arbitrary number of transactions: Either they are based on the mini-batch design or they compute an update for each transaction at the expense of slow throughput. The algorithms with mini-batches derive the batch size from their error bound. Accordingly, they lose their error guarantee if an incomplete batch is processed. The use case of fraud discovery requires a fast response whenever the fraud rate increases. As mentioned above, fixed-size batches are obstacles in this setting. We, therefore, consider the problem of mining transactional data streams with error bounds that hold independently of the number of processed transactions. More precisely, we decouple the probabilistic guarantee from the update interval to achieve the high throughput of mini-batch-based algorithms and the flexibility of updates per transaction at the same time to develop algorithms that can be queried after any number of transactions.

C2 Our second main contribution is as follows: We first present an algorithm improving the estimation of an itemset's frequency for the past, when it was not counted, as additional transactions are received in the stream. This is a distinguishing feature compared to the state-of-the-art algorithms. They profit from longer data streams by obtaining more accurate statistics for each itemset from the time only when they start counting it. Our algorithm additionally uses its statistics to derive a dynamic estimate for the past based on conditional probabilities. More precisely, our data stream mining algorithm, called Partial Counting, maintains the count of an itemset together with the counts of its immediate subsets to compute conditional probabilities given the counts of the subsets. It obtains an average F-score of 0.9 in our extensive empirical evaluation, which compares to the state-of-the-art algorithms mentioned above in contribution C1. However, the F-score obtained by Partial Counting is below that of FDPM*. The time and space required by this algorithm are moderate in comparison to the best algorithms. Remarkably, the probabilistic inference sketched above outperforms the well-established Lossy Counting algorithm by Manku and Motwani (2002) in terms of F-score (Lossy Counting obtained an average F-score of 0.86 only).

This is one of our main motivations for our second algorithm, called Dynamic Threshold Miner (DTM). It computes a dynamic confidence threshold derived from Chernoff's bound based on the user-defined frequency threshold θ and confidence parameter δ. We formally prove that the relative frequency of any itemset is

approximated by our DTM algorithm with at most some small user-defined error ϵ, with probability at least $1-\delta$. With additional transactions in the data stream, the threshold gets tighter. When the algorithm starts counting a new potentially frequent itemset, it uses the same dynamic bounding scheme derived from Chernoff's bound to estimate the itemsets frequency in the past, when it has not been counted. Because the bound depends on the number of transactions, the algorithm can be queried after any number of transactions to obtain an updated result. We extensively compare our algorithm empirically to the state-of-the-art algorithms. It achieves an excellent average F-score of 0.98, outperforming all state-of-the-art algorithms. This result shows that our probabilistic reasoning works well in practice. We describe the two algorithms above in detail in Chapter 4.

The set of frequent itemsets can become infeasible large, essentially restraining e.g. understandability. To facilitate knowledge extraction, we therefore aim at a further reduction of the set of patterns. This is additionally motivated by the fact that frequent itemsets have a *language bias* towards short patterns. It is caused by the fact, that any subset of a frequent itemset is not only frequent, but has typically a higher frequency than the itemset containing it. Thus, the increase of the frequency threshold eliminates long patterns, i.e., we have a language bias towards short patterns. Long patterns can be especially useful in knowledge acquisition, as they represent a combination of conditions (i.e., items). Closed itemsets reduce the pattern space compared to frequent itemsets, but the set of closed patterns can still be huge. Frequent closed itemsets, on the other hand, are controlled by the frequency threshold and as discussed above suffer from the language bias towards short patterns. To overcome this language bias, we propose to mine the family of *relatively strongly* or $\tilde{\Delta}$-*closed* itemsets (Boley et al., 2009b) from data streams. For a growing data stream, the number of absolutely Δ-closed patterns (Boley et al., 2009b) increases with the stream length, in contrast to relatively $\tilde{\Delta}$-closed patterns. The size of the output of this parameterized pattern class can be effectively controlled with the parameter $\tilde{\Delta}$ without the language bias towards short patterns. Hence, the output can contain long patterns. This third main contribution is presented in Chapter 5 and can be summarized as follows:

C3 We consider the problem of mining the family of relatively $\tilde{\Delta}$-closed itemsets. The output of this pattern class can be controlled effectively. We present the SCSM algorithm mining *relatively strongly closed itemsets* from data streams. The algorithm maintains a fixed-size sample, where the sample size is determined by Hoeffding's inequality. It is a probabilistic algorithm with a classical ϵ-δ-bound. More precisely, the sample size is chosen in a way that with probability at least $1-\delta$, the relative frequency in the sample deviates from that in the data stream by at most ϵ. The fixed sample size allows us to cast the problem into that of mining *absolutely* Δ-closed itemsets. They have several algorithmic advantages. In particular, they can be enumerated efficiently with a closure operator and are stable against changes (Boley et al., 2010). Our algorithm efficiently maintains and updates the family of Δ-closed itemsets from the sample. More precisely, it decomposes the update operation computing the closure based on a case distinction

for the outcome of the intersection of a closed set with the newly added and removed transactions. With this strategy, the algorithm can avoid the computation of the closure operator in many cases and hence save a significant amount of time. We use the reservoir sampling scheme (Knuth, 1997; Vitter, 1985), which allows it to update the patterns at any time after the initial sample has been completed. Hence, our algorithm follows the *anytime* paradigm (Dean and Boddy, 1988) with respect to the sample. Our empirical evaluation confirms the high approximation quality of the SCSM algorithm in terms of precision and recall. Furthermore, it demonstrates that it is significantly faster than the BATCH algorithm generating the set of strongly closed patterns for each mining request anew.

In addition, we evaluate the suitability of strongly closed itemsets for the classical task of *concept drift detection* and computer-aided *product configuration recommendation* (Falkner et al., 2011). Our extensive experimental results confirm that already a few strongly closed patterns are capable of detecting concept drifts for a broad variety of such drifts and various algorithmic parameters. The latter is important to detect drifts, even if the parameters are not well-tuned. For the problem of product configuration recommendation, we evaluate the suitability of strongly closed sets empirically. Our empirical results with real-world data from an industrial project demonstrate the suitability of strongly closed patterns for this task. The recommendation based on strongly closed sets clearly outperforms a purely frequency-based recommendation approach by up to 37% fewer user queries, indicating the suitability of strongly closed sets for this task. Besides this positive result on the application of strongly closed sets, we discovered that the characteristics of real-world data from industry differ largely from those of synthetic transaction benchmark data sets.

1.4. Previously Published Work

The central ideas and algorithms of this dissertation have been previously published in conference proceedings and one journal article listed below. The journal article contains the potential application scenarios of the strongly closed patterns covered in this thesis.

1. Daniel Trabold, Tamás Horváth and Stefan Wrobel. Effective approximation of parametrized closure systems over transactional data streams. In *Machine Learning*, 2019. (Trabold et al., 2019)

2. Daniel Trabold and Tamás Horváth. Mining Strongly Closed Itemsets from Data Streams. In *Proceedings of the 20th International Conference on Discovery Science*, 2017. (Trabold and Horváth, 2017)

3. Daniel Trabold and Tamás Horváth. Mining Data Streams with Dynamic Confidence Intervals. In *Proceedings of the 18th International Conference on Big Data Analytics and Knowledge Discovery*, 2016. (Trabold and Horváth, 2016)

4. Daniel Trabold, Mario Boley, Michael Mock, Tamás Horváth. In-Stream Frequent Itemset Mining with Output Proportional Memory Footprint. In *Proceedings of the LWA 2015 Workshops: KDML, FGWM, IR, and FGDB*, 2015. (Trabold et al., 2015)

1.5. Outline

The remainder of this dissertation is structured as follows:

Chapter 2 introduces the necessary basic concepts of itemset and data stream mining. It defines the formal notions and notation used throughout this thesis and presents the formal problems considered in subsequent chapters.

Chapter 3 covers the existing related work on frequent itemset mining from transactional data streams. For completeness, we cover all three window models. This short survey discusses each algorithm and facilitates the selection of an algorithm with some desired properties for a given problem.

Chapter 4 considers the problem of mining frequent itemsets from data streams in the landmark model with probabilistic error bounds. For this setting, two algorithms are presented. Both algorithms can produce results after an arbitrary number of transactions. One converges in the limit. For the other, we prove its theoretical probabilistic error bound. An extensive empirical evaluation compares them to the state-of-the-art and demonstrates their strengths.

Chapter 5 presents the first algorithm for mining relatively strongly closed itemsets from data streams with the landmark model. It is based on reservoir sampling and has a probabilistic error guarantee. The empirical evaluation demonstrates clearly that it is not only faster than simply recomputing the set of relatively strongly closed sets from the data stream, but also achieves both a very good recall and a high precision. We further demonstrate the suitability of strongly closed sets for two potential applications. First, the classical problem of concept drift or change detection in data streams. Second, the problem of computer-aided product configuration recommendation.

Chapter 6 briefly summarizes the contributions of this thesis and discusses their merits and limitations. Finally, we mention some directions for future research.

Appendix A describes and documents the tuning of the hyper-parameters of the algorithms used in Chapter 4 in detail.

2. Notions and Problem Definitions

This chapter introduces the notions and notation used within this thesis and defines the problems considered therein. First, the classical definitions and notation from the itemset mining literature are introduced in Section 2.1, considering static fixed-size data sets. Data streams are the subject of Section 2.2. Their characteristics give rise to challenges not present in the static setting. Data streams can be analyzed under different window models which essentially differ in how they assign weights to the transactions in the stream. At the end of that section, the definitions for itemset mining from static data are redefined for the streaming setting. Finally, Section 2.3 introduces and defines the problems considered in this dissertation. They have in common that they consider data streams under the landmark model and mine frequent itemsets or parameterized subsets of these.

2.1. Itemset Mining

Motivated by the analysis of shopping baskets, the problem of frequent itemset mining was introduced by Agrawal et al. (1993). Given a large set of transactions, where each transaction consists of the items bought by a customer in one purchase, the goal of frequent itemset mining is to identify sets of items that are commonly bought together. Any combination of items is called an *itemset*. Given a set of items $I = \{i_1, i_2, \ldots, i_n\}$, both a transaction T and an itemset X are nonempty subsets of I. The transactions constitute the input and the itemsets are the output. The length or size of an itemset is defined by the number of items it contains. An itemset with k items is called a *k-itemset*. Each transaction is identified by a unique transaction id (TID). A transaction T is said to support an itemset X if $X \subseteq T$. Any collection of transactions is called a database of transactions or transactional data set. Given a database $\mathcal{D}$ of transactions, the *support set* of an itemset X in $\mathcal{D}$, denoted by $\mathcal{D}[X]$, is the set of all transactions in $\mathcal{D}$ which contain X, i.e., $\mathcal{D}[X] = \{T \in \mathcal{D} : X \subseteq T\}$. The *support count* of an itemset X in a database $\mathcal{D}$, denoted $sup_{\mathcal{D}}(X)$, is the number of transactions containing X, i.e., $sup_{\mathcal{D}}(X) = |\mathcal{D}[X]|$. We omit $\mathcal{D}$ when it is clear from the context, i.e., instead of $sup_{\mathcal{D}}(X)$ we will write $sup(X)$.
The problem of frequent itemset mining has received much attention in the data mining community, as it is a very fundamental task with many potential applications. These include association rule mining (Agrawal et al., 1993), fraud discovery, text mining, product recommendation, e-learning, and web clickstream analysis (Fournier-Viger et al., 2017) amongst others. Due to their easy interpretability, (interesting) itemsets are advantageous for applications where comprehensibility is amongst the requirements.

In contrast, sophisticated modeling techniques such as support vector machines (Vapnik and Chervonenkis, 1974) or artificial neural networks (Haykin, 1999) produce non-understandable black-box models. The primary challenge in frequent itemset mining is that for a set of n items there can be $2^n - 1$ frequent itemsets and hence, it is infeasible to enumerate all of them for large n. Even moderate sizes of n result in huge numbers of frequent itemsets. To reduce this enormous set to manageable sizes, different further measures of *interestingness* have been proposed, such as maximality (Mannila and Toivonen, 1997), closedness (Pasquier et al., 1999a), strong closedness (Boley et al., 2009a), high utility (Tseng et al., 2010), or crucialness (Das and Zaniolo, 2016).

A transactional data set is called *dense* if each transaction contains most of the items. It is called *sparse*, if each transaction contains only a few out of the many possible items. Transactional data sets can be assumed to be sparse or even very spare in general. The number of items bought by a customer in a single transaction is usually extremely small compared to the number of offered products. This is true for traditional supermarket data but even more so for online markets such as Amazon. Nonetheless, some transactional data sets of different origin are dense by nature. As an example, consider a system of sensors, where each sensor reports a value at every point in time. Such a system will produce a dense transactional data set as output. The *density* of a transactional data set or database $\mathcal{D}$ can be defined as

$$\sum_{T \in \mathcal{D}} \frac{|T|}{|I| \cdot |\mathcal{D}|} \quad ,$$

where $|T|$ denotes the number of items in transaction T, $|I|$ denotes the number of items, and $|\mathcal{D}|$ the number of transactions.

As it turns out, different encodings are suitable for sparse and dense data. Figure 2.1 illustrates the two most common approaches to orthogonally encode itemsets. Figure 2.1a shows a set of transactions. An itemset can be encoded explicitly as a set (Figure 2.1b) or as a bit-vector (Figure 2.1c). In the explicit set-based encoding the transaction ids (TIDs) of all supporting transactions are stored in the support set of an item. In the bit-vector representation, there is one bit-vector for each item[1]. The size of each bit-vector is identical to the number of transactions. The bit at position j of some item i is set to `true` if the item i occurs in transaction j. For sparse data sets, the set-based encoding is more efficient whereas for dense data sets the bit-vector representation is more compact. Besides, the computation of set intersections is faster for sets of bit-vectors than for sets of numbers. Computing such intersections is a common operation for the enumeration of itemsets and a fundamental step in many algorithms.

The $2^n - 1$ itemsets over a set I of items with $|I| = n$ together with the empty set form a lattice as illustrated in Figure 2.2. The number of patterns[2] is usually reduced via constraints on the measure of interest. The most widely used constraint is a user-defined frequency threshold $\theta \in (0, 1]$. The *frequency* of an itemset X in a database $\mathcal{D}$ is defined

[1] Note that bit-vectors can also be defined for each transaction, with a bit for each item.

[2] A pattern is a more general concept than an itemset. We use the terms interchangeably when there is no emphasis on itemsets.

TID	Items
1	ab
2	bc
3	ab
4	bcd

(a) transactions

Item	Support set
a	{1,3}
b	{1,2,3,4}
c	{2,4}
d	{4}

(b) set-based encoding

Item	Bitset			
a	T	F	T	F
b	T	T	T	T
c	F	T	F	T
d	F	F	F	T

(c) bit-vector encoding

Figure 2.1.: Illustration of itemset encodings.

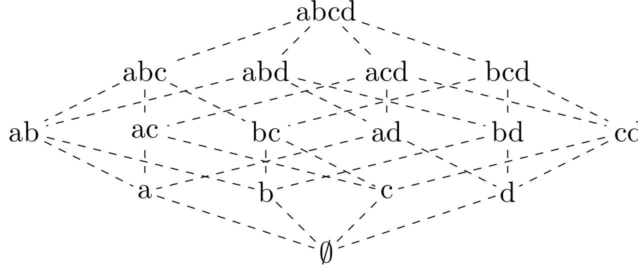


Figure 2.2.: Itemset lattice for the items a, b, c, d.

as

$$freq_{\mathcal{D}}(X) = \frac{sup_{\mathcal{D}}(X)}{|\mathcal{D}|} \quad ,$$

i.e., it is simply the support count of an itemset divided by the size of the data set. The frequency threshold partitions the pattern space into patterns that are less frequent than θ and those that occur at least as frequently as θ. Itemsets which are less frequent than θ are called *infrequent* and those which are at least as frequent as θ are the *frequent* itemsets. Formally, an itemset X is *frequent* in a database $\mathcal{D}$ for some user-defined frequency threshold $\theta \in (0, 1]$, if $sup_{\mathcal{D}}(X) \geq \theta|\mathcal{D}|$, where $|\mathcal{D}|$ denotes the cardinality of $\mathcal{D}$, i.e., the number of transactions in the database. For a database of fixed size, the frequency threshold θ can be expressed as an absolute minimum support threshold $minSup = \theta|\mathcal{D}|$. For a minimum support threshold, an itemset X is frequent in $\mathcal{D}$ if $sup_{\mathcal{D}}(X) \geq minSup$. While it appears purely technical at first to replace the relative frequency threshold θ with an absolute minimum support threshold $minSup$, the latter will be useful for the introduction of further concepts.

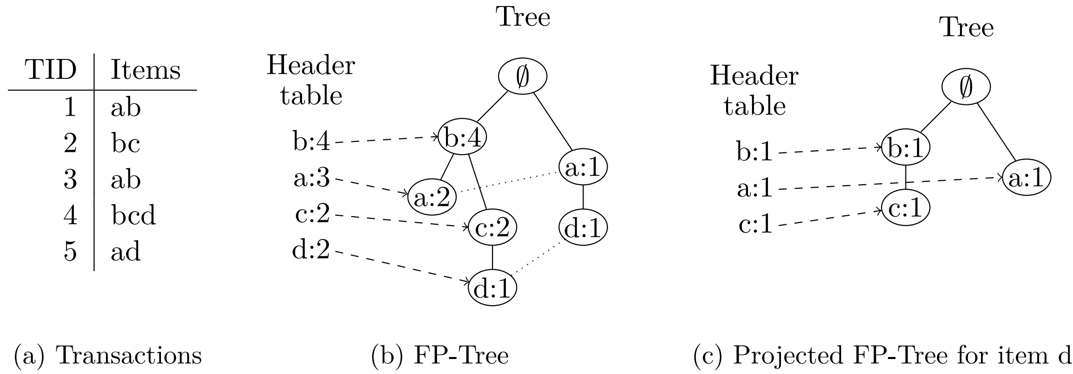
Let $\mathcal{F}$ denote the family of frequent itemsets for a database $\mathcal{D}$ and frequency threshold θ. Mannila and Toivonen (1997) introduced the concept of the *border* of $\mathcal{F}$, denoted $Bd(\mathcal{F})$. It consists of the itemsets X such that all proper (non-empty) subsets of X are frequent (i.e., belong to $\mathcal{F}$) and all proper supersets of X are infrequent (i.e., do not belong to $\mathcal{F}$). The sets X in $Bd(\mathcal{F})$ that are in $\mathcal{F}$ are called the *positive border* $Bd^+(\mathcal{F})$; those sets X in $Bd(\mathcal{F})$ that are not in $\mathcal{F}$ are the *negative border* $Bd^-(\mathcal{F})$. Thus, $Bd^+(\mathcal{F})$ and $Bd^-(\mathcal{F})$ form a partitioning of $Bd(\mathcal{F})$.

The two most recognized algorithms for the task of frequent itemset mining are called Apriori (Agrawal and Srikant, 1994) and FP-growth (Han et al., 2000). Apriori is a level-wise algorithm that scans the data set once for each itemset length. Starting with $k = 1$, it counts the support of the k-itemsets and then generates candidate $k + 1$-itemsets from the frequent k-itemsets. It then increments k and repeats the process until the set of candidates becomes empty. As pruning criteria for the candidate generation, the algorithm uses the following observation called *Apriori* or *downward closure property*: any subset of a frequent itemset must be frequent. Apriori has inspired many other algorithms. As it turns out, the candidate generation and repeated scanning of the transactions in the data set are the bottleneck of all Apriori-based algorithms. They work well as long as the frequent itemsets are small.

FP-growth (Han et al., 2000) works without candidate generation. It needs exactly two passes over the data set. In the first pass, the support count of all items is counted. The result is used to define a frequency descending total order on the items. The second pass is used to construct a frequent pattern tree (FP-Tree). This is a prefix tree data structure with an additional header table and links between nodes. Each node in the tree represents an item. Any path from the root to a node corresponds to the set of transactions containing all items along the path. Nodes representing the same item are linked. The header table contains one entry for each item. It stores the total support count of that item and maintains a pointer to the item in the tree. All other nodes of this item can be identified via the node links. Frequent itemsets are mined from the tree via recursive tree projections. A good reference for further details on FP-Trees and the FP-growth algorithm is (Borgelt, 2005).

As the FP-Tree is a central data structure in frequent itemset mining, it is illustrated with an example in Figure 2.3. The transactions on the left (Figure 2.3a) result in the tree in the middle (Figure 2.3b). The links from the header table are visualized as dashed lines and the node links as dotted lines. The tree projection for an item is computed by visiting all nodes of this item via the node links. For each visited node there is exactly one path to the root node. This path is added to the projected tree without the node of the item itself. The support count of parent nodes along this path is reduced to that of its children in the projection. The projected tree for item d is shown in Figure 2.3c.

One challenge in frequent itemset mining stems from the choice of the frequency threshold θ. How should a user choose this threshold for an unknown data set? Some initial knowledge about the data set at hand is required to pick a threshold neither too high nor too low. If the threshold is set too high, no or only very few patterns will be found. Highly frequent patterns are often trivial and known by domain experts. If the threshold is set too low, the output may contain too many patterns; typically much more than a user can inspect and it will take a long time to enumerate all those patterns. The standard approach to find an appropriate threshold is to start with a high value for θ and lower it until a good trade-off between the threshold value and the number of desired patterns has been found. A way to circumvent the selection of the threshold is to define the number k of output patterns in advance and return the k patterns with highest



TID	Items
1	ab
2	bc
3	ab
4	bcd
5	ad

(a) Transactions (b) FP-Tree (c) Projected FP-Tree for item d

Figure 2.3.: FP-Tree data structure.

frequency. This leads to the definition of the top-k frequent patterns: An itemset X is a *top-k* frequent itemset in a database $\mathcal{D}$ if there are less than k frequent itemsets with support count at least $sup_{\mathcal{D}}(X)$ in $\mathcal{D}$.

Mining top-k frequent itemsets seems easier at first. However, from a practitioner's point of view, it merely shifts the problem. For sure the output will contain the k most frequent patterns, but the topmost frequent itemsets are often known to business experts. Thus the challenge of choosing the *right* threshold remains an issue even for top-k pattern mining. It is arguable whether it is less severe for the parameter k than θ, as k has a clear semantic interpretation.

The major challenge in frequent itemset mining is the *exponential growth* in the number of frequent itemsets as the support threshold is lowered. This problem is sometimes referred to as *combinatorial explosion* (Chi et al., 2004). Recall that for n items there are up to $2^n - 1$ potentially frequent patterns. This means that already for 20 items, there are more than 1,000,000 patterns and for only 50 items there are more than 10^{15} potential frequent patterns. The tremendous output space can be reduced with additional constraints on the pattern class. Such constraints will be the subject of the next two sections. The first covers lossless representations of frequent itemset, i.e., which allow deriving all frequent itemsets with their exact support count. The second section describes lossy representations. They allow to reconstruct all frequent patterns, but with approximate support count only. For the patterns explicitly stored, the support count will be known; for all other frequent patterns, only a lower bound of their true support count is provided.

2.1.1. Closed and Crucial Itemsets

Two pattern classes have been defined, which result in lossless compression of the set of frequent patterns: closed and crucial patterns. The advantage of these pattern classes is that the output space is typically much smaller compared to frequent itemsets, while the support count of each frequent itemset can still be exactly derived. The more common class is the family of closed patterns (Pasquier et al., 1999a): An itemset X is *closed* in a database $\mathcal{D}$ if $\nexists Y \supset X$ with $sup_{\mathcal{D}}(Y) = sup_{\mathcal{D}}(X)$.

In other words, a pattern is closed if and only if each of its proper supersets has a strictly lower support count. Patterns that share the same support count with at least one superset are not part of the output; they can be reconstructed from it.

Closed sets can be enumerated efficiently, i.e., with polynomial delay (Ganter and Reuter, 1991; Gély, 2005). They can be characterized by *closure operators*. More precisely, let I be some finite set and $\sigma : 2^I \to 2^I$ be a function, where 2^I denotes the power set of I. Then σ is *extensive* if $X \subseteq \sigma(X)$, *monotone* if $X \subseteq Y$ implies $\sigma(X) \subseteq \sigma(Y)$, and *idempotent* if $\sigma(X) = \sigma(\sigma(X))$ for all $X, Y \subseteq I$. If σ is extensive and monotone, then it is a *preclosure*; if, in addition, it is idempotent, then it is a *closure* operator on I. It is a well-known fact (see, e.g., Davey and Priestley (2002)) that for the above definition of closedness there exists a closure operator σ such that X is closed if and only if $\sigma(X) = X$.

The set of closed patterns is usually further reduced by considering only the closed patterns that reach a certain frequency threshold θ, i.e., the *frequent closed* patterns, or the k most frequent closed, i.e., the *top-k closed* patterns. The definitions of both pattern classes are straightforward:

An itemset X is *frequent closed* in a database $\mathcal{D}$ if it is both frequent in $\mathcal{D}$ and closed in $\mathcal{D}$.

An itemset X is a *top-k closed* itemset in a database $\mathcal{D}$ if it is closed in $\mathcal{D}$ and there are less than k closed itemsets with support count greater than $sup_{\mathcal{D}}(X)$ in $\mathcal{D}$.

Crucial patterns are a rather recent notion introduced by Das and Zaniolo (2016). The idea of crucial patterns is to eliminate from the set of closed patterns those that have a support count which is identical to the sum of the support counts of the immediate closed supersets. The definition of this pattern class is technical and builds upon the Fp-Tree. We refer to Das and Zaniolo (2016) for the technical details. The definition will not be needed for this work.

The definitions are illustrated with a tiny example in Figure 2.4. Given the transactions in Figure 2.4a, the frequency threshold $\theta = 0.5$ and $k = 1$, the concepts of frequent, closed, frequent closed, top-k closed, and crucial itemsets are illustrated in Figures 2.4b to 2.4f, respectively. The concepts of strongly closed (Figure 2.4g) and maximal frequent itemsets (Figure 2.4h) will be introduced in the following section.

2.1.2. Strongly Closed and Maximal Frequent Itemsets

If it is not necessary to reconstruct the *exact* support count of all frequent itemsets, then there are pattern classes which compress the output even more than closed and crucial patterns; these pattern classes include *strongly closed* (Boley et al., 2009b) and *maximal frequent* itemsets (Mannila and Toivonen, 1997). The two pattern classes allow us to reconstruct all frequent itemsets without their exact support count. The exact support count is only known for the itemsets which belong to the output patterns.

Recall that a pattern is closed if the support count of all of its proper supersets is strictly lower than the support of the pattern itself. A difference of a single transaction suffices to make a pattern closed. This property is generalized in the following definition:

(a) `transactions`

TID	Items
1	ab
2	bc
3	ab
4	bcd

(b) `freq.`

Itemset	Sup
a	2
b	4
c	2
ab	2
bc	2

(c) `closed`

Itemset	Sup
b	4
ab	2
bc	2
bcd	1

(d) `freq.-closed`

Itemset	Sup
b	4
ab	2
bc	2

(e) `top-1 closed`

Itemset	Sup
b	4

(f) `crucial`

Itemset	Sup
ab	2
bc	2
bcd	1

(g) `2-closed`

Itemset	Sup
b	4
ab	2
bc	2

(h) `maximal freq.`

Itemset	Sup
ab	2
bc	2

Figure 2.4.: Illustration of the pattern class definitions. Transactions are shown in 2.4a. Let $\theta = 0.5$, $k = 1$, and $\Delta = 2$.

An itemset X is *strongly closed* or more precisely Δ-*closed* in a database $\mathcal{D}$ for some $\Delta > 0$ integer if for all Y with $X \subsetneq Y \subseteq I$ it holds that $sup_{\mathcal{D}}(Y) \leq sup_{\mathcal{D}}(X) - \Delta$.

Notice that ordinary closed itemsets are 1-closed. The number of strongly closed patterns shrinks fast with increasing Δ (cf. Boley et al. (2009a); Trabold and Horváth (2017)).

An itemset X is *strongly closed frequent* or more precisely Δ-*closed frequent* in a database $\mathcal{D}$ if it is both Δ-closed and frequent in $\mathcal{D}$.

We recall some basic algebraic and algorithmic properties of Δ-closed itemsets from Boley et al. (2009b). For a transaction database $\mathcal{D}$ over I and integer $\Delta > 0$, let $\hat{\sigma}_{\Delta,\mathcal{D}} : 2^I \to 2^I$ be defined by

$$\hat{\sigma}_{\Delta,\mathcal{D}}(X) = X \cup \{e \in I \setminus X : |\mathcal{D}[X]| - |\mathcal{D}[X \cup \{e\}]| < \Delta\}$$

for all $X \subseteq I$. It holds that $\hat{\sigma}_{\Delta,\mathcal{D}}$ is a preclosure on I that is not idempotent (Boley et al., 2009b). For an itemset $X \subseteq I$, consider the sequence

$$\hat{\sigma}^0_{\Delta,\mathcal{D}}(X), \hat{\sigma}^1_{\Delta,\mathcal{D}}(X), \hat{\sigma}^2_{\Delta,\mathcal{D}}(X), \ldots$$

with

$$\begin{aligned} \hat{\sigma}^0_{\Delta,\mathcal{D}}(X) &= X, \\ \hat{\sigma}^1_{\Delta,\mathcal{D}}(X) &= \hat{\sigma}_{\Delta,\mathcal{D}}(X), \text{ and} \\ \hat{\sigma}^{l+1}_{\Delta,\mathcal{D}}(X) &= \hat{\sigma}_{\Delta,\mathcal{D}}(\hat{\sigma}^l_{\Delta,\mathcal{D}}(X)) \end{aligned}$$

for all integer $l \geq 1$. This sequence has a smallest fixed point, giving rise to the following definition: For all $X \subseteq I$, let $\sigma_{\Delta,\mathcal{D}} : 2^I \to 2^I$ be defined by $\sigma_{\Delta,\mathcal{D}}(X) = \hat{\sigma}^k_{\Delta,\mathcal{D}}(X)$ with $k = \min\{l \in \mathbb{N} : \hat{\sigma}^l_{\Delta,\mathcal{D}}(X) = \hat{\sigma}^{l+1}_{\Delta,\mathcal{D}}(X)\}$. The proof of the claims in the theorem below can be found in Boley et al. (2009b).

Algorithm 1 Closure (Boley et al., 2009b)

input: $X \subseteq I$ and integer $\Delta > 0$
require: data set $\mathcal{D}$ over I
output: $\sigma_{\Delta,\mathcal{D}}(X)$

1: $C \leftarrow X$; $\mathcal{D}' \leftarrow \mathcal{D}[X]$
2: **repeat**
3: **for all** $e \in I \setminus C$ **do**
4: **if** $|\mathcal{D}'| - |\mathcal{D}'[e]| < \Delta$ **then** $C \leftarrow C \cup \{e\}$; $\mathcal{D}' \leftarrow \mathcal{D}'[e]$
5: **until** $\mathcal{D}'$ has not been changed in Loop 3–4
6: **return** C

Theorem 1. *Let $\mathcal{D}$ be a transaction database over some finite ground set I and $\Delta > 0$ an integer. Then (i) for all $X \subseteq I$, X is Δ-closed in $\mathcal{D}$ if and only if $X = \sigma_{\Delta,\mathcal{D}}(X)$, (ii) $\sigma_{\Delta,\mathcal{D}}$ is a closure operator over I, and (iii) for all $X \subseteq I$, the closure $\sigma_{\Delta,\mathcal{D}}(X)$ of X can be computed by Algorithm 1 in time $O(\|\mathcal{D}[X]\|_0)$, where $\|\mathcal{D}[X]\|_0 = \sum_{T \in \mathcal{D}[X]} |I \setminus T|$.*[3]

Using the fact that $\sigma_{\Delta,\mathcal{D}}$ is a closure operator, the family of all Δ-closed itemsets of a data set $\mathcal{D}$ can be enumerated by the following divide and conquer folklore algorithm (see, e.g., Gély (2005)): Generate first recursively all Δ-closed supersets of a set that contain a certain item $e \in I$, and then all that do not contain it. This algorithm lists all Δ-closed itemsets non-redundantly, with polynomial delay, and in polynomial space (see Boley et al. (2010) for some further properties of this algorithm).
Another approach to reduce the number of patterns is called maximal frequent itemsets (Mannila and Toivonen, 1997). These are precisely the patterns in the positive border of the frequent itemsets:

An itemset X is *maximal frequent* in a database $\mathcal{D}$ if it is frequent in $\mathcal{D}$ and all its proper supersets $Y \supset X$ are infrequent.

Obviously, maximal frequent itemsets form a subset of frequent itemsets. The patterns at the positive border are often less frequent than those closer to the bottom element of the lattice.
While strongly closed itemsets can be generated efficiently (i.e., with polynomial delay (Ganter and Reuter, 1991; Gély, 2005)), maximal frequent itemsets cannot be listed in output polynomial time (Boros et al., 2003), unless P = NP. This is an important advantage of strongly closed patterns.
The relationships among different pattern classes are illustrated in Figure 2.5. While

[3] In Boley et al. (2009b), two different algorithms are discussed for the computation of $\sigma_{\Delta,\mathcal{D}}$. They both have a time complexity of $O(\|\mathcal{D}[X]\|_0)$, assuming that empty transactions are not allowed. In practice, the two algorithms behave differently from the point of view of the running time, depending on the sparsity of $\mathcal{D}$ (cf. Boley et al. (2009b) for a detailed discussion).

$$\begin{array}{ccc} \texttt{strongly closed frequent} \subseteq & \texttt{frequent closed} & \subseteq \texttt{frequent} \\ & \cup\!| & \\ & \texttt{crucial frequent} & \\ & \cup\!| & \\ & \texttt{maximal frequent} & \end{array}$$

Figure 2.5.: Relationships between frequent pattern classes.

transactions

TID	Items
1	ab
2	bc
3	ab

crucial

Itemset	Sup
ab	2
b	3

2-closed

Itemset	Sup
ab	2

(a) `crucial` $\supset$ `2-closed`

transactions

TID	Items
1	ab
2	bc
3	ab
4	bc

crucial

Itemset	Sup
ab	2
bc	2

2-closed

Itemset	Sup
b	4
ab	2
bc	2

(b) `crucial` $\subset$ `2-closed`

Figure 2.6.: Examples illustrating the relationship between crucial and strongly closed patterns with minSup = 2. For the transactions on the left the crucial patterns are a superset of the 2-closed, for the transactions on the right the relation is reversed i.e. the crucial patterns are a subset of the 2-closed.

there are some clear subset relationships, crucial, maximal, and strongly closed patterns are incomparable in general with respect to containment. This fact is illustrated for crucial frequent and strongly closed frequent patterns in the example in Figure 2.6. On the left, the crucial patterns form a superset of the 2-closed patterns, whereas on the right, with one additional transaction, the crucial patterns become a subset of the 2-closed patterns.

Given the various pattern classes, the family of strongly closed frequent patterns is the one that is effectively controllable by the user without any language bias (e.g., preferring short patterns by increasing the frequency threshold). The size of this pattern class can be controlled with the frequency threshold θ and the strength of the closure Δ, whereas the other frequent pattern classes can be parameterized solely by θ.

2.2. Data Streams

This section introduces the necessary background from data stream mining. We start with some basic properties of data streams and discuss how they differ from traditional (batch) data sets in Section 2.2.1. In particular, data streams give rise to several challenges not present in the static setting. For the analysis of such data streams, different window models have been proposed in the literature. They differ in the way they consider the past of the stream and are subject of Section 2.2.2. After this general introduction to data streams, we focus on transactions. Transactional data streams are covered in Section 2.2.3, where the definitions from Section 2.1 are adopted for data streams.

2.2.1. Properties of Data Streams

While most algorithms for data mining work on static *batch* data sets (i.e., data sets which reside on disk and can be accessed multiple times), many real-life applications produce continuous data streams. Examples of such natural data streams come from industrial applications such as, call detail record analysis, fraud detection, sensor monitoring, recommendations, or network security (Bhattacharyya et al., 2011; Etzion et al., 2016; Zhang and Zhou, 2004). A data stream is regarded as a potentially unbounded sequence of elements. The nature of the elements that compose the stream is application-specific. For example, transactional systems produce transactional streams, where each element is a transaction, sensor systems produce streams of sensor readings, and agent-based systems produce streams of actions of the agents. Typical examples of streaming environments will illustrate the additional challenges arising from data streams in comparison to the batch setting.

In telecommunication, service providers monitor the use of their network for various reasons, for example, to control service quality, to identify upcoming bottlenecks, or to identify abusive use of their network in near real-time. In phone fraud detection, the goal is to "identify users who use a network service without the intention to pay for that use" (Etzion et al., 2016). The faster such abusive use is detected, the lower the loss for the company. As the telecommunication network is used continuously, the next fraudulent call can happen at any time.

In finance, multiple payment transactions are being processed every second. Only very few, typically one in several thousand transactions are fraudulent (Bhattacharyya et al., 2011). To detect them, the stream of transactions needs to be monitored permanently. When a successful fraud pattern is discovered, transactions conforming to that pattern will not be authorized for payment. As specific behavior is effectively blocked, fraudsters continuously adapt their strategy and look for new ways to circumvent the fraud prevention system. The system has no control over the order in which the transactions arrive. The next transaction can be legitimate or fraudulent.

In Industry 4.0 and environmental observations (Mainwaring et al., 2002), sensor monitoring becomes increasingly important. Large sensor systems are deployed to monitor either machines or the environment, often with the goal of a better understanding of the dynamics of the monitored system. Each sensor in such a network is one source of

data of a data stream. While the memory on each sensor is limited, the observed system provides an unbounded source of data. As new data comes in, old data must be removed from the memory to make room for the new data.
Online market places, content providers, and social media optimize their service with recommendation engines (Ricci et al., 2011). These systems suggest products or content based on the behavior of other users. Consider as examples the platforms YouTube (Davidson et al., 2010) for videos or Twitter (Kwak et al., 2010) for messaging. Each video or message in these systems is an element of an ever-growing stream of content. A topic shared and discussed by several users today might be of little interest next week. Thus, such systems need to constantly adapt to the changes in the data stream.
Computer networks exchange packets with each other at high rates. Modern network security solutions monitor the network traffic to identify attackers or temporary block certain origins or destinations. Network monitoring systems (Cranor et al., 2002) need to analyze the network stream in real-time to react fast to any incident. One particular type of network attacks are denial of service attacks. The goal of these attacks is to flood a system with more requests than it can handle such that it will break down. The attack is successful if the data arrival rate surpasses the processing time to handle each request.
These examples illustrate that data streams differ from traditional static data sets stored on disk in several ways. The main characteristics of data streams include (cf. Babcock et al. (2002); Golab and Özsu (2003)):

- Data elements arrive continuously.
- The system has no control over the order in which the elements arrive.
- A data stream is potentially unbounded in size.
- Every element from the stream can only reside for a short time in memory.
- The data distribution of a stream can change over time.
- The data arrival rate can surpass the processing speed of an algorithm.

The potentially unbounded nature of data streams and the limited physical memory of any computer demand for the development of algorithms that work with limited resources, especially with limited main memory. Even if an entire stream can be stored on external memory, the analysis of all the data takes too long to keep up with the arrival rate of new data. The *focus* of this thesis lies therefore on algorithms that use only the physical main memory of a computer to track and store information from the stream and to analyze it. Data compression in the form of compact synopsis structures is the standard way to reduce the amount of memory required. Various approaches have been suggested for data compression, including sketches (Flajolet and Martin, 1985), sampling (Vitter, 1985), and histograms (Guha et al., 2001). All these methods discard, however, some information.

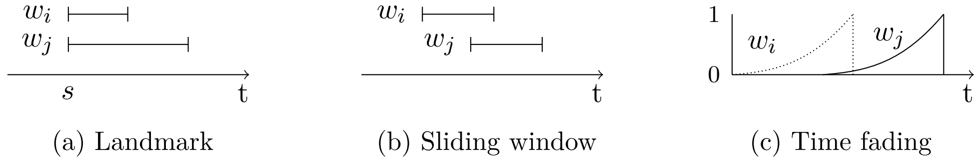


Figure 2.7.: Illustration of the window models with two windows w_i and w_j. For the time fading model, the height indicates the weight of each transaction.

Irrespective of the technology used to reduce the amount of data, streaming algorithms operate on a subset of the data from the stream due to the limited memory. Such algorithms are therefore often approximate, as they can not access past data, which is no longer kept in memory. Sophisticated inference mechanisms are used to derive some information from the synopsis about the data which is not explicitly encoded in the synopsis. Different existing synopses will be the subject of Chapter 3 and new synopses will be presented in Chapters 4 and 5.

2.2.2. Data Stream Models

For the analysis of data streams, different window models have been proposed in the literature. A window model defines the set of elements from the data stream which constitute the input of the analysis. The models differ in the development of the size of the window they consider and in the way they assign a weight to each element from the stream. The three most important window models are illustrated in Figure 2.7.

The *landmark* model (Zhu and Shasha, 2002) considers all elements of the stream from some fixed landmark start time s up to the current element t (see Figure 2.7a). This implies that the window gets larger as more elements arrive in the data stream. Each element is assigned the same weight. As a consequence, the influence of each element on the final result gradually vanishes as the window gets larger because with increasing window size, each element represents a smaller share of the entire window. The growing nature of the window makes the design of algorithms for this model challenging because the available memory is limited and the data streams are assumed to be unbounded.

The *sliding window* model (Datar et al., 2002) considers the k most recent elements for some integer $k > 0$. In this model, the window has a fixed size (see Figure 2.7b). The first k elements are added to the window. From the $k+1$-st element onward, whenever a new element arrives the oldest element is deleted from the window. Once the window is completely filled, the elements within the window all have the same constant weight $1/k$. Elements outside the window have zero weight.

The *time-sensitive sliding window* model (Lin et al., 2005) is a variation of the sliding window model. It considers a temporal window of a fixed size which is moved forward as time passes. The number of elements arriving at each point in time can vary in this model. In contrast to the regular sliding window model discussed above, the number of

elements arriving and leaving the window is neither identical nor fixed. It is impossible to predict the relative weight of a transaction under this model, as it depends both on the arriving and the leaving transactions.

The *time fading* or *damped* model (Zhu and Shasha, 2002), sometimes referred to as the *time decayed* or *time-tilted* model, is a model in which elements are associated with different weights. In this model, more recent elements have a higher weight than older ones (see Figure 2.7c). With each new arriving element, the weight of all elements decays by a factor such that elements seen long ago will eventually be removed from consideration. The number of elements considered is limited. This model is sometimes regarded as a variation of the landmark model (e.g., Gupta et al. (2010)). At the same time, it shares the characteristic that old elements will eventually be discarded with the sliding window model.

Irrespective of the window model, the data in the window changes as new elements arrive in the data stream and the learned model needs to be updated to reflect the change. The naive solution would be to compute the output *from scratch* for the changed window. This approach has *high costs* in terms of the time required to compute the update. *Incremental* solutions are typically *faster*; they either update the model for each new incoming element or fill a buffer with incoming elements and process all elements from the buffer when the buffer is full or upon request. The first approach is called element-based update or more specifically, *transaction-based update* when the elements in the data stream are transactions. The second approach is often referred to as *mini-batch* since the elements from the buffer are processed in small batches. Element-based updates provide the most accurate result reflecting the change of every element. However, for large windows, the result will change at most slightly with each new element. It is thus questionable whether such element-wise updates are needed. Computing updates in mini-batches is often faster because there are fewer invocations of the update operation and because the update with mini-batches can often be implemented more efficiently. In the case of frequent itemset mining, frequency-based pruning is more effective for larger mini-batches.

After the discussion of the different models, we now focus on transactional data streams.

2.2.3. Transactional Data Streams

A *transactional data stream* is a data stream in which the elements are transactions. Formally, a transactional data stream $\mathcal{S}$ is a potentially unbounded sequence of transactions $\mathcal{S} = \langle T_1, T_2, \ldots, \rangle$. Transaction T_1 is the first and oldest transaction. The transactions can vary in their size as is the case for market basket transactions or they can have a fixed size such as payment transactions or call detail records. The terms stream, data stream, and transactional data streams will be used henceforth as synonyms.

All the definitions in Section 2.1 have been stated for static (fixed size) data sets. They can be easily formulated for data streams. Consider a potentially unbounded data stream $\mathcal{S} = \langle T_1, T_2, \ldots \rangle$, where each transaction T_i contains an arbitrary number of elements of the ground set I, i.e., $T_i \subseteq I$. The finite subsequence of the first $t \in \mathbb{N}$ transactions is denoted as $\mathcal{S}_t = \langle T_1, T_2, \ldots, T_t \rangle$. The support set and support count of itemsets for

finite data streams can be defined similarly to those of fixed size databases. The support set of an itemset X in $\mathcal{S}_t$, denoted by $\mathcal{S}_t[X]$, is defined as the set $\{T \in \mathcal{S}_t : X \subseteq T\}$. The support count of X in $\mathcal{S}_t$, denoted as $sup_{\mathcal{S}_t}(X)$, is the cardinality of $\mathcal{S}_t[X]$, i.e., $sup_{\mathcal{S}_t}(X) = |\mathcal{S}_t[X]|$. With the definitions of (unbounded) data streams and those for the support set and support count of an itemset in such a stream, all the above definitions for itemsets are applicable to transactional data streams. It suffices to replace the database $\mathcal{D}$ with the finite transactional data stream $\mathcal{S}_t$ and $sup_{\mathcal{D}}(X)$ with $sup_{\mathcal{S}_t}(X)$ in each definition to obtain the definition for the streaming case. With $\mathcal{F}_{t,\theta}$ we denote the family of frequent itemsets at time t for the stream $\mathcal{S}_t$ and the frequency threshold θ. Formally, $\mathcal{F}_{t,\theta} = \{X \subseteq I : sup_{\mathcal{S}_t}(X)/|\mathcal{S}_t| \geq \theta\}$.
Mining a stream under the landmark model imposes an additional challenge when it comes to closed itemsets or strongly closed itemsets. As the window gets larger, more and more itemsets will become closed or strongly closed. For very long streams, the set of frequent closed itemsets might be almost as large as the set of frequent itemsets. Accordingly, the effect of compression decreases.

2.3. Problem Definitions

From the different window models for data stream mining, the landmark model is the most challenging, as it considers a growing number of transactions from the stream. The problems considered in this thesis all concern mining frequent itemsets or subsets of frequent itemsets from data streams under the *landmark* model with error guarantees independent of the size of the update. That is, we explicitly decouple the error bound from the update interval to combine the speed of mini-batch-based algorithms with the flexibility of transaction by transaction enumeration. We consider the following problems:

1. Mining *frequent* itemsets from a stream of transactions generated by a fixed, but unknown distribution.

2. Mining *frequent* itemsets from transactions generated independently.

3. Mining *strongly closed* itemsets from transactions generated independently.

All problems consider a data stream $\mathcal{S} = \langle T_1, T_2, \ldots \rangle$ and single-pass algorithms. The problems differ in the process generating the transactions T_i and the set of patterns mined from $\mathcal{S}$.
The first problem is concerned with transactions T_i generated independently according to some *fixed*, but *unknown* distribution $\mathcal{D} : 2^I \to [0, 1]$. We write $D \sim \mathcal{D}$ to indicate that the random variable D is distributed according to $\mathcal{D}$. Formally, we have $T_i \subseteq I$ and $T_i \sim \mathcal{D}$ for each point in time $i \in \mathbb{N}$. Given a frequency threshold θ and a single pass over the data stream, the goal is to enumerate all itemsets X that are frequent in $\mathcal{S}_{t'} = \langle T_1, T_2, \ldots, T_{t'} \rangle$, i.e., $sup_{\mathcal{S}_{t'}}(X) \geq \theta$, for each point in time $t' \in \mathbb{N}$. This problem is defined formally as follows:

Problem 1: Mining frequent itemsets from streams with fixed unknown distribution:
Given some *fixed*, but *unknown* distribution $\mathcal{D} : 2^I \to [0,1]$, a potentially unbounded data stream $S = \langle T_1, T_2, \ldots \rangle$ with transactions T_i generated independently at random according to $\mathcal{D}$, i.e., $T_i \sim \mathcal{D}$, an integer $t' \in \mathbb{N}$ defining the sequence $\mathcal{S}_{t'} = \langle T_1, T_2, \ldots, T_{t'} \rangle$, and a threshold $\theta \in (0,1]$, *return* all frequent itemsets $X \subseteq 2^I$, i.e., itemsets which satisfy $sup_{\mathcal{S}_{t'}}(X) \geq \theta$.

Fixed, but unknown distributions are common in human behavior at the subconscious level and in natural environmental processes. Examples are written texts and spoken language. Each sentence is a transaction and the words are the items. The combination of words to form sentences is coined by factors such as origin, education, social status, and others. The analysis of such data can reveal information about the author or a group of authors. Koppel et al. (2005) use features extracted from texts to identify an author's native language. Alzahrani et al. (2012) survey the work on plagiarism detection in natural languages and programming languages. The individual distribution of word combinations, i.e., frequent itemsets from fixed but unknown distributions, can help to solve both of the above problems. A solution for the identification of fixed unknown distributions is the subject of Section 4.2.
Another related problem is the analysis of streams with independently generated transactions. In many application domains, transactions originate from several independent sources. Consider the classical setting of market basket analysis. The stream of sales transactions is generated by many customers, each with individual needs. In such settings, there is no fixed distribution. The transactions T_i are instead generated *independently* by some *unknown* probability distribution $\mathcal{D}_i$ over the space $2^I \setminus \{\emptyset\}$ of all transactions. For all $t \in \mathbb{N}$, let $[t]$ denote the set $\{1, \ldots, t\}$. For any non-empty itemset $X \subseteq I$ and $i \in [t]$, this random generation of the transactions gives rise to the probability $p_i(X)$ that X is a subset of T_i, i.e., to

$$p_i(X) = \mathbf{Pr}_{T \in \mathcal{D}_i}[X \subseteq T]$$

where the subscript on the right-hand side indicates that the probability is taken w.r.t. the random generation of T according to $\mathcal{D}_i$. For any $X \subseteq I$ and $i \in [t]$, let X_i be the binary random variable indicating whether or not $X \subseteq T_i$. It follows from the definitions above that the X_is are independent Poisson trials with success probability $\Pr(X_i = 1) = p_i(X)$ that is *unknown* for the mining algorithm.
Given a data stream $\mathcal{S}_t = \langle T_1, T_2, \ldots, T_t \rangle$, a threshold $\theta \in [0,1]$ specified by the user, and some $t' \in [t]$, an itemset $X \subseteq I$ is called *interesting* w.r.t. $\mathcal{S}_{t'}$ if $p^{(t')}(X) \geq \theta$, where $p^{(t')}(X)$ is the average success probability of X in $\mathcal{S}_{t'}$, i.e.,

$$p^{(t')}(X) = \frac{p_1(X) + \ldots + p_{t'}(X)}{t'} \quad ;$$

X is called *frequent* w.r.t. $\mathcal{S}_{t'}$ if $f^{(t')}(X) \geq \theta$, where $f^{(t')}(X)$ is the relative frequency of X in $\mathcal{S}_{t'}$, i.e.,

$$f^{(t')}(X) = \frac{X_1 + \ldots + X_{t'}}{t'} \quad .$$

We note that the above notion of *interestingness* strongly relates to that of *frequency*. Indeed, according to Poisson's theorem, for any $\epsilon > 0$,

$$\lim_{t' \to \infty} \Pr\left(\left|f^{(t')}(X) - p^{(t')}(X)\right| < \epsilon\right) = 1 \ ,$$

i.e., the probability that the (relative) frequency $f^{(t')}(X)$ of X in $\mathcal{S}_{t'}$ arbitrarily approximates to the average success probability $p^{(t')}(X)$ tends to 1 when $t' \to \infty$.
Recall the broad range of applications ranging from bioinformatics to webstream analysis (Fournier-Viger et al., 2017) of frequent itemsets from Section 2.1. These patterns are a human-understandable form of knowledge. Algorithms mining such patterns from data streams are of high practical importance for those who are interested in understanding the characteristics or patterns of these streams. Motivated by this high practical relevance, the problems of answering interestingness and frequency queries for any $X \subseteq I$, at any time point $t' > 0$, are considered in this thesis. More precisely, we consider the following two problems:

Problem 2: Finding interesting itemsets from data streams: *Given* a single pass over a data stream $\mathcal{S}_t = \langle T_1, \ldots, T_t \rangle$ with transactions generated independently at random according to some unknown distributions $\mathcal{D}_1, \ldots, \mathcal{D}_t$, an integer $t' \in [t]$, a threshold $\theta \in (0, 1]$, and a query of the form *"Is X interesting w.r.t. $\mathcal{S}_{t'}$?"* for some $X \subseteq I$, *return* TRUE along with $p^{(t')}(X)$ if X is interesting w.r.t. $\mathcal{S}_{t'}$; FALSE otherwise.

Problem 3: Listing frequent itemsets from data streams: *Given* a single pass over a data stream $\mathcal{S}_t = \langle T_1, \ldots, T_t \rangle$ of transactions, an integer $t' \in [t]$, a threshold $\theta \in (0, 1]$, and a query of the form *"Is X frequent w.r.t. $\mathcal{S}_{t'}$?"* for some $X \subseteq I$, *return* TRUE along with $f^{(t')}(X)$ if X is frequent w.r.t. $\mathcal{S}_{t'}$; FALSE otherwise.

Notice the difference between Problems 2 and 3. While Problem 2 assumes that the transactions T_i are generated by independent unknown distributions $\mathcal{D}_i$, Problem 3 does not assume this independence. For both problems, we assume that the queries are received by the algorithm at time t', that the data stream is traversed in one pass, and that the full set of transactions the algorithm has already seen cannot be stored by the algorithm. Note that these two problems are more general than Problem 1, as they consider data streams with transactions generated from unknown independent probability distributions, instead of a single unknown probability distribution. Solutions for these two problems will be presented in Section 4.3.
Despite their practical relevance, the problems considered so far neglect the various techniques proposed to reduce the size of the output. Smaller outputs have at least two benefits. For human experts smaller sets are easier to understand. On the algorithmic side, a smaller output might require less summary information and could thus be beneficial for the streaming setting where memory is limited. Which of the various compression techniques is most suitable? Closed, crucial, and strongly closed patterns face the challenge that the longer the stream is processed under the landmark model the more itemsets will become closed, crucial, or Δ-closed for a fixed (absolute) Δ with high

probability. Maximal patterns, on the other hand, can not be enumerated efficiently (Boros et al., 2003). Strongly closed patterns have a number of advantages. First of all, the parameter Δ provides an additional level of flexibility over the other pattern classes. Second, ordinary closed patterns are a special case of strongly closed patterns: It suffices to set $\Delta = 1$ for ordinary closed patterns. Finally, with the help of the closure operator, closed patterns can be mined in output polynomial time (Boley et al., 2010). Since each strongly closed set corresponds to a sharp drop in frequency, they represent the itemsets that stand apart and are more stable against small changes in the input stream than ordinary closed sets for example. In practice, strongly closed sets form a *compact*, often tiny subset of the family of frequent patterns, which makes them especially useful in the context of stream mining. Still, the challenge remains that the number of strongly closed patterns grows with the stream length. This problem can, however, be circumvented by considering a relative strength $\tilde{\Delta} \in (0, 1]$ instead of the absolute one Δ. It removes the problem of absolute strength for growing data streams, as it is always relative to the stream length. More precisely, for a threshold $\tilde{\Delta} \in (0, 1]$, an itemset $X \subseteq I$ is *relatively* $\tilde{\Delta}$*-closed* in $\mathcal{S}_t$ if

$$\frac{|\mathcal{S}_t[X]|}{|\mathcal{S}_t|} - \frac{|\mathcal{S}_t[Y]|}{|\mathcal{S}_t|} \geq \tilde{\Delta} \qquad (2.1)$$

holds for all Y with $X \subsetneq Y \subseteq I$. For $\tilde{\Delta}$-closed itemsets, any proper extension of an itemset X must decrease its relative frequency by at least $\tilde{\Delta}$. Thus, $\tilde{\Delta}$ indicates the relative *strength* of the closure. If it is clear from the context, the adverb "relatively" will be omitted. Relatively $\tilde{\Delta}$-closed itemsets correspond to those itemsets with a sharp drop in relative frequency. For market basket data, each such set might describe a customer segment, while most frequent sets do not define meaningful customer groups. Motivated by different real-world applications (e.g., concept drift detection, computer-aided product configuration discussed in Section 5.4), we consider the following problem:

Problem 4: Listing $\tilde{\Delta}$-closed sets from data streams: *Given* a single pass over a data stream $\mathcal{S}_t = \langle T_1, T_2, \ldots, T_t \rangle$ over a set I of items, a threshold $\tilde{\Delta} \in (0, 1]$, and an integer $t' \in [t]$, *list* all itemsets $X \subseteq I$ that are $\tilde{\Delta}$-closed in $\mathcal{S}_{t'} = \langle T_1, T_2, \ldots, T_{t'} \rangle$.

A solution to this problem is the subject of Chapter 5. Note that the definition of *relative* $\tilde{\Delta}$-closedness for $\mathcal{S}_t$ above can equivalently be reformulated by that of *absolute* Δ-closedness (Boley et al., 2009b) as follows: X is relatively $\tilde{\Delta}$*-closed* in $\mathcal{S}_t$ if and only if it is absolutely Δ*-closed* in $\mathcal{S}_t$ for $\Delta = \lceil t\tilde{\Delta} \rceil$, that is, $|\mathcal{S}_t[X]| - |\mathcal{S}_t[Y]| \geq \Delta$ for all Y with $X \subsetneq Y \subseteq I$. In general, the family of Δ-closed itemsets in a transaction database $\mathcal{D}$ is denoted by $\mathcal{C}_{\Delta,\mathcal{D}}$. In particular, the family of Δ-closed itemsets in $\mathcal{S}_t$ is denoted by $\mathcal{C}_{\Delta,\mathcal{S}_t}$.

3. Related Work

This chapter serves as a short *survey* on the state-of-the-art algorithms mining frequent itemsets from transactional data streams. We start with a brief introduction to the topic in Section 3.1. The three main sections of the chapter focus on the various window models for data stream mining. In particular, we discuss algorithms for the *landmark model* in Section 3.2, those for the *sliding window model* in Section 3.3, and finally an algorithm for the *time fading model* in Section 3.4. In Section 3.5, we compile a taxonomy of the algorithms and a compact table with their key characteristics to facilitate the selection of an algorithm with desired properties. We conclude this chapter in Section 3.6 with a brief summary.

3.1. Frequent Itemset Mining from Data Streams

For a good survey on mining data streams, the reader is referred to Gaber et al. (2005). This review paper covers general concepts for data stream mining, such as approximation schemes like sketches, and considers different mining techniques like clustering, classification, frequent itemset mining, and time series analysis. The survey by Gama (2012) has a similar focus. It is less all-embracing concerning the mining techniques but covers one specific algorithm for frequent pattern mining more detailed. While both surveys provide an overview of data stream mining in general, they do not cover the relevant literature on frequent itemset mining from data streams. A survey focusing on frequent itemset mining from transactional data streams is provided by Cheng et al. (2008). It encompasses a total of nine algorithms, including one mining closed itemsets and one with a rare time model (Giannella et al., 2003). The majority of the algorithms described considers the landmark model. We discuss several additional algorithms, all mining frequent itemsets from data streams.

Recall from Section 2.2.2 that the difficulty of mining data streams depends on the particular window model considered. For the landmark model, any single-pass algorithm is approximate in nature, i.e., the output can contain more than all frequent itemsets or not all of them (Hidber, 1999). The central problem is that it is infeasible to count all itemsets. However, the frequency of itemsets can change and hence, infrequent itemsets that have not been counted may become frequent. Many algorithms use techniques to approximate the frequency of itemsets for the past, in which they have not been counted, once they start counting them. Though in this dissertation we focus on the landmark model only, for completeness we overview the sliding window and the time fading models as well. Algorithms for these two window models can produce exact results as long as the entire window fits in memory, which is generally assumed for these algorithms.

TID	Items
1	ab
2	bc
3	ab
4	bcd

Figure 3.1.: Example transactional data stream.

To illustrate some of the data structures, we introduce a tiny data stream with four transactions in Figure 3.1. Each transaction is identified by a transaction id (TID). We will refer to this example whenever needed.

3.2. Landmark Algorithms

In this section, we describe nine prominent algorithms mining frequent itemsets from transactional data streams under the landmark model.

Carma

The algorithm Carma (Hidber, 1999) is a two-pass algorithm consisting of two distinct phases. Each phase requires one pass over the data. The first phase computes a superset of the truly frequent itemsets. The second phase determines the exact support count for each itemset identified in phase one. This allows the algorithm to return the exact result after the second pass. The following description focuses solely on the first phase because the focus of this work is on single-pass algorithms. The algorithm is designed for dynamically changing frequency thresholds during the process of mining the data stream. Because this detail is neglected by all other algorithms, we will describe Carma in a simplified version for a fixed threshold. The algorithm maintains an itemset lattice as its data structure. For each arriving transaction, it (i) updates the count of all itemsets already in the lattice and (ii) adds new itemsets if all of their subsets are in the lattice and frequent. Pruning happens not for every transaction, but at regular intervals after $\max(\lceil 1/\theta \rceil, 500)$ transactions have been processed. The number 500 is chosen arbitrarily by the author of Carma. Pruning removes all itemsets with cardinality ≥ 2 from the lattice which have become infrequent. In contrast, 1-itemsets are never pruned, which implies that their counts are always exact. For each itemset in the lattice, the algorithm maintains three numbers: (i) the time t when the itemset was added to the lattice, (ii) the count of the item since t, and (iii) the maximal missed count of the itemset. The maximal missed count is derived from the maximal missed count of all subsets and based on the frequency threshold, whichever is lower determines the true maximal missed count. More precisely the maximal missed count for an itemset X is defined by

$$\text{maxMissed}(X) = \min \left(\min_{Y \subset X} \left(\text{maxMissed}(Y) + \text{count}(Y) - 1 \right), \lfloor (t-1)\theta \rfloor + |X| - 1 \right) ,$$

where count(Y) denotes the support count of itemset Y in the buffer and $|X|$ the cardinality of X. In case of dynamic frequency thresholds, the term $\lfloor (t-1)\theta \rfloor$ needs to take the changes of the threshold into account (see Hidber (1999) for the details).

`Discussion` CARMA explicitly assumes that the user changes the frequency threshold during the mining process and considers such changes. Most other authors do not consider such changes explicitly, but rather assume that the threshold is either not changed or the algorithm works with a new threshold, once it has been adjusted by the user. Many algorithms require additional parameters to control the error. CARMA requires no such additional parameter, which makes its application easy compared to such algorithms where the output depends on the choice of additional parameters. This simple design, however, limits the control of the analyst over the algorithm. The PARTIAL COUNTING algorithm of this thesis has been inspired by the design of CARMA and in particular, by the recursion of approximating the maximal missed transactions for an itemset. Instead of the fixed approximation scheme that does not improve as more transactions arrive, several approximations will be proposed in Section 4.2. Two of these become more accurate as the number of transactions increases.

Lossy Counting

The LOSSY COUNTING algorithm by Manku and Motwani (2002) is perhaps the most recognized algorithm in this field. Using an additional error parameter $0 < \epsilon < \theta$, typically with $\epsilon \ll \theta$, it provides the following guarantees: (i) The algorithm is complete, i.e., it returns all frequent itemsets, (ii) no itemset with frequency below $(\theta - \epsilon)t$ is generated, where t is the length of the data stream, and (iii) all estimated frequencies are less than the true frequencies by at most $\epsilon \cdot t$.
LOSSY COUNTING first buffers the transactions in as many blocks as memory available, each of size $\left\lceil \frac{1}{\epsilon} \right\rceil$ and processes then all buffered blocks together as a single batch. The number of stored itemsets does not depend on θ but on the number of buffered blocks, denoted by β. The algorithm starts counting all itemsets which occur with a support count of at least β in the buffer. The larger β the fewer itemsets the algorithm counts. It is thus important that β be a large number. To reduce the number of considered candidate itemsets, the algorithm first identifies all items which are less frequent than β in the buffer. An itemset is no longer counted, if at some point its frequency is below or equal to the number of buckets processed so far. If an itemset was not counted for the past, the maximal error in the count of the itemset is the number of buckets processed before the current batch. This error is recorded separately and added to the frequency count. All itemsets are stored in a trie structure and updated with each buffer. To save memory, the trie is stored without the usual pointers in tree structures but encoded compactly in a single array. The nodes are arranged in the array corresponding to a pre-order traversal of the tree and always updated in this order by the algorithm. The set of frequent patterns is produced from the trie reporting all patterns with frequency $f \geq (\theta - \epsilon) \cdot t$.

`Discussion` The error guarantee introduced with LOSSY COUNTING has inspired many other algorithms to provide the same guarantee with different designs. In this respect the algorithm is seminal for the field.
The storage of itemsets depends on the number of blocks in the buffer, the size of the buffer on the available memory, and the size of each block on the parameter ϵ. The storage space is thus independent of the threshold θ. If ϵ is set to a constant factor of θ, as suggested by the authors, then the threshold indirectly determines how many blocks fit in a given amount of memory. For a small number of blocks, a huge amount of unnecessary information is stored by this algorithm.

FDPM

FDPM by Yu et al. (2004) takes two user-defined parameters to control reliability (δ) and the number of blocks in memory (k). It guarantees the following: (i) all itemsets with frequency at least θ are in the result with probability $1-\delta$, (ii) no itemset with frequency less than θ is in the result, and (iii) the probability that the estimated support count for an itemset is identical to the true support count is at least $1-\delta$.
The algorithm processes k blocks, each of size $(2+2\ln(2/\delta))/\theta$. A larger k reduces the runtime, at the expense of space. It keeps two structures, the set of frequent patterns in the current block and the set of overall frequent patterns. For each block, it first identifies potentially frequent patterns, then merges them with the overall frequent patterns, and prunes the set of overall frequent patterns, whenever a condition on the size of this set holds. For a data stream $\mathcal{S}_t$, all itemsets with frequency at least $\theta-\epsilon_t$ with $\epsilon_t = \sqrt{(2\theta\ln(2/\delta))/t}$ are kept in memory, where ϵ_t is derived from Chernoff bounds. FDPM is sound but incomplete.

`Discussion` The algorithm uses a dynamic error ϵ_t which decreases as the stream gets longer. In contrast to many related algorithms, the tolerated error on the mined patterns gets thus smaller as the stream gets longer. This property is very desirable for any stream mining algorithm. Our DTM algorithm developed in Section 4.3 uses similar probabilistic reasoning. It still has three main features distinguishing it from FDPM. In particular, DTM estimates the frequency for the uncounted period, it can be queried after any number of transactions without losing its bound, and it applies different probabilistic reasoning.

Stream Mining

The STREAM MINING algorithm by Jin and Agrawal (2005) is another algorithm with an additional error parameter ϵ for some $0<\epsilon\leq 1$. Based on ϵ, it provides the following guarantees: (i) The algorithm returns all frequent itemsets and (ii) all itemsets returned by the algorithm have a frequency of at least $\theta\cdot(1-\epsilon)$. The algorithm processes 1- and 2-itemsets immediately and buffers transactions to process larger itemsets later in batches. The buffer has a variable size contrary to many other algorithms that use a fixed buffer size. Processing of larger itemsets is delayed until a certain condition on

the average number of two-itemsets per transaction holds. This condition depends both on θ and ϵ. The algorithm then reduces the count of each 2-itemset and removes all 2-itemsets with count 0 before it generates larger itemsets from the buffer in a level-wise manner using the Apriori property. After each level has been processed, all counters of this level are reduced by 1 and all itemsets with counter 0 are removed. Conceptually, the algorithm runs on an itemset lattice structure. At the end, all itemsets with counter at least $\theta t - r$ will be output, where t is the stream length in transactions and r the number of reduction steps until t.

`Discussion` The pruning of supersets by means of the reduction step is very weak compared to the standard pruning approach. As a result, a very large number of infrequent sets is enumerated by this algorithm, which is feasible only for streams with short transactions. The experimental evaluation of the algorithm focuses on data sets in which the average size of frequent itemsets is only 4 or 6. The correlation between the number of two-itemsets and 4- or 6-itemsets is higher than the correlations with the number of larger k-itemsets. It is an open question, how accurate the approach performs for larger k as this correlation gets weaker.

SApriori

The SApriori algorithm by Sun et al. (2006) adapts the Apriori algorithm to data streams. It has an error and a confidence parameter ϵ and δ, respectively. The stream is divided by the algorithm into blocks of size $2\theta \log(2/\delta)/\epsilon^2$. Each block is used for mining frequent itemsets of some particular length. To find the frequent k-itemsets, it needs to process k blocks. Once the largest frequent itemsets have been found, all remaining blocks can be ignored by the algorithm. Alternatively, the algorithm could be restarted to adapt to potential changes in the data distribution.

`Discussion` While, on the one hand, the algorithm is extremely fast, on the other hand, it requires the largest transaction buffer size and must process several buffers in order to produce reasonable F-scores. In contrast to all other algorithms which can adapt at least to some extent to changes in the data stream, this algorithm assumes that the distribution in the stream is static. With such a strong assumption, it is an open question, why the algorithm does not mine itemsets of all lengths from the first block and ignores all remaining blocks.

EStream

EStream(Dang et al., 2008) is a complete, but not sound mining algorithm that has an error and a maximum pattern length parameter $0 < \epsilon < \theta/2$ and L, respectively. It provides the following guarantees: (i) The frequency of all 1-itemsets is correct, (ii) the error in the estimated frequency of any 2-itemset is at most ϵ, and (iii) the error in the estimated frequency of all other itemsets is at most 2ϵ.

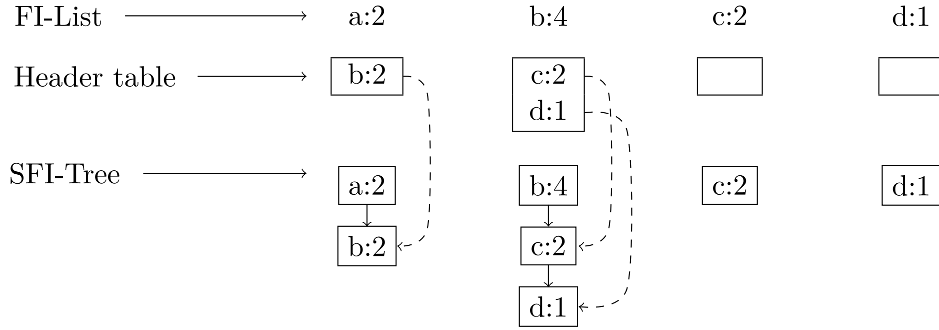


Figure 3.2.: Data structures used by the DSM-FI algorithm for the transactions from Figure 3.1.

The algorithm stores frequent itemsets in a trie structure and does not use any transaction buffer. Each transaction is immediately processed and then discarded. For each itemset length k, EStream specifies a decreasing minimum frequency threshold as a function of k and L, and stores a k-itemset as a candidate if all of its $(k-1)$-subsets reach the minimum threshold calculated for $k-1$. Whenever $\lceil 2^{L-2}\epsilon^{-1} \rceil$ transactions have been processed, the algorithm prunes all infrequent, but the 1-itemsets.

Discussion Since it uses no buffer, EStream stores many additional itemsets, which are in fact infrequent. The idea that smaller subsets need to occur more frequently than larger itemsets is intended to reduce the set of infrequent itemsets stored. However, when an itemset first occurs in a transaction, it must be added as a candidate if the subsets satisfy the frequency constraint, even if it will be removed in the next pruning step. Furthermore, both the output and the frequency thresholds for subsets depend on the specification of L. The bound on the error of the estimated frequency holds only for the correct L. This parameter is hard to set for a user in advance. For too small or too large values of L, the algorithm might miss frequent itemsets. While the idea to process transactions immediately without a buffer is very charming, there is a clear cost associated with the frequent updates of such a strategy. The benefits of buffering transactions for a while and processing them in mini-batches seems to be worth the additional delay and memory costs in most cases and has a clear runtime benefit.

DSM-FI

DSM-FI is proposed by Li et al. (2008a). Similarly to Lossy Counting, it has an additional error parameter $0 < \epsilon < \theta$ and provides the same error guarantee as Lossy Counting. This algorithm processes transactions in mini-batches. It uses a forest as its central summary structure. The forest contains a prefix tree for each sub-frequent item. Each tree has a classical header table like an FP-Tree, with pointers to nodes in the tree. Nodes in the tree for the same item are linked in a list in the usual way. The structure is illustrated in Figure 3.2.

Transactions within a batch are processed individually. For each transaction, the items are sorted in lexicographic order. The transaction itself and all suffixes of the transaction are inserted into the forest summary structure. After all transactions from one mini-batch have been processed, a pruning step removes all infrequent items from all trees. Trees with infrequent items at the root node are removed entirely.
The set of frequent items is produced from the forest only upon request. For each tree, the algorithm tests the maximal itemset encoded in the tree first. If this set is frequent, all subsets are frequent too. Otherwise, it is enumerated into m subsets, each of size $m-1$. The strategy is applied recursively until a frequent set is found, which stops the recursion. This enumeration strategy pays only off when most of the frequent itemsets are of size at least $m/2$. To see this, consider a frequent itemset lattice. Starting from the largest element, the number of enumerated sets is smaller if most elements in the lattice are of size greater than half of the size of the maximal element.

`Discussion` The algorithm does not take full advantage of the fact that it processes transactions in mini-batches. It first processes each transaction individually and only identifies infrequent items after an entire mini-batch has been processed and each transaction has been projected and inserted into the forest structure. Lossy Counting first scans all transactions in the buffer to identify infrequent items to discard. The DSM-FI algorithm would profit from a scan of the transactions in the mini-batch to update the frequency statistics of single items, as it could then ignore all infrequent items.
Another issue originates from the enumeration scheme when producing a result. Consider a data stream containing the four simple transactions `ab, ac, ad, ae`. Assume they are all frequent. Then the DSM-FI algorithm will first construct the five-itemset `abcde`, all 5 four-itemsets, all 10 three-itemsets and finally reports the four frequent two itemsets. It would have thus been better to enumerate itemsets bottom-up in this example. In many real-world scenarios, the situation is similar; most transactions contain only a very small subset of all items. The enumeration strategy thus needs to generate huge amounts of infrequent itemsets until it finally finds the frequent ones. This can be fixed by changing the enumeration order of the tree structure. Starting with small itemsets and producing larger ones from the tree would result in a faster algorithm for many data sets.

hMiner

The hMiner algorithm (Wang and Chen, 2009) extends the idea of the hCount algorithm (Jin et al., 2003). Both algorithms rely on hashing. While hCount mines frequent items, hMiner mines frequent itemsets. It provides a guarantee for the case that the items are independent.[1] If, however, the items were truly independent, then the mining problem would be trivial. Indeed, in this case, it suffices to count the frequencies of single items and compute the frequencies of larger itemsets using basic results

[1] The central proof of the algorithm assumes that items are independent.

from probability theory. Itemset mining is interesting (both from the application and algorithmic aspects) only if items are *not* independent. For this non-trivial case, the algorithm does not provide a guarantee.

The algorithm works as follows. It hashes all subsets of each transaction with a single hash function. The hash function has the standard form $h(x) = (a \cdot x + b) \mod m$, where a and b are arbitrary prime numbers and m is the size of the hash table. The size m is determined based on user-defined error and confidence parameters. Each entry in the table records the number of accesses, the last access time, and points to the frequent itemsets hashed to that position. For the frequent itemsets, estimated and exact counts are maintained. The estimated counts account for the past when the itemset was not counted. The exact ones maintain the exact count from when counting of the itemset starts up to the current transaction. The support count is derived from the number of accesses of the hash entry and the counts of all frequent items for that entry. New itemsets are added to the hash structure if the itemset is a 1-itemset or all subsets are frequent. Pruning does not check the entire hash table but controls only those entries that have been accessed while adding a transaction. It removes all itemsets from such accessed hash entries that are no longer frequent.

`Discussion` This algorithm enumerates every subset of each transaction. Such an enumeration is only feasible if the transactions are small since a transaction of size n has $2^n - 2$ proper non-empty subsets. The experimental evaluation of the authors considers artificial streams with only up to 7 items per transaction. The mechanism to estimate support counts assumes that all infrequent items hashed to one position have the same probability to occur in any transaction. This is very unlikely for at least two reasons: First, in most data sets both real-world and simulated, items have different probabilities (cf. Figure 5.17, page 107). Second, itemsets of different sizes can be mapped to the same hash bucket. In general, larger itemsets are less frequent than their subsets. The central error analysis of hMiner assumes that the items in a transaction are independent. If, however, the items were independent of each other, there is no need for frequent itemset mining, as the likelihood of each itemset depends in this case purely on its size and not on the items involved.

SA-Miner

The Support-Approximation miner (SA-Miner) algorithm (Li and Jea, 2014) is an algorithm without any formal error guarantee. It uses feed-forward neural networks to learn a function to predict the frequency of a k-itemset based on the sum of the frequencies of its subsets. More precisely, given $(x_1, \ldots, x_m)$ with $m \leq k - 1$ and y, where the x_is are the sum of the support counts of all i-subsets of the k-itemset and y is the support count of the k-itemset, the algorithm learns a function $f : \mathbb{N}^m \to \mathbb{N}$ to predict the support count of a k-itemset given the support counts of its subsets up to size m. While the support count of the subsets up to size m must be explicitly

maintained, the support count of any larger itemset is approximated given the learned function. Hence, the name *support approximation* of the method. This eliminates the need to explicitly maintain the support counts of larger sets.

The algorithm processes transactions in mini-batches. The first batch is used to learn the prediction function f. In subsequent batches, only *small* itemsets, i.e., those up to size m, are counted and support counts of larger itemsets are approximated by the learned function. The value *small* is determined by the square root of the average transaction size. Any itemset smaller than this value is considered to be *small*, all others to be large. The functional dependencies are learned with feed-forward neural networks. Rather than just using a simple two-layer feed-forward architecture, genetic algorithms (Fraser and Burnell, 1970) are employed to generate a pool of neural network architectures and weights. This pool is updated and iteratively reduced until a single network remains, which is then trained to convergence with back-propagation (Rumelhart et al., 1986), a standard algorithm for such training.

Once the network has been trained, mini-batches are processed continuously to update the exact counts of all *small* itemsets. The algorithm tests for concept drifts by either checking the average transaction length or comparing the singleton distributions of the mini-batches. The description mentions both options, without further specifying which one is used by the algorithm. If a drift is detected, the support relationships need to be relearned, i.e., a new neural network must be trained form the current batch. The algorithm can work in both the landmark and the sliding window model. In the landmark model, counts can only increase; in the sliding window model, the support counts of small sets are reduced for the batch leaving the window. The support approximation technique is only employed if there is a mining request from a user. It is then used to predict the frequencies of the non-counted itemsets.

`Discussion` The algorithm tries to combine techniques popular in other areas of computer science such as genetic algorithms and neural networks with frequent itemset mining. The idea to approximate the count of large itemsets as a function of the support count of their subsets attempts to exploit the fact that the support count of a superset depends on the support count of its subsets. How well this approach works for large itemsets or long data streams is not known and needs further investigation, especially with regards to some error bound. It is a well-known fact that neural network training requires a lot of data and many training epochs for good fits. Simple linear regression might be faster during training and provide similar results in terms of the approximation quality. The merits of this algorithm are that it explicitly detects concept drifts and adapts to them.

3.3. Sliding Window Algorithms

In this section, we consider the four most prominent algorithms mining frequent itemsets under the sliding window model.

SWIM

The Sliding Window Incremental Miner (SWIM) by Mozafari et al. (2008) produces an exact output with some optional maximal delay d. Conceptually a window is divided into several panes of identical size. The delay parameter d controls the maximal number of panes processed before reporting a frequent itemset. For each arriving pane, there is one leaving the window. The delay is bounded by the number of panes that constitute the entire sliding window.

For a new arriving pane, the algorithm performs two steps. First, it updates the support counts of all frequent itemsets. Then it searches new frequent patterns in the pane. For each new frequent pattern, it creates an entry in the tree storing the frequent itemsets. In addition to the current support count, it maintains an array of support counts for all panes in the window. These counts will be updated over time as old panes move out of the window. The support count is exact after the last pane has left the window in which the itemset was infrequent. The additional array is then discarded. The itemset is reported after this delay as frequent for each window, in which it was frequent. If the user does not want to wait, the algorithm counts the frequency of new itemsets in all panes, which will produce the exact result immediately.

`Discussion` The idea to delay the report of frequent itemsets is attractive. It reduces the computation time at the expense of a reporting delay. For applications that are not time-critical the additional delay may be acceptable. It allows processing more transactions in a given time. This is of interest whenever an application produces data streams at a high rate. The algorithm also has the flexibility to produce exact results immediately.

WSW

The weighted sliding window (WSW) algorithm (Tsai, 2009) considers sliding windows, where the user can assign different weights to each slide of a window. This model is more general than the time fading model, as the weights can be set more flexible. It is an open question, whether the results will be intuitive for freely defined weights. The algorithm considers windows based on time intervals instead of transactions. That is, each slide can contain a different number of transactions.

WSW stores the transaction identifiers for each item. It computes the frequent itemsets level-wise utilizing the Apriori property by computing the intersection of the sets of transaction identifiers. For each window, the support set of each item is computed only once, but each time the window is moved forward, the support set is multiplied with the new weight and the candidate itemsets are generated again. This candidate generation step is one of the bottlenecks of Apriori-based approaches. In addition, the support sets of larger itemsets are computed via a set intersection, which is a costly operation.

`Discussion` Both the idea of individual weights for each slide and slides based on time instead on the number of transactions provide more flexibility than many other algorithms. Despite this flexibility, the algorithm is, however, very simple and re-executes Apriori for updated data sets. A large number of empirical results indicate that Apriori is slow compared to newer algorithms. The general idea of WSW does not depend on Apriori but could use another algorithm to run faster. Even then a lot of computation would be repeated. Streaming algorithms should exploit the fact that they have already computed a partial result for an earlier state of the data stream to reduce the time per update. The WSW algorithm does not take advantage of such a strategy.

MFI-TransSW

The Mining Frequent Itemsets with a Transaction-sensitive Sliding Window (MFI-TransSW) algorithm (Li et al., 2006) maintains a bit-vector for each item. The bit-vector of item x takes the value `true` at position t if x occurs in transaction T_t. When a new transaction arrives, all bits are left-shifted and the new transaction is encoded in the last bits. The window is moved continuously with each arriving transaction. Bit vectors that are no longer needed (i.e., when all bits are set to `false`) are removed. The set of frequent itemsets is computed upon request.
Frequent itemsets are generated level-wise. The support set is obtained by computing the component-wise logical `AND` between bit-vectors. The Apriori property is used to prune unpromising candidates.

`Discussion` The algorithm relies on a very simple idea. It recomputes the set of frequent itemsets for each mining request from the bit-vectors and ignores all previous results. This is feasible if the mining requests are rare. If, however, the user queries the set of frequent itemsets regularly, the strategy is way more expensive than maintaining the set of frequent itemsets and updating them with new transactions. The bit-vector data structure is suitable for dense data streams but fits less well to sparse data streams.

CPS-Tree

Tanbeer et al. (2009) describe a dynamic reorganizing tree structure that is mined with the FP-growth algorithm. This approach is called Compact-Pattern-Stream-Tree Sliding Window (CPS-TreeSW). Its idea is to calculate the support descending order on the items and reorganize the tree at regular intervals. Consider a window which is divided into several panes, each of identical length. The size of a pane defines the interval after which the tree is restructured. Nodes at the leaf level maintain separate support counts for each pane. All other nodes maintain a single count. Each node has pointers to its parent, a sibling, and its children. A global item list maintains the global support count of each item. The list contains a pointer to the first node for the item in the tree. When the window is moved one pane forward in the data stream, the counts from the oldest pane can be easily subtracted from any node because they are known at the leaf level. New transactions are added to the existing tree based on the *old* support

descending order of the items. After all transactions of the new pane have been inserted into the tree, the new global order is derived based on the updated item list and the tree is restructured with respect to the support descending order. The set of frequent itemsets can be mined from the updated tree with the FP-GROWTH algorithm.

`Discussion` Maintaining separate support counts at the leaves for each pane eliminates the need to recompute the support count for the transactions to be deleted. This is a clear advantage for any algorithm working with mini-batches. Keeping this information only at the leaf level, where it is truly needed, reduces the overall cost. The restructuring of the tree comes at some cost. While it keeps the tree small, it may not pay off in terms of overall runtime if the tree changed only a little. A measure on the deviation of the tree from the perfect order could be added to the CPS-Tree to optimize the trade-off between restructuring and mining time. The more compact the tree, the faster FP-GROWTH will run on the tree. The recomputation of the entire output set from scratch after each mini-batch has a higher cost compared to algorithms that update the set of frequent patterns directly. On the other hand, the CPS-TREESW algorithm has a very small memory footprint, as it needs to maintain only the tree which is bounded by the number of transactions in the window, whereas the number of frequent itemsets is exponential in the size of the number of distinct items.

3.4. Time Fading Window Algorithms

This section contains one single algorithm for the time fading model.

estDec

The ESTDEC algorithm (Chang and Lee, 2003) is an approximate algorithm without a formal error analysis. It maintains the exact support count of all singleton items and a subset of all larger itemsets. Given two user-defined parameters *decay base* $b > 1$ and *decay base life* $h \geq 1$, it derives the *decay rate* $d = b^{(1/h)}$ to discount the weight of older transactions. Let n denote the total number of transactions in the stream discounted by the decay rate.

For each new arriving transaction T^+, the algorithm first updates the number of transactions in the stream with the user-defined decay rate d. It then updates the support count of all monitored itemsets respecting d. Itemsets which become less frequent than a pruning threshold $\theta_{pr} < \theta$ are removed from the monitored set. Each new item in T^+ that is not monitored is added to the prefix-tree structure used to store the itemsets. Single items are never pruned. Larger sets are added if their maximal estimated support count divided by n is larger than some insertion threshold $\theta_{ins} < \theta$. The maximal estimated support count of a k-itemset is simply the minimal support count of all its $(k-1)$-subsets. The minimal estimated support count of a k-itemset is derived from the union bound of its $(k-1)$-subsets. The error in the support count is defined as the difference between the maximal and minimal estimated support count. Upon request, ESTDEC retrieves all itemsets from the tree that have a support larger than $\theta \cdot n$.

`Discussion` The result of the algorithm depends largely on the thresholds θ_{ins} and θ_{pr} used for insertion and pruning, respectively. Most other algorithms use θ as insertion threshold and do not have a specific insertion threshold θ_{ins}. The pruning threshold θ_{pr} is often derived from some error bound. estDec provides more flexibility with its explicit thresholds, but no bound. The explicit computation of the updated result for each individual transaction reduces the throughput of the algorithm compared to algorithms that update mini-batches.

3.5. Discussion

The key characteristics of the algorithms mining frequent itemsets from data streams are summarized in Table 3.1. In addition to the algorithms described above, the table lists the characteristics of the following three less cited algorithms: AP_Stream (Silvestri and Orlando, 2007), DSCA (Jea and Li, 2009), and BFI Stream (Li et al., 2008b).
The table considers the dimensions *Problem definition*, which comprises, the pattern type, the error in the output and the false result, *Pattern maintenance*, composed of the update interval and the enumeration approach, and *Data handling*, which considers the dynamics of the threshold, the synopsis structure and the use of bit-vectors.
By definition, the table includes only algorithms mining frequent patterns. The most popular window model for frequent itemset mining from transactional data streams is the landmark model, followed by the sliding window model. The time fading model is the least prominent. For the landmark model, most algorithms produce false-positive results. A second broad category includes those algorithms for which the output error is not specified, i.e., the algorithm can produce both false-positive and false-negative results at the same time. The false negative approach is exotic, despite its benefits. All Algorithms for the sliding window model produce exact results. We observe a strong correlation between the output and the false result. More precisely, three of the algorithms with false positive output limit false patterns to the interval $\theta - \epsilon \leq freq(X) < \theta$ and three use looser bounds. Algorithms with false positive and false negative results do not have a specific proven error bound, they come with undefined false results. This is equivalent to stating that the false result is in the interval $0 < freq(X) < \infty$. The algorithms with exact results produce no false results. Concerning the update strategies, mini-batching and updates per transaction are on par. The favorite listing strategy is bottom-up enumeration. Given the downward closure property, this is very reasonable. All algorithms, except for one, use static thresholds. Regarding the data structures, trees seem to be the most prominent choice. Bit vectors are used by only one algorithm on our list.
To assist in the selection of an algorithm, we present a decision tree in Figure 3.3 for the choice based on the window model, the output, and the update strategy.
While most designs have their merits and limitations, some algorithms seem to work only for special cases. These are hMiner, Stream Mining, DSM-FI, and SA-Miner. The first two were evaluated only on data sets with very short average transaction sizes of length up to 4. Their general design is not suited for large transactions, as they

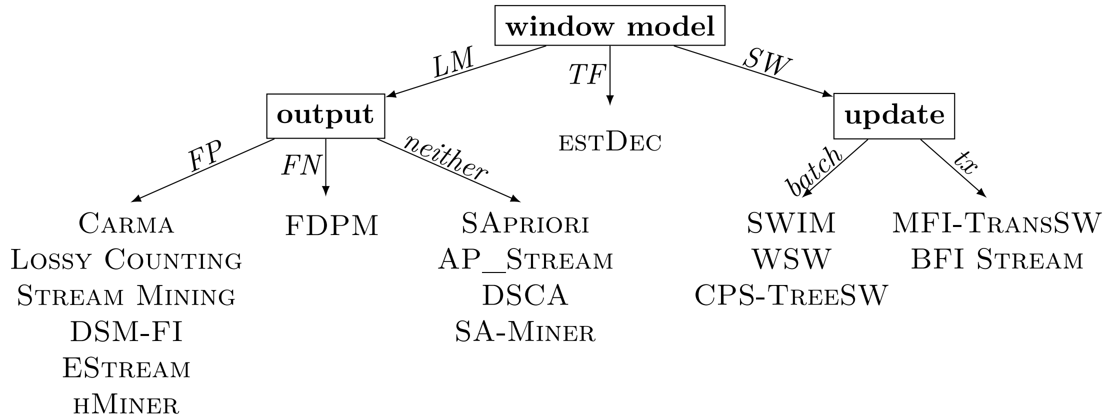


Figure 3.3.: Decision tree for the selection of an algorithms mining frequent itemsets from data streams. (LM = landmark, SW = sliding window, TF = timefading, tx = transaction)

both enumerate every subset of each transaction (hMiner) or almost all subsets of a set of transactions (Stream Mining). Both approaches are infeasible for large transactions. DSM-FI is very slow because of its top-down enumeration approach requiring it to check a very large number of non-frequent itemsets. SA-Miners design is overly complicated without any proof that the proposed method works for data streams with large transactions.

3.6. Summary

In this chapter, we have presented a survey of the literature on frequent itemset mining from transactional data streams. For each covered algorithm, we discussed its idea as well as the advantages and potential limitations of the design. To support the selection of an algorithm with particular properties for a given setting, we compiled a table with the main characteristics and a decision tree to assist the choice. In summary, our (short) survey supports the understanding of the sate-of-the-art and reveals that the majority of algorithms are designed for the landmark model.

	Problem definition			Pattern maintenance		Data handling			
Algorithm	**WM**[1]	**Output**[2]	**False result**	**Update**	**Enumeration**	**Threshold**	**Synopsis structure**	**BV**[3]	**Reference(s)**
Carma	LM	FP	$0 < freq(X) < \theta$	transaction	bottom-up	static	lattice	no	Hidber (1999)
Lossy Counting	LM	FP	$\theta - \epsilon \leq freq(X) < \theta$	mini-batch	bottom-up	static	tree/array	no	Manku and Motwani (2002)
Stream Mining	LM	FP	$\theta - \epsilon \leq freq(X) < \theta$	transaction	bottom-up	static	lattice	no	Jin and Agrawal (2005)
DSM-FI	LM	FP	$\theta - \epsilon \leq freq(X) < \theta$	transaction	top-down	static	tree	no	Li et al. (2008a)
EStream	LM	FP	$\theta - 2\epsilon < freq(X) < \theta$	transaction	bottom-up	static	tree	no	Dang et al. (2008)
hMiner	LM	FP	$0 < freq(X) < \theta$	transaction	undefined	static	hash table	no	Wang and Chen (2009)
SApriori	LM	FP & FN	unspecified	mini-batch	bottom-up	static	table	no	Sun et al. (2006)
AP_Stream	LM	FP & FN	unspecified	mini-batch	bottom-up	static	table	no	Silvestri and Orlando (2007)
DSCA	LM	FP & FN	unspecified	transaction	bottom-up	static	tree	no	Jea and Li (2009)
SA-Miner	LM	FP & FN	unspecified	mini-batch	bottom-up	static	tree & ANN[4]	no	Li and Jea (2014)
FDPM	LM	FN	$Pr(0) \geq 1 - \delta$	mini-batch	bottom-up	dynamic	table	no	Yu et al. (2004, 2006)
MFI-TransSW	SW	exact	none	transaction	bottom-up	static	bit-vector	yes	Li and Lee (2009); Li et al. (2006)
BFI Stream	SW	exact	none	transaction	bottom-up	static	tree	no	Li et al. (2008b)
SWIM	SW	exact	none	mini-batch	bottom-up	static	tree	no	Mozafari et al. (2008)
WSW	SW	exact	none	mini-batch	bottom-up	static	tid-sets	no	Tsai (2009)
CPS-TreeSW	SW	exact	none	mini-batch	bottom-up	static	tree	no	Tanbeer et al. (2009)
estDec	TF	FP	$0 < freq(X) < \theta$	transaction	bottom-up	static	lattice	no	Chang and Lee (2003)

Table 3.1.: Most important characteristics of algorithms mining frequent itemsets from data streams.

[1] Window Model: LM = landmark, SW = sliding window, TF = timefading

[2] Output: FP = false psoitive, FN = false negative

[3] BV = Bit-vector

[4] ANN = artificial neural network

4. Frequent Itemset Mining from Transactional Data Streams

This chapter considers the problem of mining frequent itemsets from data streams under the landmark model with probabilistic guarantees for the output patterns that are independent of the update cycle. We start with a high-level description of our contribution in Section 4.1. In the main sections, we present two algorithms. The first one, called PARTIAL COUNTING is the subject of Section 4.2. It is based on a simple probabilistic reasoning. The second, more sophisticated algorithm, called DTM, is presented in Section 4.3. It has a dynamic probabilistic error bound that allows it to produce results after *any* number of transactions. We extensively evaluate our algorithms empirically and compare them to the state-of-the-art in Section 4.4. Our results demonstrate that the DTM algorithm outperforms the state-of-the-art algorithms in terms of F-score. We discuss our contribution in Section 4.5 and summarize this chapter in Section 4.6.

4.1. Contribution

In this section, we provide a short overview of the contributions in this chapter, before presenting them in detail. Recall from Chapter 2 that mining data streams under the landmark model is challenging because, in general, it is infeasible to keep the entire stream in main memory. Thus a small synopsis is required. It must store the essential information from the data stream to approximately capture both the past and the present information. Clearly, any algorithm with a guarantee for the analysis of such a data stream must guarantee that the synopsis is suitable to infer information about the past and the current state of the stream. Some of the algorithms for this task, such as SA-MINER by Li and Jea (2014), come without a formal analysis of the output's error. Others, with a formal analysis of the error, provide awkward error bounds (e.g., the DSM-FI algorithm by Li et al. (2008a)) or even worse contain flaws (e.g., hMINER by Wang and Chen (2009) (cf. Section 3.2)). State-of-the-art algorithms with proven error bounds require fixed-size updates. They either process each transaction individually, which is slow, or fixed-size mini-batches. While the mini-batch-based update is fast, it limits flexibility. In particular, if such an algorithm is queried for an incomplete mini-batch, then it cannot guarantee any clear error bound. We decouple the (probabilistic) guarantee from the update interval to obtain a fast and flexible algorithm.
In particular, we first propose the PARTIAL COUNTING algorithm inspired by CARMA (Hidber, 1999). Our algorithm combines explicit counts of itemsets with probabilistic reasoning to estimate the support count of an itemset for the past, while it was not counted.

The estimation relies on Bayes' theorem and the observed itemset frequencies. A distinct feature of the algorithm is that the estimate improves with additional transactions in the data stream. In fact, it converges in the limit.
As a second scenario, we then take a different probabilistic view and consider transactions generated by *dynamic* distributions. In this setting, the transactions are generated independently at random with some unknown distribution over all transactions. The occurrence of an item X in the i-th transaction T_i gives rise to the binary variable X_i indicating whether or not $X \in T_i$. In fact, the X_is are independent Poisson trials. For this setting, we designed the algorithms DCIM[1] (Trabold and Horváth, 2016) and DTM[2]. Both algorithms rely on the same theory and share the concept of dynamic bounds. The algorithms guarantee that the estimate of the frequency θ is correct within a *dynamic* error of ϵ, with a probability of at least $1-\delta$ for some user-defined confidence $\delta > 0$. This bound is formally proven and improves as more and more transactions are received from the stream. That is, ϵ gets gradually smaller with additional transactions. DTM improves upon DCIM in terms of F-score. In this thesis, we only cover the DTM algorithm because the two algorithms are very similar and DTM obtains superior F-scores. This algorithm estimates the frequency of a new (potential) frequent itemset for the unobserved past not by the frequencies of its subsets, but by a dynamic bound. The bound is always below the frequency threshold and converges towards it for infinite streams.
We extensively compare our algorithms Partial Counting and DTM to five state-of-the-art algorithms. Our experiments demonstrate the good mining quality of our algorithms and show that they are competitive in both runtime and memory, given the quality of the mining results. DTM outperforms all state-of-the-art algorithms in terms of F-score over a large variety of data streams.

4.2. The Partial Counting Algorithm

This section introduces the Partial Counting algorithm approximating the frequency of frequent itemsets from a data stream of transactions at any point in time. Our goal is to design an algorithm that produces a sequence $\hat{\mathcal{F}}_{1,\theta}, \hat{\mathcal{F}}_{2,\theta}, \ldots$ of families of candidate frequent itemsets such that at each point in time $t \in \mathbb{N}$, the family $\hat{\mathcal{F}}_{t,\theta}$ approximates $\mathcal{F}_{t,\theta}$ closely, while maintaining a small memory footprint. Before we present the algorithm, we introduce two further notations. We define the support count of an itemset X in a data stream $\mathcal{S} = \langle T_1, T_2, \ldots, \rangle$ from time i to time j by

$$C_X^{i,j} = |\{k \in \mathbb{N} : i \leq k \leq j, X \subseteq T_k\}| \ ,$$

i.e., the support count of X for the interval is equal to the number of transactions from T_i to T_j that contain X. Let $p_i(X) = \mathbf{Pr}[X \subseteq T_i]$. We write $p(X)^{i,j}$ to denote the average success probability of the event that itemset X is being supported by the transactions

[1] Dynamic Confidence Intervals Miner
[2] Dynamic Threshold Miner

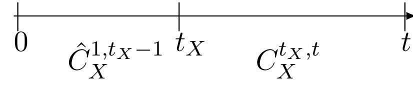


Figure 4.1.: **Partial Counting working principle:** The occurrences of itemset X are counted from t_X. $C_X^{t_X,t}$ denotes this observed count for X from t_X to t. As the support count of X is not available for the period 1 to $t_X - 1$, it must be estimated for this time interval; $\hat{C}_X^{1,t_X-1}$ denotes this estimated count.

from T_i to T_j, i.e.,

$$p(X)^{i,j} = \frac{p_i(X) + \ldots + p_j(X)}{j - i + 1} \quad .$$

The inputs to our Partial Counting algorithm are a minimum frequency threshold $\theta \in (0, 1]$ and, for each time point $t \in \mathbb{N}$, the most recent transaction $T_t \subseteq I$. For each $t \in \mathbb{N}$, the algorithm estimates the frequency of all frequent itemsets with respect to the transaction data stream $\langle T_1, \ldots, T_t \rangle$, in one pass and without storing the transactions $T_1, \ldots, T_{t-1}$ seen earlier.

For all $t \in \mathbb{N}$, the algorithm keeps only those itemsets in the memory that have been estimated as frequent after having processed T_t; all other itemsets, except for the 1-itemsets, are removed from the memory. Thus, in contrast to the Lossy Counting algorithm by Manku and Motwani (2002), the space complexity of the algorithm is only $O\left(|\hat{\mathcal{F}}_{t,\theta}|\right)$, where $\hat{\mathcal{F}}_{t,\theta}$ is the family of itemsets estimated as frequent with respect to $T_1, \ldots, T_t$.

The algorithm is depicted in Algorithm 2. To process the next transaction T_t arriving at time t, the algorithm takes in a first step (lines 4–12) all non-empty subsets X of T_t (line 4) and increments the counter for X if X is already in the memory (lines 5–6). Otherwise, if X is a singleton or it is a k-itemset for some $k > 1$ and all of its $(k-1)$-subsets are already in the memory, it stores X with some additional auxiliary information. It utilizes the Apriori property, i.e., that a k-itemset cannot be frequent if at least one of its $(k-1)$-subsets is infrequent. More precisely, for X we store a quadruple $(x.set, x.s, x.t, x.count)$ that will be used to estimate the frequency of X for the data stream $T_1, \ldots, T_{t-1}$.

The components of the quadruple are defined as follows: Variable $x.set$ stores the itemset X itself. All subsets with exactly one item less (i.e., Y such that $|Y| = |X| - 1$) along with their count are stored in the set variable $x.s$. The time t_X, i.e., when the counting of X starts, is kept in $x.t$. Finally $x.count$ stores the count of the itemset. It is important to note that the subsets of T_t are processed in increasing cardinalities, as otherwise potential new frequent itemsets can be lost (see lines 4,8, and 9).

In a second step (lines 13–16), the algorithm then prunes all quadruples corresponding to itemsets X from the memory that satisfy $|X| > 1$ and are estimated as infrequent at point t. In line 14, the algorithm first calculates an estimation of the support count of X; different strategies for this estimation step will be presented in Section 4.2.1 by noting that all these strategies recursively estimate the support count from the counts (i.e., $x.count$) maintained by the algorithm using some statistical inference. Figure 4.1

Algorithm 2 Partial Counting

```
1: Initalization
2: F ← ∅                      // current set of frequent patterns with auxiliary information
3: Processing of transaction T_t
4: for X ⊆ T_t in increasing cardinality do                                  // counting
5:     if ∃x ∈ F with x.set = X then
6:         increment x.count
7:     else
8:         if |X| = 1 ∨ (∀Y ⊂ X : ∃y ∈ F with y.set = Y ∧ |Y| = |X| − 1) then
9:             x.s ← {(y.set, y.count) : y ∈ F ∧ y.set ⊂ x.set ∧ |y.set| = |x.set| − 1)}
10:            x.t = t
11:            x.count = 1
12:            F ← F ∪ {x}
13: for x ∈ F with |x.set| > 1 do                                              // pruning
14:     compute Ĉ_x^{1,t}                                                // see Section 4.2.1
15:     if Ĉ_x^{1,t}/t < θ then
16:         delete x from F
17: Output after timepoint t
18: for x ∈ F do
19:     if |x.set| > 1 ∨ x.count/t > θ then output (x.set,Ĉ_x^{1,t})
```

illustrates the general setting: For an itemset X (re)counted from time t_X, its support count for the period from 1 to $t_X - 1$ must be estimated from the information available at time t. If the frequency derived from this estimation is below the threshold θ, X is removed from the memory. Thus, when an itemset becomes frequent it is stored and counted as long as it is estimated as frequent. When it becomes infrequent, it is immediately pruned.

For a query after time point t, Partial Counting outputs all itemsets with their estimated support counts from $\mathcal{F}$ that meet the minimum frequency condition (line 18–19). According to the construction, all itemsets X in $\mathcal{F}$ with $|X| > 1$ will automatically be part of the output. In summary, we have a true online algorithm that returns a family $\hat{\mathcal{F}}_{t,\theta}$ of itemsets predicted as frequent from the stream of transactions from the beginning of the stream up to time t.

4.2.1. Support Approximation Strategies

This section describes a generic framework for support count estimation and presents different strategies for this problem. Except for one, all of the strategies in this section are based on some careful combination of the observed counts with the estimated ones that are derived from conditional probabilities.

We write $\hat{C}$ for estimated support counts. As illustrated in Figure 4.1, whenever there is some observed support count for an itemset X, the estimated support count for the entire period of all transactions is given by the estimation $\hat{C}_X^{1,t_X-1}$ of the support count for the period $[1, t_X - 1]$ for which the support count of X is not available plus the *observed* support count $C_X^{t_X,t}$ for the interval $[t_X, t]$ in which X has been counted. As long as no observed count for X exists at time t, its estimated support count is 0, i.e.,

$$\hat{C}_X^{1,t} = \begin{cases} \min\left(\lfloor\theta(t_X - 1)\rfloor, \hat{C}_X^{1,t_X-1}\right) + C_X^{t_X,t} & \text{if } t \geq t_X \\ 0 & \text{o/w .} \end{cases}$$

We now present different strategies to compute $\hat{C}_X^{1,t_X-1}$. The first estimation strategy is similar to the one proposed by Hidber (1999). The other two are natural strategies based on conditional probabilities.

Upper Bound Estimation (ube) This estimation strategy takes the minimum estimated support count of all $(k-1)$-subitemsets Y of a k-itemset X, i.e.,

$$\hat{C}_X^{1,t_X-1} = \min_{\substack{Y \subset X, \\ |Y|=|X|-1}} \hat{C}_Y^{1,t_X-1} \quad .$$

Clearly, this formula gives an upper bound on the true support count of X. Notice that this is a static strategy in the sense that it does not improve the estimation $\hat{C}_X^{1,t_X-1}$ as further transactions arrive.

Estimation Based On Conditional Probabilities We now turn to more complex, dynamic estimation strategies. They are based on the probabilistic view of frequencies that for any itemset X and $t \in \mathbb{N}$

$$p(X)^{1,t} = p(Y)^{1,t} \cdot p(X|Y)^{1,t}$$

for any $Y \subset X$. To estimate $p(X)^{1,t}$, we need to estimate $p(X)^{1,t_X-1}$, as all information about X is available for the interval $[t_X, t]$. We estimate $p(X)^{1,t_x-1}$ by estimating (i) $p(Y)^{1,t_X-1}$ and (ii) $p(X|Y)^{1,t_X-1}$:

(i) Regarding $p(Y)^{1,t_X-1}$, we estimate it recursively by

$$p(Y)^{1,t_X-1} \approx \frac{\hat{C}_Y^{1,t_X-1}}{t_X - 1} \tag{4.1}$$

by noting that the support counts are stored from the very beginning for all 1-itemsets.

(ii) Regarding $p(X|Y)^{1,t_X-1}$, we make use of the fact that $[t_X, t]$ is a common observation period for both X and Y and the assumption that the underlying distribution $\mathcal{D}$ is stationary and estimate $p(X|Y)^{1,t_X-1}$ by

$$p(X|Y)^{1,t_X-1} \approx p(X|Y)^{t_X,t}, \tag{4.2}$$

which, in turn, can be calculated by

$$p(X|Y)^{t_X,t} = \frac{C_X^{t_X,t}}{C_Y^{t_X,t}} \ . \tag{4.3}$$

One can show that for sufficiently large t_X and t, (4.2) gives a close estimation with high probability.

Putting together, from (4.1), (4.2), and (4.3) it follows that $p(X)^{1,t_X-1}$ can be estimated by

$$p(X)^{1,t_X-1} \approx \frac{\hat{C}_Y^{1,t_X-1}}{t_X - 1} \cdot \frac{C_X^{t_X,t}}{C_Y^{t_X,t}} \ . \tag{4.4}$$

As the frequency of X in $[1, t_X - 1]$ is identical to the probability $p(X)^{1,t_X-1}$, by (4.4) the support count $\hat{C}_X^{1,t_X-1}$ can be estimated by

$$\begin{aligned} \hat{C}_X^{1,t_X-1} &= p(X)^{1,t_X-1} \cdot (t_X - 1) \\ &\approx \frac{\hat{C}_Y^{1,t_X-1} \cdot C_X^{t_X,t}}{C_Y^{t_X,t}} \ . \end{aligned}$$

The two strategies presented below build upon the idea discussed above. They differ from each other only by the particular choice of Y. All strategies have in common that they estimate the support counts only for itemsets X with $|X| \geq 2$.

(a) Minimum estimation (me): This strategy uses the single subset Y that results in the minimum estimated count for X, i.e.,

$$\hat{C}_X^{1,t_X-1} = \begin{cases} \min\limits_{\substack{Y \subset X, \\ |Y|=|X|-1}} \dfrac{\hat{C}_Y^{1,t_X-1} \cdot C_X^{t_X,t}}{C_Y^{t_X,t}} & \text{if } |X| > 1 \\ 0 & \text{o/w (i.e., if } |X| = 1) \ . \end{cases}$$

(b) Average estimation (ae): Averaging is a standard technique to combine several uncertain predictors for obtaining a more robust result than any individual predictor would give. This strategy averages over all Y, i.e.,

$$\hat{C}_X^{1,t_X-1} = \begin{cases} \dfrac{1}{|X|} \cdot \sum\limits_{\substack{Y \subset X, \\ |Y|=|X|-1}} \dfrac{\hat{C}_Y^{1,t_X-1} \cdot C_X^{t_X,t}}{C_Y^{t_X,t}} & \text{if } |X| > 1 \\ 0 & \text{o/w (i.e., if } |X| = 1) \ . \end{cases}$$

An Illustrative Example To illustrate the three different strategies presented above, we use the small example given in Table 4.1. It shows a total of 8 transactions in the first row, t in the second row, and the frequencies of a, b, and ab in rows three to five. The last three rows show the estimated frequencies for the three strategies.

transactions	a	b	a	ab	a	b	a	ab
t	1	2	3	4	5	6	7	8
freq(a)	1	0.5	0.66	0.75	0.8	0.6	0.71	0.75
freq(b)	0	0.5	0.33	0.5	0.4	0.5	0.43	0.5
freq(ab)	0	0	0	0.25	0.2	0.17	0.14	0.25
ube	0	0	0	0.5	0.4	0.33	0.29	0.38
me	0	0	0	0.5	0.4	0.25	0.21	0.33
ae	0	0	0	0.5	0.4	0.29	0.23	0.35

Table 4.1.: A data stream of transactions illustrating the different estimation strategies ube, me, and ae. The first row shows the transactions, the second row t, the next three rows the frequencies for itemsets a, b and ab respectively. The last three rows show the estimated frequencies for the itemsets a, b, and ab for the three strategies presented.

In the example, the first estimation occurs for transaction 4, as this is the first occurrence of ab and is counted from this transaction onwards. Up to and including the third transaction, the set a occurs twice and b once. All estimations rely on the counts of a and b for the first three transactions. The strategies start to differ from the 6th transaction (i.e., from $t = 6$) onwards. We will, therefore, illustrate the different strategies for the 6th transaction. $C_{ab}^{4,6} = 1$ for all strategies.

(i) ube estimates $\hat{C}_{ab}^{1,3} = min(2, 1) = 1$, which gives

$$\hat{C}_{ab}^{1,6} = \hat{C}_{ab}^{1,3} + C_{ab}^{4,6} = 1 + 1 = 2$$

and thus, a frequency of $\frac{2}{6} = 0.33$.

(ii) me estimates $\hat{C}_{ab}^{1,3} = min(1 \cdot \frac{1}{2}, 2 \cdot \frac{1}{2}) = 0.5$, which gives

$$\hat{C}_{ab}^{1,6} = \hat{C}_{ab}^{1,3} + C_{ab}^{4,6} = 0.5 + 1 = 1.5$$

and a frequency of $\frac{1.5}{6} = 0.25$.

(iii) ae estimates $\hat{C}_{ab}^{1,3} = \frac{1}{2}(1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{2}) = 0.75$, which gives

$$\hat{C}_{ab}^{1,6} = \hat{C}_{ab}^{1,3} + C_{ab}^{4,6} = 0.75 + 1 = 1.75$$

and a frequency of $\frac{1.75}{6} = 0.29$.

All other estimations are computed accordingly.

4.2.2. Implementation Issues

In this section, we discuss some important optimizations of the Partial Counting algorithm and briefly describe some implementation issues, including caching of intermediate results, processing of transactions in mini-batches, the data structure, insertion into the data structure, and pruning.

To speed up the algorithm, we employ two optimizations. First, observe that PARTIAL COUNTING employs a recursive strategy to estimate the unobserved frequencies $\hat{C}_X^{1,t_X-1}$ based on the frequencies of some or all subsets $Y \subset X : |Y| = |X| - 1$. Computing explicitly all steps of the recursion requires exponentially many recursive calls in the size of X. The algorithm processes itemsets in increasing cardinality (cf. line 4). Thus the estimated frequencies for all subsets $Y \subset X$ have already been computed when X is processed. The algorithm, therefore, caches all estimated frequencies in each update step. This essentially reduces the exponential number of recursive calls in the size of X to a set of lookups in the size of X, i.e., the time complexity for the update of X shrinks from exponential to *linear*.

As a second optimization, we add a simple transaction buffer to store up to B transactions for some integer $B > 0$. This strategy is used by many related algorithms (see, e.g., Manku and Motwani (2002); Sun et al. (2006); Yu et al. (2004)). While there is no mining request and the buffer is not full, transactions are simply added to the buffer one by one. Whenever the buffer is full or the algorithm is queried for the current set of frequent itemsets, all the transactions from the buffer are processed at once. The addition of a buffer changes lines 4, 10 and 11 of the algorithm. Instead of looping over a single transaction in line 4, the algorithm iterates over all subsets of the transactions stored in the buffer. Let s denote the number of transactions in the buffer. Then line 10 changes to $x.t = t - s$. In line 11, instead of setting the count of an itemset to 1, the count is set to the count of the itemset in the buffer. More precisely, let c denote the count of an itemset X in the buffer. Then line 11 changes to $x.count = c$. The buffer has the advantage that it reduces the number of iterations (cf. lines 3–16) in the algorithm and increases the throughput of the algorithm, without modifying the overall approach. Whereas other algorithms require specific sizes for their buffer, e.g., Manku and Motwani (2002); Sun et al. (2006); Yu et al. (2004), the choice of the buffer size B for PARTIAL COUNTING is arbitrary and does not affect the accuracy of the algorithm.

The data structure $\mathcal{F}$ can be stored as a prefix tree. This allows for a compact representation of the family of itemsets, as it suffices to store for each itemset X only a single item. Indeed, the itemset corresponding to a node can uniquely be recovered by concatenating the items on the path from the root to the node at hand. Furthermore, it also allows for pruning entire branches, if such a node has to be deleted that has further children.

The set $x.s$ can be stored compactly as an array of integers. For an itemset X with $|X| = k$, there are k different $(k-1)$-subsets. We sort all these subsets Y lexicographically and take only the index of the missing item i (i.e., which satisfies $Y \cup \{i\} = X$) to store the count for itemset Y. Thus, we keep k counters for the subsets and one for the itemset X.

Theoretically, we may start observing an itemset X as soon as the estimated frequencies for all subsets $Y \subset X$ have reached the minimum frequency threshold θ. X may not occur in the transaction (or buffer), when this condition is met. However, X may become frequent only when it occurs in the current transaction (buffer). That is, we

start counting X, with the transaction (buffer) containing X after all Y have already reached the minimum frequency threshold. The price we pay for this is that the first estimation of $p(X|Y)$ is 1 and as such, very inaccurate.
An itemset that is inserted into $\mathcal{F}$ at time t is never pruned in t. Otherwise, it would not have been inserted. We exploit this by skipping the pruning test for itemsets which were inserted in time t.

4.3. The Dynamic Threshold Miner

This section presents the Dynamic Threshold Miner (DTM) algorithm for approximating the solution of Problem 2 defined in Section 2.3 (page 26). While the algorithm is designed for the above problem, it also solves Problem 3 (cf. page 26) as will be shown empirically in Section 4.4. It is based on the idea of dynamical confidence thresholds, developed for the Dynamic Confidence Interval Miner DCIM algorithm (Trabold and Horváth, 2016). Though DTM builds upon the theory of DCIM, it regularly obtains superior F-scores because its estimation for the past is more accurate and it is less restrictive when starting to count new potential frequent itemsets. While DCIM distinguishes three states for an itemset DTM associates any itemset with one of two states. Both algorithms have in common that they dynamically estimates the frequency for the unobserved period. We now focus on the DTM algorithm. At any time it associates each itemset with one of the following states: *interesting* or *uninteresting*. The state of an itemset X depends on its estimated frequency $\hat{f}(X)$ and a dynamic confidence threshold ϵ calculated by Chernoff's bound. An itemset is interesting if $\hat{f}(X) \geq \theta - \epsilon$; otherwise it is uninteresting. Figure 4.2 illustrates the two states.
As the stream evolves and more transactions become available, ϵ gets smaller and thus, the interval of state interesting becomes smaller over time as illustrated in Figure 4.3. The interval for interesting itemsets, shown in gray in the figure, shrinks fast in the beginning and then slower and slower.
The DTM algorithm stores all interesting and singleton itemsets (independently of their current state). Similarly to other related stream mining algorithms, we assume that the algorithm is provided a buffer that can be used to store B transactions for some $B > 0$ integer.
The main steps of the DTM algorithm are given in Algorithm 3. It has three input parameters: A frequency threshold $\theta \in (0, 1]$, a confidence parameter $\delta \in (0, 1)$ for defining the dynamic confidence interval, and a buffer size $B \in \mathbb{N}$. For each itemset $X \subseteq I$ stored by the algorithm, it records the index τ of the transaction since X has (again) been stored by the algorithm ($X.\tau$), the number of occurrences of X in the transactions from transaction T_τ ($X.count$), the current state of X ($X.state$), and an estimate of the frequency of X ($X.est$). When a new transaction arrives, it is added to the buffer. If the buffer becomes saturated, it is first processed with Algorithm 4 and then emptied. Finally, when the algorithm receives an interestingness query for an itemset X, it returns "uninteresting" if X is not among the itemsets stored. Otherwise, it returns "interesting" if its estimated relative frequency $X.est$ is at least θ since $X.\tau$.

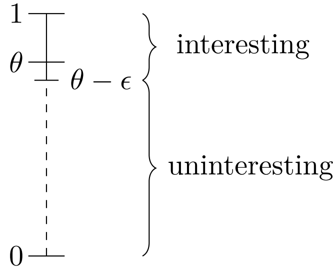


Figure 4.2.: Itemset states used by the DTM algorithm.

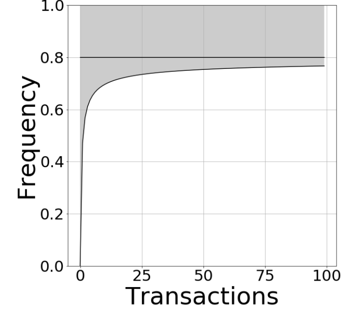


Figure 4.3.: Development of the regions interesting (top) and uninteresting (bottom) for the first 100 transactions and $\theta = 0.8$.

Algorithm 3 DTM Main

Parameters: frequency threshold $\theta \in (0, 1]$, confidence parameter $\delta \in (0, 1)$, and buffer size $B \in \mathbb{N}$

Initialization:
  for all $i \in I$ **do** $\{i\}.\tau = 1$, $\{i\}.count = 0$, $\{i\}.state = \text{Infreq}$

Processing transaction T_t at time t
  set Buffer[$t \bmod B$] $= T_t$
  if $t \bmod B = B - 1$ **then** process and empty the buffer // see Algorithm 4

Processing query X at time t
  if $(X.state = \text{Int}) \wedge (X.est \geq \theta)$ **then**
    return "interesting" with $X.oe$
  else
    return "uninteresting"

We now turn to the description of processing the buffer (see Algorithm 4). All itemsets are of state uninteresting initially. DTM does not store uninteresting itemsets, except for singletons. The state of an itemset X w.r.t. a data stream $\mathcal{S}_t = \langle T_1, \ldots, T_t \rangle$ is based on the following probabilistic reasoning: Let $p^{(t)}(X) = \mu > 0$. Since

$$p^{(t)}(X) = \frac{\mathbb{E}[X_1 + \ldots + X_t]}{t} \ ,$$

where $\mathbb{E}$ denotes the expected value, applying Chernoff bounds to Poisson trials we have

$$\mathbf{Pr}\left(\frac{X_1 + \ldots + X_t}{t} \leq (1 - \epsilon)\mu\right) \leq e^{-t\mu\epsilon^2/2}$$

for any $0 < \epsilon < 1$ (see, e.g., Ch. 4 of Mitzenmacher and Upfal (2005)). Bounding the right-hand side by δ, we get

$$\epsilon \geq \sqrt{\frac{-2\log\delta}{t\mu}} \ .$$

Algorithm 4 Dynamic Threshold Miner (DTM)

Input: frequency threshold $\theta \in (0,1]$, confidences $\delta \in (0,1)$, buffer size $B \in \mathbb{N}$

1: **function** ESTIMATE(integer s)
2:     **return** $\max\left(0, \left(1-\sqrt{\frac{-2\log\delta}{s\theta}}\right)\theta\right)$

Process the buffer at time t:
3: **for all** $\emptyset \neq X \subseteq I$ such that $\nexists Y \subset X$ with $Y.state$ = Unint **do**
4:     c = support count of X in the buffer
5:     **if** $|X| = 1$ **then**
6:         $X.count = X.count + c$
7:         **if** $X.count/t \geq$ Estimate(t) **then** $X.state$ = Int, $X.est = X.count/t$
8:         **else** $X.state$ = Unint
9:     **else** // $|X| > 1$
10:         **if** $X.state$ = Unint **then** // i.e., X is not in the data structure
11:             $M = (\text{Estimate}(t-B)\cdot(t-B)+c)/t$
12:             **if** $M \geq$ Estimate(t) **then**
13:                 $X.state$ = Int, $X.\tau = t - B + 1$, $X.count = c$, $X.est = M$
14:         **else**
15:             $X.count = X.count + c$
16:             $X.est = (\text{Estimate}(X.\tau-1)\cdot(X.\tau-1)+X.count)/t$
17:             **if** $X.est \geq$ Estimate(t) **then** $X.state$ = Int
18:             **else** $X.state$ = Unint

Thus, for any $t > \frac{-2\log\delta}{\mu}$, we have

$$\mathbf{Pr}\left(\frac{X_1+\ldots+X_t}{t} \leq \left(1-\sqrt{\frac{-2\log\delta}{t\mu}}\right)\mu\right) \leq \delta \ .$$

Now consider the case that

$$\frac{X_1+\ldots+X_t}{t} \leq \left(1-\sqrt{\frac{-2\log\delta}{t\theta}}\right)\theta \ . \tag{4.5}$$

One can easily check that if X was interesting, i.e., $\mu \geq \theta$, then we had

$$\left(1-\sqrt{\frac{-2\log\delta}{t\mu}}\right)\mu \geq \left(1-\sqrt{\frac{-2\log\delta}{t\theta}}\right)\theta$$

whenever

$$t = O\left(\frac{-\log\delta}{\theta}\right) \ .$$

Thus, the probability that (4.5) occurs and X is *interesting* is bounded by δ, implying that X is *not* interesting with probability at least $1 - \delta$.

Using the probabilistic reasoning sketched above, an itemset X is regarded by Algorithm 4 as uninteresting (Unint) for a data stream $\mathcal{S}_t$ if (4.5) holds and interesting otherwise. All singleton itemsets are processed and stored by the algorithm regardless of their state (see lines 5–8). For an itemset X of state uninteresting, DTM first calculates an estimate of its relative frequency (lines 11 and 1–2). If it does not satisfy condition (4.5), then the state is changed to interesting and X is stored (lines 12–13). Finally, in lines (15–18) the state is updated for interesting itemsets according to the probabilistic reasoning described above.

Implementation In this section, we describe some of the specifics and optimizations of the implementation of our DTM algorithm.

We experimented with two different implementations. The first is Apriori-based, the second one is based on the FP-Tree data structure. For the Apriori-based version, we use a column-based table as the buffer, where each column corresponds to one item. For every column, we store the set of ids of the transactions that contain the corresponding item. For this purpose we assign a number to each transaction, starting from 1. Counting itemsets is easy with this structure, as it suffices to compute the intersection of the sets corresponding to the columns of the items in the itemsets (Zaki et al., 1997). The size of the intersection is the support count of the itemset in the set of transactions in the buffer. We use classical transaction projection to reduce the incidence sets to the smallest possible size. This speeds-up further projections (Boley et al., 2009b).

For the FP-Tree-based design, we simply use an array for the buffer. Once the buffer is full or upon a query, we construct an FP-Tree from the buffer and enumerate itemsets with breadth-first-search (or level-wise). While depth-first-search is typically faster (cf., e.g., Qinghua Zou et al. (2002); Wang et al. (2003)), the design of the algorithm requires level-wise processing. To reduce the number of computed projections, we cache one projected tree for each level. The cache always stores the last projected tree for each level, except the current one. Once the projections at the current level have been computed, the next tree does not need to be reprojected from the root node. Rather, it can be projected from an already projected and cached tree, if a parent of the tree to be projected is in the cache. This improves the runtime efficiency, however at the expense of some additional memory. We have tested both implementations and found that the tree-based version is several times faster than the other one.

The itemsets stored by DTM are kept in a classical prefix tree structure with lexicographical sorting. Each node stores its first transaction, i.e., the time when it was created, the count, its state, and the optimistic estimate.

4.4. Empirical Evaluation

The goal of the empirical evaluation is to compare the performance of the algorithms developed within this thesis to other state-of-the-art algorithms. The main features considered for this comparison are the mining *quality*, the amount of *memory*, and the *runtime*. The mining quality is measured in terms of F-score, which is obtained by comparing the output of the approximate streaming algorithm to the output of an exact non-streaming algorithm processing the transactions from the stream. Given the correct output from the non-streaming algorithm, it is easy to compute the exact numbers of correctly identified frequent itemsets, i.e., true positives (TP), any additionally reported frequent itemset which is not frequent, i.e., false positives (FP), and any missing frequent itemset, i.e., false negatives (FN). The F_1-score or simply F-score is defined as:

$$\text{F-score} = \frac{2TP}{2TP + FP + FN} \quad .$$

4.4.1. Data sets

In this section, we describe the data sources used in the experiments. Our evaluation relies on classical publicly available real-world benchmark data sets and artificially generated data streams with known characteristics. Two popular sources for the task of frequent itemset mining are the UCI machine learning repository (Dua and Graff, 2019) and the Frequent Itemset Mining Dataset Repository[3]. While the former provides data sets for a broad range of data mining tasks, the latter focuses specifically on data sets for frequent itemset mining. Both sources provide classical data sets for the non-streaming setting. Since there is no equivalent source providing data streams and the above data sets are well-known in the frequent itemset mining literature, these standard data sets will be used for our evaluation. Data streams can be generated from them by feeding the transactions one after one to an initially empty stream or by repeatedly drawing the next transaction with replacement from the data set. The first approach results in deterministic data streams, which are limited in length to the number of transactions in a data set. The second one can be used to generate streams that are larger than the data set the stream is generated from. For some algorithms and parameters, the number of transactions that are required to reach their error bound exceeds the number of transactions in some data sets. This is a limitation of the first approach. The second approach results in streams that contain the same transaction multiple times. We have generated streams with 10 million transactions from these data sets by randomly drawing the next transaction. However, the experimental results show that these streams have very little variation and most algorithms find the frequent itemsets easily. The frequent itemsets for 1M transactions of such a stream are nearly identical or even identical to the frequent itemsets for 10M transactions of such a stream. Since our goal is to identify strengths and weaknesses, we do not report these results and work with the first approach to separate the algorithms more clearly. Table 4.2 lists the real-world benchmark data sets used in the empirical evaluation and their key characteristics.

[3] http://fimi.ua.ac.be/data/

Data set	Instances	Binary attributes	Average transaction length	Density
Adult*	48,842	104	9	0.086538
Chess (kr-vs-k)	28,056	58	7	0.120690
Kosarak	990,002	41,270	8	0.000196
Poker-hand	1,025,010	95	11	0.115789
T10I4D100k	100,000	870	10	0.011612
T40I10D100k	100,000	942	40	0.042044

Table 4.2.: Benchmark data sets used for the empirical evaluation.
* = considering all nominal columns

Some interesting aspects, such as, for example, the effect of the number of items, the impact of the average size of the transactions and others, can not be measured clearly on these classical benchmark data sets because the characteristics of interest can not be controlled for them. Therefore, additional artificially generated data streams are used to investigate the behavior of the algorithms for the variation of a single parameter under controlled settings. Two further advantages of artificial data streams are that they can generate streams of arbitrary length and with more variation than streams generated by repeatedly drawing from a fixed set. Artificial data streams are generated with the IBM quest market basket data generator (Agrawal and Srikant, 1994), which is the state-of-the-art tool for generating transactional benchmark data sets with certain characteristics. The program is widely used for the evaluation of algorithms mining frequent itemsets (see, e.g., Wang and Chen (2009)).
The parameters of this tool are:

- D: number of transactions in k
- N: number of different items in k
- T: average number of items per transaction
- I: average length of maximal patterns
- L: number of patterns
- C: correlation between patterns.

To obtain data streams with different characteristics, all parameters were systematically modified. Table 4.3 lists the distinct values used for each parameter. The start configuration is $D = 1,000$, $N = 10$, $T = 20$, $I = 4$, $L = 10k$, and $c = 0.5$. All other configurations are obtained from this start configuration by modifying the values of a single parameter. In total, 32 artificial data streams have been generated by different variations of these parameters.
To distinguish between the two very different data sources, the real-world benchmark data sets will be referred to as *real-world* or *UCI* data and the artificially generated data streams as *artificial* or *QUEST*.

Parameter	Semantic	Values
D	number of transactions in k	$100; 1,000; 10,000$
N	number of items in k	$1; 10; 100; 1,000$
T	average number of items per transaction	$10; 20; 30; 40; 50; 60; 70$
I	average length of maximal patterns	$4; 6; 8; 10; 12; 14; 16; 18; 20$
L	number of patterns	1k; 10k; 100k; 1M
C	correlation between patterns	$0; 0.25; 0.5; 0.75; 1$

Table 4.3.: Parameters used for synthetic data stream generation.

4.4.2. Design of Experiment

In this section, we describe the design of our experiments. Each data set is considered and processed as a data stream providing the transactions one after the other to the algorithm. Each algorithm is queried at regular intervals for the set of frequent itemsets. These intervals are data set specific and are chosen equidistant such that after each additional tenth of the data stream a new result is obtained. For each intermediate result, the ground truth is computed with an exact non-streaming algorithm, considering the first n transactions. The ground truth serves as a gold-standard, against which the outputs of the approximate streaming algorithms are compared. Some algorithms require a specific block size for their error guarantee to hold. For them, we compute intermediate and ground truth results after as many blocks have been processed as have been filled completely by the i-th tenth of the stream and compare this result to the ground truth computed for the same number of transactions.

The algorithms are evaluated at five thresholds corresponding to approximately 1k, 5k, 10k, 50k, and 100k frequent itemsets. These thresholds cover a broad range of applications. Small sets of frequent patterns can be manually investigated. This, however, becomes more and more challenging with larger numbers of frequent itemsets. While large sets of frequent patterns are computationally more interesting, they are practically less relevant because patterns with lower frequency are often less significant. Hence 100k seems a reasonable bound. As the number of frequent itemsets for a data stream might change over time, the thresholds are determined based on the ground truth result for the entire stream.

The ten results for each data stream and the five thresholds give a total of 50 results for each data set. These results are further condensed to obtain a single number from all measurements considering either the average or the worst-case performance. The worst-case corresponds to the minimal F-score, the maximal memory, and the maximal runtime. It provides a lower bound of the performance of an algorithm. Since the real-world and artificial data exhibit different characteristics, the results will be aggregated for both data sources separately and will be reported whenever appropriate.

For the empirical comparison, the most relevant algorithms mining data streams under the landmark model were chosen (cf. Section 3.2). In particular, Carma (Hidber, 1999), Lossy Counting (Manku and Motwani, 2002), SApriori (Sun et al., 2006), FDPM (Yu et al., 2004) and EStream (Dang et al., 2008). The algorithms DSM-

FI (Li et al., 2008a), Stream Mining (Jin and Agrawal, 2005), hMiner (Wang and Chen, 2009), and SA-Miner (Li et al., 2014) have such specific properties that make them unsuitable for the general mining setting. In particular, the enumeration scheme of DSM-FI requires the algorithm to check very large numbers of non-frequent itemsets, making it very slow. This is an undesired property for any stream mining algorithm. Since hMiner enumerates every subset of each transaction and Stream Mining enumerates almost every subset of a set of transactions, both approaches result in an exponentially large set and work therefore only for very short transactions. hMiner has been evaluated with an average maximal itemset size of 4, Stream Mining for transactions with size up to 6. Finally, SA-Miner is overly complicated. Its support approximation approach with artificial neural networks requires a lot of data for the training of the network, yet even with this data, it is unclear whether such a support approximation relation can be learned for most real-world data sets.
All algorithms were (re)implemented in the Java programming language and use similar data structures whenever appropriate. The individual implementation of an algorithm might be less or more sophisticated than the implementation by the respective authors, but this reimplementation with similar data structures and transaction processing ensures maximal comparability among the algorithms. Our goal is not to present the most sophisticated implementations of these algorithms, but rather to have comparable implementations. While the results can altogether be further improved concerning memory and runtime through more sophisticated data structures, they demonstrate the performance of the algorithms in relation to each other. For Carma, we buffer the transactions and process all transactions from the buffer at once with the FP-growth algorithm. Since this algorithm has no formal error bound, this does not change the error, but it significantly improves the processing speed. We found that our implementation of the FDPM algorithm was slow due to its intermediate pruning of the set of patterns from the current batch. To speed up the algorithm, we skip this pruning and generate an FP-Tree for each batch, which makes our modified implementation of the algorithm very fast. We refer to the modified version as FDPM*.
Most algorithms have specific parameters that affect their performance. To better understand their impact and to have a fair comparison, we tune these parameters before the comparison of algorithms with each other. The details of our parameter tuning are provided in the Appendix.
All experiments were run on computers with the following hard and software configuration: Intel(R) Xeon(R) CPU E5-2650 @ 2.00GHz equipped with 64GB of memory running Debian GNU/Linux 9 with Kernel Version 4.18.10 and the OpenJDK Java version 1.8.0_181.

4.4.3. Experimental Comparison

This section compares the performance of the algorithms to each other based on the optimal parameters as evaluated in the Appendix. Table 4.4 summarizes them. The comparison considers the effect of varying the following aspects:

- Frequency threshold

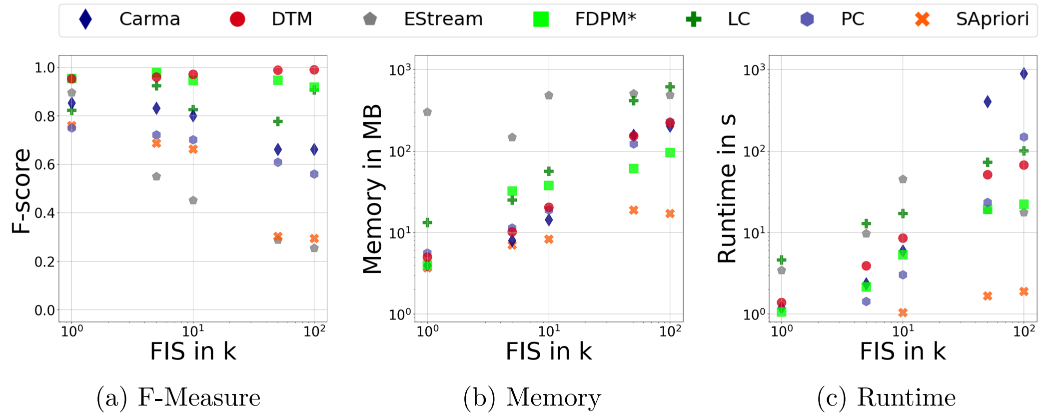
Algorithm	**Parameter(s)**
DTM	$\delta = 0.1$
EStream	$\epsilon = 0.1\theta; k =$ as from ground truth
Lossy Counting (LC)	$\epsilon = 0.01\theta$
Partial Counting (PC)	strategy = average estimation
FDPM*	$k = 5; \delta = 0.0001$
SApriori	$\epsilon = 0.002\theta; \delta = 0.0001$

Table 4.4.: Optimal parameters for frequent itemset mining algorithms.

- Length of the data stream
- Number of different items
- Average number of items per transaction
- Average length of maximal patterns
- Number of patterns
- Correlation between patterns

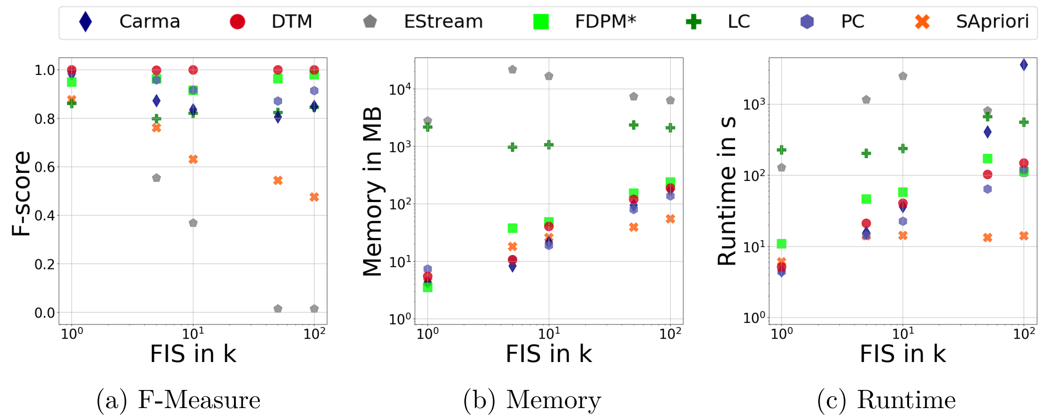
The first two aspects will be evaluated both on the real-world benchmark data from Table 4.2 and the artificial data streams obtained with the IBM quest market basket data generator. The further aspects can only be evaluated on the artificial data streams, as these parameters can not be controlled for the real-world data.
For a single data stream, ten results are computed as described previously. They are aggregated based on a worst-case scenario, taking the minimum F-score and the maximum memory and runtime, as they provide a lower bound on the performance of an algorithm. A good lower bound is very desirable for a new unknown data stream. Across the various data streams, these worst-case results are averaged to obtain a single number for all data streams.

Frequency threshold As described above, frequent itemsets are mined at five data stream specific thresholds corresponding to approximately 1k, 5k, 10k, 50k, and 100k frequent itemsets. These values cover a broad range of applications. Results for the real-world data streams are shown in Figure 4.4 and those for the artificial data streams in Figure 4.5. To obtain a unique order we compute the averages over all threshold values and rank the algorithms by this order.
For the F-scores in Figures 4.4a and 4.5a, we observe that the order of the algorithms is consistent across both data sources. The overall rank of the algorithms from best to worst is as follows: DTM, FDPM*, Partial Counting, Lossy Counting, Carma, SApriori, and EStream. In case of high frequency thresholds and few frequent itemsets, SApriori and EStream perform way better than for lower frequency thresholds.



(a) F-Measure (b) Memory (c) Runtime

Figure 4.4.: Average effect of varying the frequency threshold on (a) F-Measure, (b) Memory, and (c) Runtime for the UCI data streams.

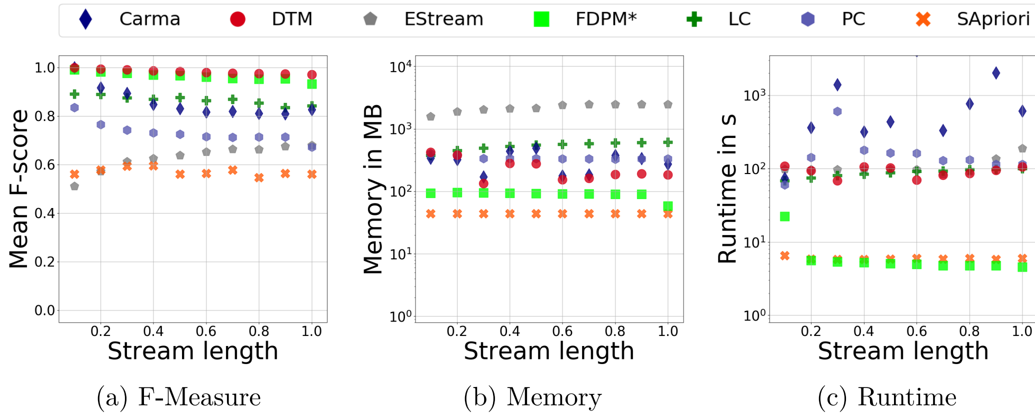


(a) F-Measure (b) Memory (c) Runtime

Figure 4.5.: Average effect of varying the frequency threshold on (a) F-Measure, (b) Memory, and (c) Runtime for artificial QUEST data.

This is probably due to the length bias introduced by the frequency threshold. For higher frequency thresholds there are fewer long itemsets, making the mining problem easier.

Concerning the memory needed to process the streams, we observe different orders for the UCI data (Figure 4.4b) and the QUEST data streams (Figure 4.5b). Noteworthy is that there is a clear trend that at lower thresholds the algorithms all require more memory than at higher thresholds. For both data sources SApriori requires significantly less memory than the other algorithms. However, it achieves poor F-scores. On the UCI data, FDPM* comes next, then Carma, Partial Counting, and DTM. For the QUEST data, we observe a different order. In particular, Partial Counting requires



(a) F-Measure (b) Memory (c) Runtime

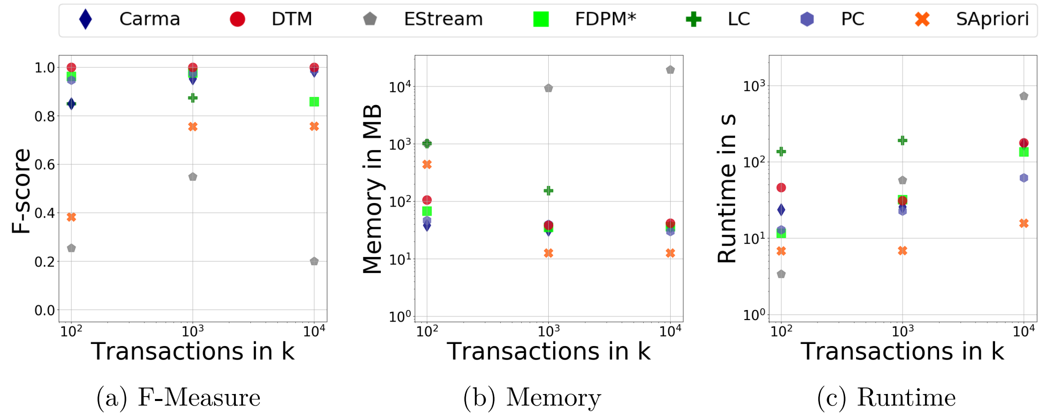
Figure 4.6.: Effect of varying the length of the data stream on (a) F-Measure (mean), (b) Memory (worst case), and (c) Runtime (worst case) for the UCI data streams.

the third least memory, followed by Carma, DTM, and FDPM*. In both cases, Lossy Counting requires the second to most memory and EStream requires the most. They both need an order of magnitude more memory than the other algorithms (Figure 4.5b).

We now turn to the description of the runtime. See Figures 4.4c and 4.5c for the results on real-world and artificial data streams respectively. First of all, we observe the general tendency that with an increasing number of frequent itemsets the algorithms take longer to process the data streams. While this general trend holds for both data sources, it is more apparent for the UCI data. On the QUEST data, there is a decrease in runtime for large numbers of frequent itemsets for EStream. The algorithm is unable to produce these results for some data streams. As we consider only completed results, the algorithm obtains hence a lower total runtime. Surprisingly, Partial Counting and DTM are amongst the fastest algorithms for the QUEST data. They rank second and third after SApriori. For the real-world streams, DTM is faster than Partial Counting. For these streams, they are both slower than SApriori and FDPM*.

In summary, DTM performs best in terms of F-score. Partial Counting is still very competitive on the artificial streams and at the same time very memory efficient and the fastest algorithm amongst those obtaining high F-scores.

Length of the data stream To analyze the effect of the length of the data stream, intermediate results for each data set are produced at regular intervals, corresponding to one-tenth of the data stream. For the artificial data streams, we consider streams with 100k, 1M, and 10M transactions and produce intermediate results after each tenth for every stream. For each data stream, we extract the average F-score for the real-world and lowest F-score for the artificial streams, the highest memory consumption, and maximal runtime. We average these results over all five frequency thresholds. The results for the real-world and artificial streams are shown in Figures 4.6 and 4.7. We discuss both data sources in turn.



(a) F-Measure (b) Memory (c) Runtime

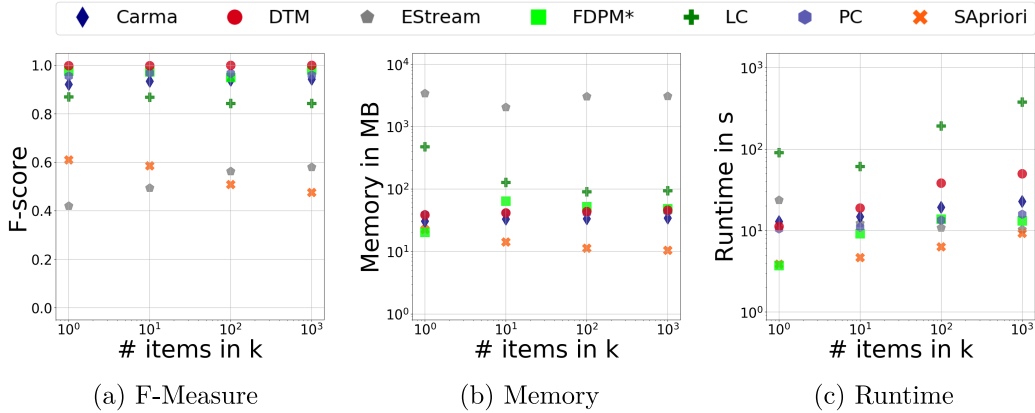
Figure 4.7.: Effect of varying the length of the data stream on (a) F-Measure, (b) Memory, and (c) Runtime for artificial QUEST data.

Some algorithms did not complete for all thresholds and stream length; we ignore these missing results. On the real-world data, we can observe for the average F-score (Figure 4.6a) that the EStream algorithm is the only one that profits from increasing stream length. Still, its F-score ends up second to last. Only SApriori performs worse. It has no clear tendency as a function of the stream length. All other algorithms show a drop in their F-score with increasing stream length. DTM consistently obtains the best F-scores, followed by FDPM*; Partial Counting ranks fifth.

Concerning memory consumption (Figure 4.6b) one can observe that only the two algorithms SApriori and FDPM* require less memory than DTM. The next best is Carma followed by Partial Counting. Lossy Counting and EStream show an increasing memory demand as the stream gets longer. Finally, we note that SApriori, FDPM*, and Partial Counting require constant memory, irrespective of the stream length.

Figure 4.6c shows the maximum runtime for each update step, i.e., each point represents the individual time required to process one-tenth of the stream. The overall order from fast to slow is SApriori, FDPM*, Lossy Counting, DTM, EStream, Partial Counting, and Carma. Carma shows the most unstable processing times.

We now turn to the artificial data streams (see Figure 4.7). For the streams with 10 Million transactions, Lossy Counting did not have sufficient memory to complete. Concerning the worst-case F-score (Figure 4.7a), we observe that it increases for the algorithms SApriori and Carma with the stream length. This is because we evaluate the experiment after every 1/10 of the stream. Thus, longer streams provide more transactions and hence, the algorithms can build more accurate statistics. DTM is again best in all cases with an F-score of 0.99, followed by Partial Counting which obtains an F-score of 0.97. While FDPM* is competitive up to 1M transactions, its F-score drops for streams with 10M transactions. We attribute this to the fact that the algorithm processes more buffers for longer streams, which causes it to miss more and more frequent itemsets. The effect of the stream length on memory is shown in Figure

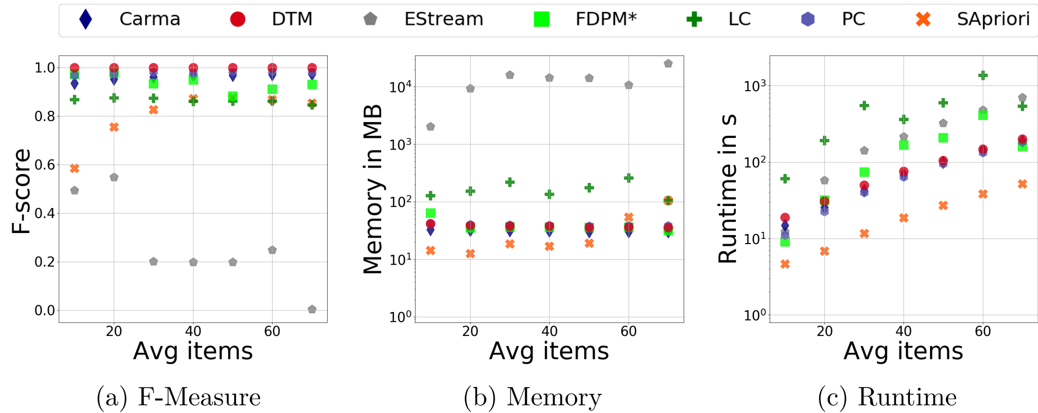


(a) F-Measure (b) Memory (c) Runtime

Figure 4.8.: Worst case effect of varying the number of items on (a) F-Measure, (b) Memory, and (c) Runtime for artificial QUEST data.

4.7b. For streams with 10M transactions, most algorithms require a similar amount of memory. SApriori requires significantly less and EStream significantly more than the other algorithms. Figure 4.7c shows the maximal time to process one update, i.e., one-tenth of the data stream. As the streams get longer, these individual updates have to process more transactions and require more time. Hence, the runtime increases. This is well expected, in contrast to the updates on the real-world data streams with constant size.
The results from this experiment clearly show the superior performance of DTM in terms of F-score. Partial Countings inference mechanism is well-suited for the artificial data streams but seems less appropriate for the real-world data streams, where its F-score decreases with increasing data stream length.

Number of different items To test the impact of the number of different items in the data streams, artificial data streams were generated with different alphabet sizes. In particular, the following sizes were used: 1k, 10k, 100k and 1M. Similarly to the previous experiment, we average the results over all five frequency thresholds. The results for these experiments are shown in Figure 4.8. Regarding the F-score (Figure 4.8a), we observe that most algorithms are agnostic to the size of the alphabet. EStream profits from larger alphabets, while SApriori performs slightly worse with increasing alphabet size. The best F-scores archive the algorithms DTM, FDPM*, and Partial Counting.
Regarding the memory consumption (Figure 4.8b), the algorithms can be partitioned into three categories. Those with a memory footprint independent of the number of items (EStream), those that require more memory as the number of items increases (FDPM*, DTM, Carma, Partial Counting), and algorithms which require less memory as the number of items grows (Lossy Counting, SApriori).
The runtime results (Figure 4.8c) indicate that all algorithms except for EStream require more time to process data streams with larger alphabets. EStream, on the other hand, gets faster as the alphabet grows. Lossy Counting and DTM are the two algo-

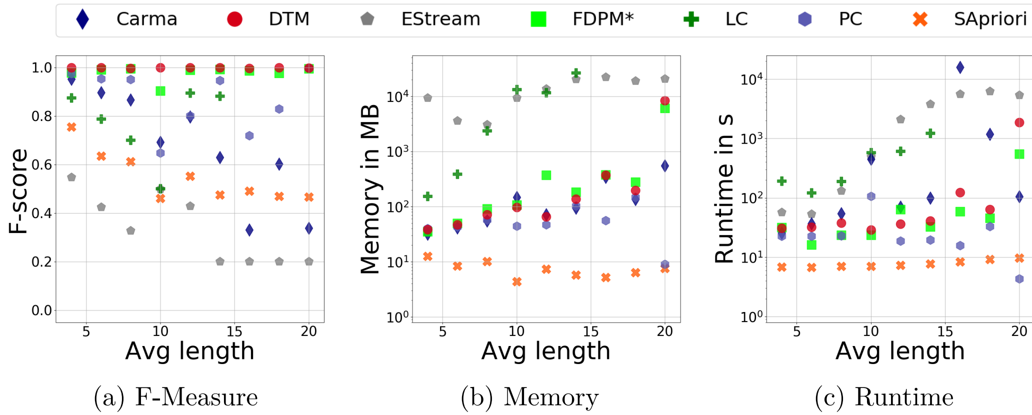


(a) F-Measure (b) Memory (c) Runtime

Figure 4.9.: Effect of varying the average number of items per transaction on (a) F-Measure, (b) Memory, and (c) Runtime for artificial QUEST data.

rithms with the largest increase in runtime with growing alphabet size. We observe that Partial Countings runtime increases slowest with the increasing number of items. More precisely, the runtime increases from 10,500 seconds for 1,000 different items to 16,000 seconds for 1M different items. Thus the algorithm is particularly well-suited for streams with very large ground sets I.

Average items per transaction For the artificial data streams, we modify the average number of items per transaction as defined in Table 4.3, considering the values 10, 20, 30, 40, 50, 60, and 70. The results are averaged over all five frequency thresholds and shown in Figure 4.9. Note first that for EStream and Lossy Counting, some experiments did not run. This is visible for EStream in the F-score for 70 items (Figure 4.9a). All other algorithms perform very well, i.e., with F-scores close to 1, irrespective of the average number of items per transaction. In particular, DTM, Partial Counting, and Carma obtain very high F-scores.
The memory consumption is shown in Figure 4.9b. We clearly observe that the algorithms fall into two categories: Those with constant memory irrespective of the average number of items per transaction and those that require more memory as the average number of items per transaction increases. Carma, DTM, Partial Counting, and FDPM* require almost the same amount of memory irrespective of the number of items. EStream and SApriori require more memory as the average number of items per transaction increases. Finally, Lossy Counting shows no clear pattern as a function of the number of average items per transaction.
The runtime increases with an increasing average number of items per transaction for all algorithms (see Figure 4.9c). The order from fast to slow is SApriori, Partial Counting, Carma, DTM, FDPM*, EStream, and finally Lossy Counting. DTM and Lossy Counting have the smallest increase in runtime which makes these algorithms suitable to process very large transactions with several hundred items per transaction. Partial Counting is efficient both in terms of memory and runtime and has a slightly



(a) F-Measure (b) Memory (c) Runtime

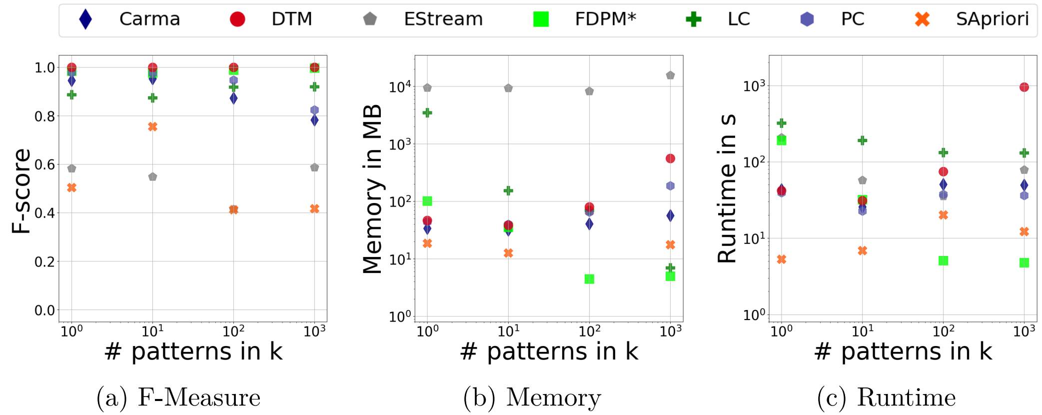
Figure 4.10.: Effect of varying the average length of maximal patterns on (a) F-Measure, (b) Memory, and (c) Runtime for artificial QUEST data.

better F-score than Carma which requires a little less memory and about the same time. DTM, on the other hand, is interesting for transactions with 100 items or more, as it can be expected to be the fastest algorithm among those with high F-scores for such streams.

Average length of maximal patterns To identify the effect of the average length of the maximal patterns, this length was systematically modified. More precisely, we considered lengths from 4 to 20, at increments of 2. As in the previous experiments, we average the results over all five frequency thresholds. They are reported in Figure 4.10. Note that Lossy Counting was unable to complete from the average length of 16 onwards. The F-scores (Figure 4.10a) are the least accessible, as they show some fluctuation. We observe the general tendency that the F-score drops as the average length increases. But this general trend shows several exceptions. We note that our algorithms are amongst those with the highest F-scores over a broad range. In particular, DTM obtains the highest F-scores and only FDPM* manages to be constantly better than Partial Counting.

The memory consumption is depicted in Figure 4.10b. We observe that for all algorithms, except for SApriori, the memory consumption clearly increases with the average length of maximal patterns. Out of the three algorithms with the highest F-scores, Partial Counting requires the least memory, followed by FDPM*. While DTM requires less space than FDPM* in many cases, it requires more memory for 20 items and this result dominates the overall average.

The runtime results are reported in Figure 4.10c. EStream shows the largest increase in runtime as the average length of maximal patterns increases. SApriori is at the opposite extreme with the smallest increase. All other algorithms rank somewhere in between. Partial Counting is overall the second fastest algorithm, followed by FDPM* and then DTM.



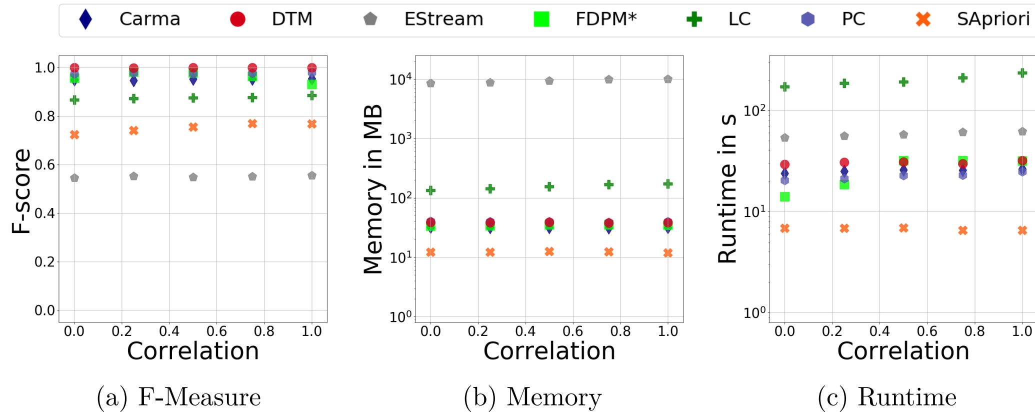
(a) F-Measure (b) Memory (c) Runtime

Figure 4.11.: Effect of varying the number of patterns on (a) F-Measure, (b) Memory, and (c) Runtime for artificial QUEST data.

Amongst the algorithms with the best F-scores, Partial Counting requires the least memory and is fast. This makes the algorithm a strong candidate for settings in which the available memory or the processing time is limited, such as, for example, mobile devices. DTM, on the other hand, obtains consistently the best F-scores.

Number of patterns For the artificial data streams, we considered streams with 1k, 10k, 100k, and 1M patterns. The averages over the worst-case results for all five frequency thresholds are shown in Figure 4.11. FDPM* and Lossy Counting did not complete for the two lowest frequency thresholds at 100k and 1M patterns. This can be seen in the results, for example, in Figure 4.11b, which shows that they require the least memory. With respect to the F-score (Figure 4.11a), we observe that with an increasing number of patterns it decreases slightly, except for DTM, FDPM* and Lossy Counting. If we rank the algorithms the top three are DTM, FDPM*, and Partial Counting.
The memory consumption is shown in Figure 4.11b. All algorithms require more memory as the number of patterns increases. Note that FDPM* and Lossy Counting appear to require less memory at lower thresholds, but this is purely caused by the fact that they did not produce all results at these thresholds.
The runtime results are reported in Figure 4.11c. With an increasing number of patterns, all algorithms tend to run longer. The overall impact of the number of patterns on the runtime is rather small. All algorithms are fast in this experiment, except for DTM. For 1M patterns, the algorithm requires the most time to compute the update. However, unlike FDPM* it completed for all data streams and thresholds.

Correlation between patterns We generate synthetic data streams with different correlation strengths between the patterns. In particular, we investigate data streams with correlation 0, 0.25, 0.5, 0.75, and 1. The results are depicted in Figure 4.12. Overall, this parameter shows little effect in all three plots. The F-scores (Figure 4.12a) for each algorithm vary only within a very close range. Only FDPM* performs slightly worse



(a) F-Measure (b) Memory (c) Runtime

Figure 4.12.: Effect of varying the correlation between patterns on (a) F-Measure, (b) Memory, and (c) Runtime for artificial QUEST data.

Algorithm	**F-score**
DTM	0.9805
FDPM*	0.9598
Partial Counting	0.9002
Lossy Counting	0.8550
Carma	0.8523
SApriori	0.6086
EStream	0.4356

Table 4.5.: Average F-score over all frequent itemset mining experiments.

for the correlation 1.0. DTM obtains once more the best F-score. The impact on memory (Figure 4.12b) is also very limited. Most notable are some runtime effects (Figure 4.12c). While overall there is very little effect, FDPM* and Lossy Counting require more time with an increasing correlation between the patterns. This effect is stronger for FDPM*. Amongst the algorithms with high F-scores, Partial Counting is the fastest.

Résumé We briefly summarize the seven experiments of this section. To obtain an overall picture, we compute the average F-score and an average rank for both memory and runtime over all experiments. These metrics are useful to derive a ranking of the algorithms. Table 4.5 shows the average F-scores. Not only does our DTM algorithm obtain the best overall F-score, but it has also the best F-score in all experiments. Other good candidates with respect to the F-score are FDPM* and Partial Counting.

Regarding the memory, the ranking of the algorithms from small to large is SApriori, Partial Counting, Carma, FDPM*, DTM, Lossy Counting, and finally EStream. We observe that Partial Counting comes second after SApriori. This makes our Partial Counting algorithm a good candidate if the goal is to obtain good results with limited memory which is the case e.g. for mobile device applications.
If we rank the algorithms from fast to slow, we obtain the following order: SApriori, Partial Counting, FDPM*, DTM, Lossy Counting, Carma, and EStream. SApriori is first, but given its overall low F-score, the algorithm does not seem to be a good choice. Partial Counting ranks again second and is thus not only interesting if the memory is limited, but also when the update of the results should be fast, i.e., when the speed at which new transactions arrive is high, such as in classical big data settings.
We conclude that DTM provides the best F-score over a large set of datasets and parameters and is hence the most accurate algorithm. If, however, the F-score is not the most important and an approximation of the family of frequent itemsets is sufficient, then Partial Counting is the most memory efficient and fastest algorithm amongst the top three in terms of F-score.

4.5. Discussion

Before we discuss the results of this chapter, we briefly recall them. We have presented two new algorithms with probabilistic reasoning for the problem of mining frequent itemsets from data streams under the landmark model. Both algorithms have the benefit that they can work with transaction buffers, but still, produce results after any number of transactions. The DTM algorithm provides a theoretical proven probabilistic error bound which guarantees that the chance that an itemset is frequent but not reported as such is bounded by $1 - \delta$, for some user-defined confidence δ. A common feature of both algorithms is that they use an estimated frequency for the past.
The Partial Counting algorithm mines frequent itemsets with a memory-bound proportional to the number of frequent itemsets in the stream. It starts counting an itemset, once all subitemsets are frequent and uses statistical inference to estimate the unobserved frequency. To achieve good F-scores, our algorithm requires less memory and time than the state-of-the-art algorithms.
Partial Counting tries to estimate the frequency of an itemset based on the frequencies of its direct subsets. We proposed three mechanisms for this estimation: upper bound, minimum, and average estimation and compared them empirically. The strategies follow different reasonings. The upper bound estimation uses the single smallest count of a subset as the estimation for the past. The other two strategies work with conditional probabilities. They compute the conditional probability of a k-itemset given its k $(k-1)$-subitemsets and multiply this conditional probability with the count of the subitemset, for each subitemset. These conditional probabilities become more precise as further and further transactions are added to the stream. The minimum strategy takes simply the minimum of all the estimations obtained from the $(k-1)$-subitemsets, whereas the average strategy computes the average over all subitemsets. To compute

the estimates based on the conditional probabilities, the algorithm tracks and stores the counts of the k-subsets when it starts to count a k-itemset. This is an overhead in terms of memory, which is feasible only for small k.

The algorithm is very similar to Carma (Hidber, 1999). The central difference comes from the estimation approach employed. Carmas strategy resembles the upper bound estimation, but differs in a small detail. While Partial Counting uses $\min\left(\lfloor\theta(t_X - 1)\rfloor, \hat{C}_X^{1,t_X-1}\right)$ to obtain an estimate for the past, Carma adds the cardinality of the itemset X to the frequency threshold multiplied by the number of transactions in the past, i.e., it computes $\min\left(\lfloor\theta(t_X - 1)\rfloor + |X|, \hat{C}_X^{1,t_X-1}\right)$. As the experimental results have demonstrated, our approach leads to better F-scores.

Our DTM algorithm is a probabilistic algorithm designed for answering interestingness queries on data streams with transactions generated by independent probability distributions. The empirical results on real-world benchmark and artificial data streams demonstrate that it achieves an excellent performance not only on mining interesting itemsets but also on frequent itemset mining from data streams, even with correlated transactions. In all of our experiments, it achieves the best F-scores and outperforms all state-of-the-art algorithms. Additionally, we observe, that its runtime increases slower than the runtime of other algorithms with an increasing number of items per transaction (cf. Figure 4.9). The algorithm can work with arbitrary buffer sizes and even process incomplete buffers, in contrast to many other approaches. Interestingly, the guarantee of DTM is independent of the buffer size. The buffer has only an impact on the update frequency and even incomplete buffers can be processed without any loss of the formal error bound. The algorithm profits from larger data streams as the error bound gets tighter as more and more transactions arrive in the stream. It is most similar to FDPM, but differs from it in the following features:

1. DTM estimates the frequency for the uncounted period,
2. it applies a different probabilistic reasoning, and
3. its output is not necessarily sound.

We conclude that our two algorithms presented in this chapter provide clear algorithmic advantages over the state-of-the-art approaches. In particular, DTMs high F-scores make this algorithm the method of choice, when exact results are required. Partial Counting is suitable if the result does not need to be as precise, but the memory and time are more restricted and the size of each transaction is limited.

4.6. Summary

This chapter proposed two new algorithms for the task of frequent itemset mining from transactional data streams under the landmark model. Although they work with a transaction buffer, they are still able to produce a result after *any* number of transactions. All state-of-the-art algorithms process transactions either individually, with low throughput, or in mini-batches with less flexibility, providing results with probabilistic

guarantees only for *completely full* mini-batches. In contrast, the probabilistic reasoning of our algorithms is independent of the number of transactions processed in each update. They combine thus the flexibility of the transaction-based update with the time advantage of mini-batch-based algorithms. For the DTM algorithm, we have proven a bound that gets tighter as more and more transactions are processed. Our extensive empirical experiments have demonstrated a superior F-score of our DTM algorithm for all but one dataset.

5. Strongly Closed Itemset Mining from Transactional Data Streams

This chapter is concerned with mining strongly closed itemsets, a parameterized subset of frequent itemsets. We start with a motivation for the choice of this pattern class in Section 5.1. Our main contribution, the Strongly Closed Stream Mining (SCSM) Algorithm, is described in Section 5.2. It incrementally updates the family of strongly closed sets based on a fixed size sample. This characteristic essentially allows the algorithm to compute an updated result for any number of transactions, once the initial sample is complete. We extensively evaluate the algorithm empirically in Section 5.3. Our empirical results confirm both the compactness of strongly closed sets, as well as the high approximation quality of our algorithm. In Section 5.4, we empirically demonstrate their suitability for two practical applications: concept drift detection and product configuration recommendation. We discuss the results of this chapter in Section 5.5 and summarize finally our contributions in Section 5.6.

5.1. Motivation

In the previous chapter, we proposed two algorithms for approximating the family of all frequent itemsets from transactional data streams and empirically demonstrated that they are able to maintain even 100K frequent itemsets in practically feasible time. While this order of magnitude is typically sufficient for mining short frequent itemsets, it is less suitable for generating long patterns. Indeed, recall from Chapter 2 that for a ground set of size n, there might be $2^n - 1$ non-empty frequent itemsets in a data stream. Assuming a *fixed* capacity of maintaining at most K patterns, the length of a longest frequent itemset is bounded by $\log_2 K$. This follows from the fact that all subsets of an itemset must be frequent themselves. The potentially huge number of the output patterns has not only a negative impact on the update time of any such data stream mining algorithm but also affects the memory. A common solution to reduce the number of output itemsets is to increase the frequency threshold. However, this approach suffers from a language bias towards short patterns; this is precisely the problem we want to address.

Another approach to deal with the time and space complexity problem above is to generate some compact representation of the family of frequent itemsets. A natural candidate would be the class of maximal frequent itemsets. If, however, P $\neq$ NP, this class of patterns cannot be enumerated efficiently (Boros et al., 2003). A different option could be to consider the family of closed frequent itemsets, which can be listed with polynomial delay (see, e.g., Ganter and Reuter (1991); Gély (2005)). While there are

several fast algorithms mining closed frequent itemsets from data streams (see, e.g., Tang et al. (2012); Yen et al. (2014)), the number of closed itemsets still may be huge and hence not compact enough. In particular, in the landmark model most frequent itemsets will typically become (frequent) closed. This is perhaps one of the main reasons why the vast majority of algorithms for closed frequent itemset mining in data streams focuses on the sliding window model. Thus, closed frequent patterns do not solve the problem of reducing the output patterns feasibly for the landmark model. Last but not least, one may consider the family of crucial frequent itemsets (Das and Zaniolo, 2016), a subset of all frequent closed itemsets containing all maximal frequent itemsets. While crucial itemsets provide the theoretically most compact lossfree representation of frequent itemsets (including support counts), they do not solve the problem either. The difference between closed and crucial frequent itemsets typically vanishes with increasing data stream length. In addition, it is an open problem whether or not crucial itemsets can be generated with polynomial delay.

To overcome the limitations of the above pattern classes, we give up the requirement for lossless representations of frequent itemsets and consider the problem of mining some *compact lossy* "synopsis" able to capture certain essential information about the data stream. This problem relaxation is motivated by the fact that completeness is not necessary to capture fundamental characteristics of a data stream[1]. We will experimentally confirm this claim in Section 5.4 for the practical problems of concept drift detection and product configuration recommendation.

Clearly, the pattern language providing lossy synopses must be chosen carefully; otherwise the essential information to be preserved might be lost in the compact representation. As we empirically show, frequent strongly closed itemsets, a parametrized subfamily of frequent closed itemsets provides such a lossy synopsis. Accordingly, in this chapter, we consider the problem of listing $\tilde{\Delta}$-closed itemsets from data streams under the landmark model, i.e., Problem 4 defined in Section 2.3 (page 27). For this problem, we present an efficient algorithm with a completeness guarantee with respect to a reservoir sample. The sample size will be derived from Hoeffding's inequality in a way that with high probability, the relative frequency of any itemset in the sample deviates from the relative frequency in the data stream at most by a small user-defined error.

The parameter $\tilde{\Delta}$ controls the output size and hence the memory. With increasing $\tilde{\Delta}$, the number of itemsets decreases fast (Boley et al., 2009b; Trabold and Horváth, 2017). It is important to note that, in contrast to frequency-based pruning, $\tilde{\Delta}$ does not limit the pattern length. Our empirical results in Section 5.3 confirm both the compactness of strongly closed sets as well as the high approximation quality of our algorithm.

5.2. The Strongly Closed Stream Mining Algorithm

In this section, we present our Strongly Closed Stream Mining (SCSM) algorithm for the $\tilde{\Delta}$-Closed Set Listing problem (Problem 4) defined in Section 2.3 (page 27). To tackle massive data streams in feasible time, we approximate the $\tilde{\Delta}$-closed sets for a data

[1] Consider the mp3 audio format as a practical example.

TID	Items
1	bd
2	bd
3	bd
4	ad
5	ad
6	cd
7	cd
8	c
9	c

(a) Transactions

Itemset	Support
d	7
ad	2
bd	3
cd	2

(b) 2-closed itemsets for transactions 1–8

Itemset	Support
c	4
d	6
ad	2
bd	2
cd	2

(c) 2-closed itemsets for transactions 2–9

Figure 5.1.: Strongly closed itemsets example.

stream $\mathcal{S}_t = \langle T_1, \ldots, T_t \rangle$ at time t from a *fixed size* random sample $\mathcal{D}_t$ generated from $\mathcal{S}_t$ without replacement. Since the order of the elements in the sample does not matter, $\mathcal{D}_t$ is regarded as a transaction database. The size s of $\mathcal{D}_t$ is chosen in a way that for all $X \subseteq I$, the discrepancy between the relative frequency of X in $\mathcal{S}_t$ and that in $\mathcal{D}_t$ is at most ϵ with probability at least $1 - \delta$, i.e., s satisfies

$$\mathbf{Pr}\left(\left|\frac{|\mathcal{S}_t[X]|}{t} - \frac{|\mathcal{D}_t[X]|}{s}\right| \leq \epsilon\right) \geq 1 - \delta \tag{5.1}$$

for any $X \subseteq I$. The parameters ϵ (*error*) and δ (*confidence*) are specified by the user. Our extensive experiments in Section 5.3.5 show that a very close approximation of the true family of $\tilde{\Delta}$-closed itemsets can be obtained in this way.
Our algorithm recalculates the family of $\tilde{\Delta}$-closed itemsets *not* after each new transaction, but either upon request or after b new transactions have been received since the last update, where b, the *buffer size*, is specified by the user. Given $\mathcal{S}_t = \langle T_1, \ldots, T_t \rangle$ and $\mathcal{S}_{t'} = \langle T_1, \ldots, T_t, T_{t+1}, \ldots, T_{t'} \rangle$ with $t' - t \leq b$, the new sample $\mathcal{D}_{t'}$ of $\mathcal{S}_{t'}$ is computed from the old sample $\mathcal{D}_t$ by $\mathcal{D}_{t'} = \mathcal{D}_t \ominus \mathcal{D}_{\text{del}} \oplus \mathcal{D}_{\text{ins}}$, where $\mathcal{D}_{\text{del}}$ (resp. $\mathcal{D}_{\text{ins}}$) is the multiset of transactions to be removed from (resp. added to) $\mathcal{D}_t$, and $\ominus$ and $\oplus$ denote the set difference and the union operations on multisets.
The algorithm will be illustrated on the example transactional data stream given in Figure 5.1 with $b = 8$ and $\Delta = 2$. The strongly closed itemsets for transactions 1-8 (respectively 2-9) are shown in Figure 5.1b (resp. 5.1c).
We sketch the sampling algorithm in Section 5.2.1 and describe the algorithm updating the family of $\tilde{\Delta}$-closed itemsets from $\mathcal{D}_t$ to $\mathcal{D}_{t'}$ in Section 5.2.2.

5.2.1. Sampling

We use *reservoir sampling* (Knuth, 1997; Vitter, 1985) for generating a random sample $\mathcal{D}_t$ of size s for a data stream $\mathcal{S}_t = \langle T_1, \ldots, T_t \rangle$, as this method does not require the stream length to be known in advance. The general scheme of reservoir algorithms

is that they first add $T_1, \ldots, T_s$ to a "reservoir" and then throw a biased coin with probability s/k of head for all $k = s+1, \ldots, t$. If the outcome is head they replace one of the elements selected from the reservoir uniformly at random with T_k. This naive version of reservoir sampling, attributed to A.G. Waterman by D. Knuth in (Knuth, 1997), generates a random sample $\mathcal{D}_t$ of $\mathcal{S}_t$ without replacement uniformly at random. That is, all elements of $\mathcal{S}_t$ have probability s/t of being part of the sample after $\mathcal{S}_t$ has been processed. We have implemented Vitter's more sophisticated version, called Algorithm Z in (Vitter, 1985).
Given a sample $\mathcal{D}_t$ of a data stream $\mathcal{S}_t = \langle T_1, \ldots, T_t \rangle$, the sample $\mathcal{D}_{t'}$ for $\mathcal{S}_{t'} = \langle T_1, \ldots, T_t, T_{t+1}, \ldots, T_{t'} \rangle$ is computed from $\mathcal{D}_t$ by repeatedly applying Algorithm Z to $\mathcal{D}_t$ and the elements in $\langle T_{t+1}, \ldots, T_{t'} \rangle$. Recall that $t' - t \leq b$, where b is the buffer size. If a transaction in the sample is replaced by a new transaction $T \in \{T_{t+1}, \ldots, T_{t'}\}$, we appropriately update a database $\mathcal{D}_{\text{del}}$ containing the transactions to be removed from $\mathcal{D}_t$ and a database $\mathcal{D}_{\text{ins}}$ containing the transactions to be added to $\mathcal{D}_t$. Clearly, $|\mathcal{D}_{\text{del}}| = |\mathcal{D}_{\text{ins}}|$. Furthermore

$$\mathbb{E}[|\mathcal{D}_{\text{del}}|] = \mathbb{E}[|\mathcal{D}_{\text{ins}}|] \leq \frac{bs}{t'} \quad . \tag{5.2}$$

This follows directly from the linearity of the expectation and from $\mathbb{E}[X_k] = s/t'$, where X_k is the indicator random variable for the event that T_k is selected for $\mathcal{S}_{t'}$. Note that in (5.2) we have inequality only in the case that $t' - t < b$, i.e., when an update is calculated upon request for an incomplete buffer; o/w we always have equality. The RHS of (5.2) approaches 0 as t' approaches infinity. For example, it is only 15 for $b = 10\text{k}$, $t' = 100\text{M}$, $\epsilon = 0.005$, $\delta = 0.001$, and $s = 150\text{k}$, where the sample size $s = s(\epsilon, \delta)$ satisfying (5.1) is calculated by Hoeffding's inequality[2], i.e.,

$$s = \left\lceil \frac{1}{2\epsilon^2} \ln \frac{2}{\delta} \right\rceil \quad . \tag{5.3}$$

In our running example in Figure 5.1, for the sake of simplicity, we assume that transaction 1 will be replaced with transaction 9 by the sampling algorithm.

5.2.2. Incremental Update

Note that the sample size s in (5.3) depends on the error and confidence parameters ϵ and δ only. That is, s does not change with increasing data stream length. Hence, both denominators in the LHS of (2.1) (page 27) will be fixed (i.e., s) for the entire mining process from the s-th transaction onward. More precisely, for any transaction database $\mathcal{D}$ of size s and $\tilde{\Delta} \in [0,1]$, the family of *relatively* $\tilde{\Delta}$-closed itemsets of $\mathcal{D}$ is equal to the family $\mathcal{C}_{\Delta,\mathcal{D}}$ of *absolutely* Δ-closed itemsets for $\Delta = \lceil s\tilde{\Delta} \rceil$. This allows us to consider the following problem equivalent to the $\tilde{\Delta}$-Closed Set Listing problem (Problem 4):

[2] We note that Hoeffding's inequality applies to samples without replacement as well (Hoeffding, 1963). A tighter bound can be derived from Serfling's inequality (Serfling, 1974). The improvement becomes, however, marginal with increasing data stream length.

Algorithm 5 Update $\mathcal{C}_{\Delta,\mathcal{D}_t}$

input: data sets $\mathcal{D}_{\text{del}}, \mathcal{D}_{\text{ins}}$ over I and $\Delta \in \mathbb{N}$
require: totally ordered set $(I, \leq)$, data set $\mathcal{D}_t$ over I, and $\mathcal{C}_{\Delta,\mathcal{D}_t}$
output: $\mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ for $\mathcal{D}_{t'} = \mathcal{D}_t \ominus \mathcal{D}_{\text{del}} \oplus \mathcal{D}_{\text{ins}}$

Main:
1: $\mathcal{C}_{\Delta,\mathcal{D}_{t'}} \leftarrow \{\emptyset\}$
2: ListClosed($\emptyset$, $\emptyset$, $\min I$)

ListClosed(C, N, i):
1: $X \leftarrow \{k \in I \setminus C : k \geq i\}$
2: **if** $X \neq \emptyset$ **then**
3: $e \leftarrow \min X$; $C_e \leftarrow C \cup \{e\}$
4: **if** $\mathcal{D}_{\text{del}}[C_e] = \emptyset \wedge \mathcal{D}_{\text{ins}}[C_e] = \emptyset$ **then** $C' \leftarrow$ Closure_$\alpha(C, e, \mathcal{C}_{\Delta,\mathcal{D}_t})$ //Case (α)
5: **else if** $\mathcal{D}_{\text{ins}}[C_e] = \emptyset$ **then** $C' \leftarrow$ Closure_$\beta(C, e, \mathcal{D}_{\text{del}}[C_e], \mathcal{C}_{\Delta,\mathcal{D}_t})$ //Case (β)
6: **else if** $\mathcal{D}_{\text{del}}[C_e] = \emptyset$ **then** $C' \leftarrow$ Closure_$\gamma(C, e, \mathcal{D}_{\text{ins}}[C_e], \mathcal{C}_{\Delta,\mathcal{D}_t})$ //Case (γ)
7: **else** $C' \leftarrow \sigma_{\Delta,\mathcal{D}_{t'}}(C_e)$ //Case (δ)
8: **if** $C' \cap N = \emptyset$ **then**
9: add $(C, e, N, C', \uparrow)$ to $\mathcal{C}_{\Delta,\mathcal{D}_{t'}}$
10: ListClosed($C', N, e+1$)
11: **else**
12: add $(C, e, N, C', \downarrow)$ to $\mathcal{C}_{\Delta,\mathcal{D}_{t'}}$
13: $Y \leftarrow \{k \in I \setminus C : k > e\}$
14: **if** $Y \neq \emptyset$ **then**
15: $e' \leftarrow \min Y$
16: ListClosed($C, N \cup \{e\}, e'$)

Problem 5: Listing Δ-Closed sets from data streams: *Given* $\mathcal{D}_t$, $\mathcal{D}_{\text{del}}$, $\mathcal{D}_{\text{ins}}$ for $\mathcal{S}_t$ and $\mathcal{S}_{t'}$ as defined in Section 5.2.1, an integer $\Delta > 0$, and the family $\mathcal{C}_{\Delta,\mathcal{D}_t}$ of Δ-closed itemsets of $\mathcal{D}_t$, *generate* all elements of $\mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ for $\mathcal{D}_{t'} = \mathcal{D}_t \ominus \mathcal{D}_{\text{del}} \oplus \mathcal{D}_{\text{ins}}$.

Instead of generating $\mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ from scratch, our goal is to design a much faster *practical* algorithm by reducing the number of evaluations of the closure operator for $\mathcal{D}_{t'}$. This is motivated by the fact that the execution of the closure operator is the most expensive part of the algorithm. We make use of the fact that the expected number of changes in $\mathcal{D}_{t'}$ w.r.t. $\mathcal{D}_t$ becomes smaller and smaller as t' increases (cf. (5.2)). Accordingly, our focus in the design of the updating algorithm is on *quickly* deciding whether an element $C' \in \mathcal{C}_{\Delta,\mathcal{D}_t}$ remains Δ-closed in $\mathcal{D}_{t'}$, where C' is obtained by $C' = \sigma_{\Delta,\mathcal{D}_t}(C \cup \{e\})$ for some $C \in \mathcal{C}_{\Delta,\mathcal{D}_t}$ and $e \in I$. Below, we show that in all of the cases when at least one of the support sets $\mathcal{D}_{\text{del}}[C \cup \{e\}]$ or $\mathcal{D}_{\text{ins}}[C \cup \{e\}]$ is empty, the problem above can be decided much faster than with the naive way of using Algorithm 1. As we empirically demonstrate in Section 5.3.6, a considerable speed-up over the naive algorithm can be achieved in this way.

We first briefly sketch the algorithm computing $\mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ from $\mathcal{C}_{\Delta,\mathcal{D}_t}$ (see Algorithm 5). It requires four auxiliary pieces of information for all strongly closed itemsets in $\mathcal{C}_{\Delta,\mathcal{D}_t}$, except for the empty set (cf. line 1 of Main). Hence, to simplify the notation, the set variables $\mathcal{C}_{\Delta,\mathcal{D}_t}$ and $\mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ in all algorithms of this section store quintuples, where the first component is the strongly closed itemset itself; the other four components are specified below. Algorithm 5 is a divide and conquer algorithm that recursively calls ListClosed with some Δ-closed set $C \in \mathcal{C}_{\Delta,\mathcal{D}_{t'}}$, forbidden set $N \subseteq I$, and minimum candidate generator element i. It first determines the next smallest generator element e (line 3) and calculates the closure $C' = \sigma_{\Delta,\mathcal{D}_{t'}}(C \cup \{e\})$ in lines 4–7; these steps are discussed in detail below. We store C', together with some auxiliary information (lines 9 and 12). The algorithm then calls ListClosed recursively for generating further Δ-closed supersets of C'. In particular, if C' does not contain any forbidden item from N, then the last element of the quintuple stored for C' is $\uparrow$ (line 9); o/w it is $\downarrow$ (line 12). After all elements of $\mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ have been generated that are supersets of C, contain e, but do not contain any element in N, the algorithm generates all closed sets in $\mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ that are supersets of C and do not contain any element from $N \cup \{e\}$ (lines 13–16).

Example 1. *Using the transactions in Figure 5.1, we show how Algorithm 5 updates the family of 2-closed itemsets for the first eight transactions (cf. Figure 5.1b) to that for the last eight (cf. Figure 5.1c). For $I = \{a, b, c, d\}$ with $a < b < c < d$ and $\mathcal{C}_{\Delta,\mathcal{D}_t} = \{d, ad, bd, cd\}$, the input to the algorithm for this update consists of $\mathcal{D}_{del} = \{t_1\}$, $\mathcal{D}_{ins} = \{t_9\}$, and $\Delta = 2$. The algorithm first initializes $\mathcal{C}_{\Delta,\mathcal{D}_{t'}} \leftarrow \{\emptyset\}$ (line 2) and then calls ListClosed($\emptyset$, $\emptyset$, a) (line 3). The recursive calls of the function ListClosed are visualized in Figure 5.2. The edges corresponding to lines 1–12 are labeled with the value of the variable C_e (cf. line 3) and the case used for the update in lines 4–7; unlabeled edges correspond to lines 14–16.*

Theorem 2. *Algorithm 5 generates all elements of $\mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ correctly, irredundantly, in total time $O\Big(|I| \cdot |\mathcal{C}_{\Delta,\mathcal{D}_{t'}}| \cdot \|\mathcal{D}_{t'}\|_0\Big)$, with delay $O(|I|^2\|\mathcal{D}_{t'}\|_0)$, and in space $O(|I| + \|\mathcal{D}_{t'}\|_0)$.*

Proof. Regarding the correctness, we only need to show that C' computed in lines 4–7 satisfies $C' = \sigma_{\Delta,\mathcal{D}_{t'}}(C \cup \{e\})$. The correctness of Closure_α (Algorithm 6), Closure_β (Algorithm 7), and Closure_γ (Algorithm 8) is shown below in Lemmas 1, 2, and 3, respectively. The proofs of the irredundancy and the time and space complexity are immediate from Boley et al. (2010) and Gély (2005) by noting that Algorithm 5 must call the closure operator for all elements in $\mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ in the worst-case. □

In the rest of this section, we give the algorithms for the cases distinguished in lines 4–7 (case (δ) is trivial) and prove their correctness.

Case (α) We first consider the case that the set $C \cup \{e\}$ with $C \in \mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ and $e \in I$ to be extended for further Δ-closed sets satisfies

$$\mathcal{D}_{\text{del}}[C \cup \{e\}] = \emptyset \text{ and } \mathcal{D}_{\text{ins}}[C \cup \{e\}] = \emptyset \tag{5.4}$$

(line 4 of Algorithm 5). The closure $\sigma_{\Delta,\mathcal{D}_{t'}}(C \cup \{e\})$ for this case can be computed by Algorithm 6; the correctness of Algorithm 6 is stated in Lemma 1 below.

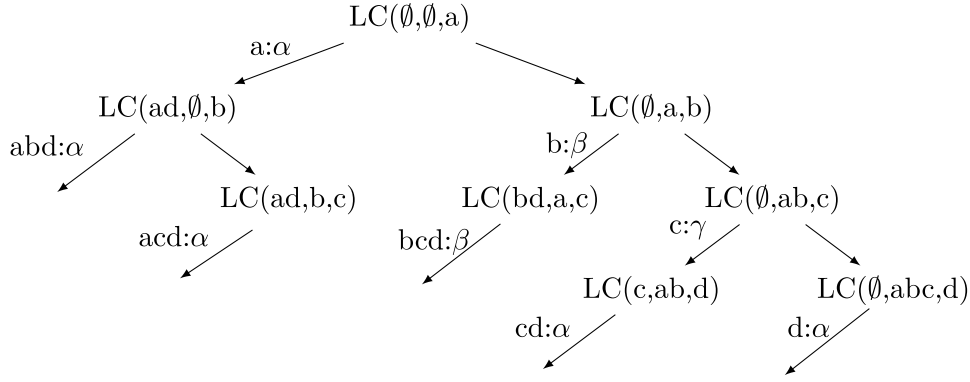


Figure 5.2.: Call stack for the update in Example 1. Labeled edges correspond to lines 1–12 of Algorithm 5; unlabeled to lines 14–16. The first component of an edge label denotes C_e (cf. line 3 of Algorithm 5), the second the case applied (cf. lines 4–7).

Algorithm 6 Closure_α

input: $C \in \mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ with $\mathcal{D}_{t'} = \mathcal{D}_t \ominus \mathcal{D}_{\text{del}} \oplus \mathcal{D}_{\text{ins}}$, $e \in I$, and $\mathcal{C}_{\Delta,\mathcal{D}_t}$
require: $\mathcal{D}_t$
output: $\sigma_{\Delta,\mathcal{D}_{t'}}(C \cup \{e\})$

1: **if** $\exists (C, e, N, C', q) \in \mathcal{C}_{\Delta,\mathcal{D}_t}$ for some N, C', and q **then return** C'
2: **else return** $\sigma_{\Delta,\mathcal{D}_t}(C \cup \{e\})$ // $\sigma_{\Delta,\mathcal{D}_t}(C \cup \{e\}) = \sigma_{\Delta,\mathcal{D}_{t'}}(C \cup \{e\})$ for this case

Lemma 1. *Algorithm 6 is correct, i.e., for all $C \in \mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ and for all $e \in I$, the output of the algorithm is $\sigma_{\Delta,\mathcal{D}_{t'}}(C \cup \{e\})$.*

Proof. Condition (5.4) implies that $\mathcal{D}_t[C \cup \{e\}] = \mathcal{D}_{t'}[C \cup \{e\}]$, where $\mathcal{D}_{t'} = \mathcal{D}_t \ominus \mathcal{D}_{\text{del}} \oplus \mathcal{D}_{\text{ins}}$. Hence, $\sigma_{\Delta,\mathcal{D}_{t'}}(C \cup \{e\}) = \sigma_{\Delta,\mathcal{D}_t}(C \cup \{e\})$ and $\sigma_{\Delta,\mathcal{D}_{t'}}(C \cup \{e\}) \in \mathcal{C}_{\Delta,\mathcal{D}_t}$, from which the proof is immediate for both cases considered in lines 1–2. □

Example 2. *In our running Example 1, the first call LC(∅,∅,a) in Figure 5.2 corresponds to case (α), as $\mathcal{D}_{ins}[a] = \mathcal{D}_{del}[a] = \emptyset$. Algorithm 6 returns ad as the closure of a in line 1, i.e., we do not need to (re)evaluate the closure operator on a.*

Case (β) We now turn to the case that $C \in \mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ and $e \in I$ fulfill

$$\mathcal{D}_{\text{del}}[C \cup \{e\}] \neq \emptyset \text{ and } \mathcal{D}_{\text{ins}}[C \cup \{e\}] = \emptyset \tag{5.5}$$

(line 5 of Algorithm 5). In Proposition 1 below we first prove some monotonicity results that will be used also for case (γ).

Proposition 1. *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be transaction databases over I. If $\mathcal{D}_1 \subseteq \mathcal{D}_2$, then for all $\Delta \in \mathbb{N}$,*

$$\mathcal{C}_{\Delta,\mathcal{D}_1} \subseteq \mathcal{C}_{\Delta,\mathcal{D}_2} \ . \tag{5.6}$$

Algorithm 7 Closure_β

input: $C \in \mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ with $\mathcal{D}_{t'} = \mathcal{D}_t \ominus \mathcal{D}_{\text{del}} \oplus \mathcal{D}_{\text{ins}}$, $e \in I$, $\mathcal{D}_{\text{del}}[C \cup \{e\}]$, and $\mathcal{C}_{\Delta,\mathcal{D}_t}$
require: $\mathcal{D}_t$
output: $\sigma_{\Delta,\mathcal{D}_{t'}}(C \cup \{e\})$

1: **if** there exists (C, e, N, C', q) in $\mathcal{C}_{\Delta,\mathcal{D}_t}$ for some N, C', and q **then**
2: $\quad C'$.count $\leftarrow |\mathcal{D}_t[C']| - |\mathcal{D}_{\text{del}}[C']|$
3: $\quad$ **for all** $i \in I \setminus C'$ **do**
4: $\quad\quad C'.\Delta_i \leftarrow |\mathcal{D}_t[C' \cup \{i\}]|$
5: $\quad\quad$ **if** C'.count $- C'.\Delta_i + |\mathcal{D}_{\text{del}}[C' \cup \{i\}]| < \Delta$ **then**
6: $\quad\quad\quad$ **return** $\sigma_{\Delta,\mathcal{D}_{t'}}(C \cup \{e\})$
7: $\quad$ **return** C'
8: **else**
9: $\quad$ **return** $\sigma_{\Delta,\mathcal{D}_{t'}}(C \cup \{e\})$

Furthermore, for all $\Delta \in \mathbb{N}$ and for all $X \subseteq I$,

$$\sigma_{\Delta,\mathcal{D}_1}(X) \supseteq \sigma_{\Delta,\mathcal{D}_2}(X) \ . \tag{5.7}$$

Proof. Let $C \in \mathcal{C}_{\Delta,\mathcal{D}_1}$ for some $\Delta \in \mathbb{N}$ and let $\mathcal{D}' = \mathcal{D}_2 \ominus \mathcal{D}_1$. Then, for any $e \in I \setminus C$, we have

$$\begin{aligned} |\mathcal{D}_2[C \cup \{e\}]| &= |\mathcal{D}_1[C \cup \{e\}]| + |D'[C \cup \{e\}]| \\ &\leq |\mathcal{D}_1[C]| - \Delta + |D'[C]| \\ &= |\mathcal{D}_2[C]| - \Delta \ , \end{aligned}$$

where the inequality follows from $C \in \mathcal{C}_{\Delta,\mathcal{D}_1}$ and from the anti-monotonicity of support sets. Hence, $C \in \mathcal{C}_{\Delta,\mathcal{D}_2}$ completing the proof of (5.6).
To show (5.7), suppose that during the calculation of $\sigma_{\Delta,\mathcal{D}_2}(X)$, the items in $\sigma_{\Delta,\mathcal{D}_2}(X) \setminus X$ have been added to X in the order $e_1, \ldots, e_k$. Let $X_0 = X$ and $X_i = X \cup \{e_1, \ldots, e_{i-1}, e_i\}$ for all $i \in [k]$. Then $|\mathcal{D}_2[X_{i-1}]| - |\mathcal{D}_2[X_i]| < \Delta$ for all $i \in [k]$ (see Algorithm 1). Since $\mathcal{D}_2[X_{i-1}] \supseteq \mathcal{D}_2[X_i]$ and $\mathcal{D}_1 \subseteq \mathcal{D}_2$, we have $|\mathcal{D}_1[X_{i-1}]| - |\mathcal{D}_1[X_i]| < \Delta$ for all i. Thus, as Algorithm 1 is Church-Rosser, all e_i will be added to $\sigma_{\Delta,\mathcal{D}_1}(X)$ as well, implying (5.7). ☐

Using Proposition 1, we have the following result for Algorithm 7 concerning case (β):

Lemma 2. *Algorithm 7 is correct, i.e., for all $C \in \mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ and for all $e \in I$, the output of the algorithm is $\sigma_{\Delta,\mathcal{D}_{t'}}(C \cup \{e\})$.*

Proof. By Condition (5.5), $\mathcal{D}_{t'}[C \cup \{e\}] \subseteq \mathcal{D}_t[C \cup \{e\}]$ and hence Proposition 1 implies that there is no $Y \in \mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ with $C \cup \{e\} \subsetneq Y \subsetneq \sigma_{\Delta,\mathcal{D}_t}(C \cup \{e\})$. Furthermore, if $\sigma_{\Delta,\mathcal{D}_t}(C \cup \{e\}) \notin \mathcal{C}_{\Delta,\mathcal{D}_{t'}}$, then $\sigma_{\Delta,\mathcal{D}_t}(C \cup \{e\}) \subsetneq \sigma_{\Delta,\mathcal{D}_{t'}}(C \cup \{e\})$. Thus, to check whether $C' = \sigma_{\Delta,\mathcal{D}_t}(C \cup \{e\})$ remains closed in $\mathcal{D}_{t'}$, it suffices to test whether

$$|\mathcal{D}_{t'}[C']| - |\mathcal{D}_{t'}[C' \cup \{i\}]| \geq \Delta \tag{5.8}$$

Algorithm 8 Closure_γ

input: $C \in \mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ with $\mathcal{D}_{t'} = \mathcal{D}_t \ominus \mathcal{D}_{\text{del}} \oplus \mathcal{D}_{\text{ins}}$, $e \in I$, $\mathcal{D}_{\text{ins}}[C \cup \{e\}]$, and $\mathcal{C}_{\Delta,\mathcal{D}_t}$
require: $\mathcal{D}_t$
output: $\sigma_{\Delta,\mathcal{D}_{t'}}(C \cup \{e\})$

```
 1: if there exists (C, e, N, C', q) in C_{Δ,D_t} for some N, C', and q then
 2:     C'' ← C ∪ {e}; D' ← (D_t ⊕ D_ins)[C'']
 3:     repeat
 4:         for all i ∈ C' \ C'' do
 5:             if |D'| − |D'[i]| < Δ then
 6:                 C'' ← C'' ∪ {i}; D' ← D'[i]
 7:     until D' has not been changed in Loop 4–6
 8:     return C''
 9: else
10:     return σ_{Δ,D_t'}(C ∪ {e})
```

further holds for all items $i \in I \setminus C'$ (lines 2–6 of Algorithm 7). If so, the algorithm returns C' in line 7, implying the correctness of Algorithm 7 for the case that $C' \in \mathcal{C}_{\Delta,\mathcal{D}_{t'}}$; the claim is trivial for the other two cases (lines 6 and 9). $\square$

We note that in our implementation of Algorithm 7 we do not calculate C'.count and $C'.\Delta_i$ in lines 2 and 4, but store and maintain them consistently. In this way, the condition in line 5 can be decided from $\mathcal{D}_{\text{del}}$, without any access to $\mathcal{D}_t$. It is important to mention that with increasing stream length, the number of elements to be deleted from $\mathcal{C}_{\Delta,\mathcal{D}_t}$ becomes smaller (cf. (5.2)) and typically, most of the elements of $\mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ are calculated by terminating in line 7.

Example 3. *In our running Example 1, the call of LC($\emptyset$,a,b) in Figure 5.2 corresponds to case (β) because $\mathcal{D}_{ins}[b] = \emptyset$ and $\mathcal{D}_{del}[b] \neq \emptyset$. Since $(\emptyset, a, b, bd, \uparrow) \in \mathcal{C}_{\Delta,\mathcal{D}_t}$, Algorithm 7 only needs to compute support queries on $\mathcal{D}_{del}$ in lines 2, 4, and 5. For all i considered in line 3, the condition in line 5 is not fulfilled. Hence, the algorithm returns db in line 7, without calling the closure operator.*

Case (γ) Finally we discuss the case that $C \in \mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ and $e \in I$ satisfy the condition

$$\mathcal{D}_{\text{del}}[C \cup \{e\}] = \emptyset \text{ and } \mathcal{D}_{\text{ins}}[C \cup \{e\}] \neq \emptyset \tag{5.9}$$

(see line 6 of Algorithm 5). The proof for this case is shown also by using Proposition 1.

Lemma 3. *Algorithm 8 is correct, i.e., for all $C \in \mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ and for all $e \in I$, the output of the algorithm is $\sigma_{\Delta,\mathcal{D}_{t'}}(C \cup \{e\})$.*

Proof. The proof is automatic for the case that the condition in line 1 of Algorithm 8 is false. Consider the case that it is true. Proposition 1 with Condition (5.9) implies that $\mathcal{C}_{\Delta,\mathcal{D}_t} \subseteq \mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ (i.e., all Δ-closed itemsets in $\mathcal{C}_{\Delta,\mathcal{D}_t}$ are preserved) and that $\sigma_{\Delta,\mathcal{D}_{t'}}(C \cup$

$\{e\}) \subseteq \sigma_{\Delta,\mathcal{D}_t}(C \cup \{e\})$. Thus, when calculating $\sigma_{\Delta,\mathcal{D}_{t'}}(C \cup \{e\})$ in Loop 3–7, it suffices to consider only the elements in $\sigma_{\Delta,\mathcal{D}_t}(C \cup \{e\}) \setminus (C \cup \{e\})$, from which the claim is immediate for this case. □

Compared to case (β), we need to calculate support counts in the entire sample $\mathcal{D}_{t'}$ for this case. However, the inner loop (lines 4–6) iterates over a typically much smaller set than the general closure algorithm (cf. lines 2–5 of Algorithm 1). Analogously to case (β), the number of new Δ-closed itemsets to be added to $\mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ becomes smaller with increasing stream length, and hence, most of the elements of $\mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ are calculated in the "then" part (line 2–8) of the "if" statement.

Example 4. *In our running Example 1, the call of LC($\emptyset$,ab,c) in Figure 5.2 corresponds to case (γ) since item c occurs only in $\mathcal{D}_{ins}$ (i.e., $\mathcal{D}_{ins}[c] \neq \emptyset$ and $\mathcal{D}_{del}[c] = \emptyset$). Since $(\emptyset, ab, c, cd, \uparrow) \in \mathcal{C}_{\Delta,\mathcal{D}_t}$, the algorithm goes into the loop 4–6, iterating over all elements of $cd \setminus c$. The condition in line 5 is not satisfied for d and thus c is returned as a new closed itemset in line 8, without calling the closure operator.*

Controlling the Time and Space Complexity of the Update

Although by Theorem 2 Algorithm 5 does not improve the worst-case time and space complexity of the batch algorithm (Boley et al., 2009b) calculating the family of strongly closed sets from scratch, our experimental results presented in Section 5.3.6 clearly demonstrate a considerable speed-up on artificial and real-world data sets. The total time depends on the cardinality of $\mathcal{C}_{\Delta,\mathcal{D}_{t'}}$, which can be exponential in $|I|$. The time and space of the update can be controlled by selecting the parameter Δ in a way that $|\mathcal{C}_{\Delta,\mathcal{D}_{t'}}| < K$ for some reasonable small K. The value of K may depend, e.g., on time or space capacity constraints. Once K has been fixed, the value of Δ can automatically be adjusted when the number of elements in $\mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ that have already been enumerated exceeds K. More precisely, suppose Algorithm 5 has generated a subset $\mathcal{C}' \subseteq \mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ with $|\mathcal{C}'| = K + 1$. For all $C \in \mathcal{C}'$, let Δ_C be the strength of C and denote $\Delta' = \min_{C \in \mathcal{C}'} \Delta_C$. Clearly, $\Delta' \geq \Delta$. Let $\mathcal{C}'' = \{C \in \mathcal{C}' : \Delta_C > \Delta'\}$. For the set obtained we have $\mathcal{C}'' \subseteq \mathcal{C}_{\Delta'+1,\mathcal{D}_{t'}}$ and $|\mathcal{C}''| \leq K$.
This change of Δ to $\Delta' + 1$ requires, however, the maintenance of auxiliary pieces of information for all already generated strongly closed sets, as well as the reconstruction of the five tuples for the closed sets remaining. More precisely, suppose $\Delta_{C,e} = |\mathcal{D}_t[C]| - |\mathcal{D}_t[C \cup \{e\}]|$ has been calculated correctly for all $C \in \mathcal{C}_{\Delta,\mathcal{D}_t}$ and for all $e \in I \setminus C$. Notice that the strength of C in $\mathcal{D}_t$ is given by $\min_{e \in I \setminus C} \Delta_{C,e}$, where the $\Delta_{C,e}$s are obtained as a byproduct of the algorithm computing the closure operator (cf. Algorithm 1). One can see that if $C \in \mathcal{C}_{\Delta,\mathcal{D}_{t'}}$ and C has not been recalculated by calling the closure operator, then $\Delta_{C,e}$ can be updated by

$$\Delta_{C,e} = \Delta_{C,e} + |\mathcal{D}_{\text{ins}}[C]| - |\mathcal{D}_{\text{ins}}[C \cup \{e\}]| - |\mathcal{D}_{\text{del}}[C]| + |\mathcal{D}_{\text{del}}[C \cup \{e\}]|$$

for all $e \in I \setminus C$. Thus, the complexity of the update for this case depends on the cardinality of $\mathcal{D}_{\text{ins}}$ and $\mathcal{D}_{\text{del}}$ only, which become smaller and smaller with increasing t' by (5.2). Finally, utilizing the algebraic properties of closure systems, the five tuples can be reconstructed by a top-down traversal of the enumeration tree corresponding to Algorithm 5.

5.2.3. Implementation Details

This section describes some aspects and optimizations in the implementation of the SCSM algorithm. The algorithm has been implemented in the Java programming language. Each new item from the data stream is mapped to the next available integer. To store numbers in collections, Java uses autoboxing, i.e., the numbers are converted to objects (Ullenboom, 2018). This results in an unnecessary memory and runtime overhead. Several frameworks have been implemented for collections of primitives, such as the "primitives collection for java"[3], "High Performance Primitive Collections"[4], and koloboke[5]. We will use the koloboke collection framework, as some preliminary experiments show that its collections are amongst the fastest.
The most expensive operation in the SCSM algorithm is the computation of the closure operator. Henceforth, the goal is to make this operation fast. The implementation follows the optimizations described in (Boley et al., 2009b) and adds a few more.
The sample is represented by a set of incidence sets for each item, i.e., for each item e we keep the set of transaction ids that contain this item. The computation of the support set of some itemset X is then obtained as the intersection of the transaction id sets for all items contained in X. We use classical data reduction to reduce the set of transaction ids to be tested. The support set of any refinement of a closed set C can be computed from the reduced transaction set $\mathcal{D}[C]$. Therefore, we reduce the sets of transaction ids of all items $e \in C \setminus I$ to the set of transaction ids of C. While this requires an additional pass over the data in the sample for each closed set, it does not change the runtime complexity of the algorithm and speeds-up the computation of all further intersections (Boley et al., 2009b).
A simple, yet powerful optimization for the computation of the closure is the addition of an early abort condition (Boley et al., 2009b). The computation of the closure of an itemset C can be stopped in Algorithm 1 whenever an element from the set of prohibited items N is added, as in this case the closure will not be part of the output (cf. line 12 of Algorithm 5). We further optimize the closure check by tracking the size of $|\mathcal{D}'| - |\mathcal{D}'[e]|$ in the case that the condition in line 4 of Algorithm 1 is not met. This allows us to skip those items e for which $|\mathcal{D}'[e]|$ is too large in successive iterations of the outer loop of the closure algorithm. For every $e \in I \setminus C$ that is added to $\mathcal{D}'$ in line 4 of Algorithm 1, we store $\Delta_e = |\mathcal{D}'| - |\mathcal{D}'[e]|$ and increment a variable, called *sum*, by Δ_e. If in the next iteration of the loop 3 – 4 the condition $\Delta_e - sum \geq \Delta$ is satisfied, then we do not need to compute the exact value of $|\mathcal{D}'| - |\mathcal{D}'[e]|$, as $|\mathcal{D}'| - |\mathcal{D}'[e]| \geq \Delta$ will hold.

[3] pcj.sourceforge.net/
[4] https://github.com/carrotsearch/hppc
[5] https://koloboke.com/

The third optimization in our closure algorithm is that it reports back all e with $|\mathcal{D}'[e]| = 0$ given C. They need not be checked in any superset of C and can thus be removed from I when calling ListClosed in line 10 of Algorithm 5.

As the final optimization, we keep track of the order in which the items e_i are added to C in line 4 of Algorithm 1 computing the closure of $\sigma_{\Delta,\mathcal{D}}(C \cup \{e\})$. Suppose that the items are added in the order $e_1, \ldots, e_k$. Then $\sigma_{\Delta,\mathcal{D}}(C \cup \{e\}) = C \cup \{e_1, \ldots, e_k\}$. When checking whether $C \cup \{e_1, \ldots, e_k\}$ is still closed in subsequent calls for the argument $C \cup \{e\}$, the items e_i can be checked in the same order. If $C \cup \{e_1, \ldots, e_k\}$ remains closed, this heuristic often outperforms a check in random order. It reduces the number of iterations of the loop in lines 2 – 5 of Algorithm 1 if the order of the items does not change. Even if it changes, we still first try to add all $e \in \{e_1, \ldots, e_k\}$ before testing $e \in I \setminus C$. Of course, once all $e \in \{e_1, \ldots, e_k\}$ have been tested, all $e \in I \setminus C \cup \{e_1, \ldots, e_k\}$ need still to be tested as well.

Finally, we note that we store the set of Δ-closed itemsets in a compact prefix tree structure, where each node represents a closed itemset or an itemset that contains a prohibited element (line 12 of Algorithm 5). The generator element e is used as the address of a child within the parent.

5.3. Empirical Evaluation

This section describes the empirical evaluation of the SCSM algorithm with the goal to demonstrate the impact of the parameters, its high approximation quality, and the speed-up obtained by the case distinction for the closure operator. The evaluation is based on artificial and real-world data streams. The artificial data streams are generated with the IBM quest market basket data generator (Agrawal and Srikant, 1994). This software generates synthetic market basket data sets based on user-defined parameters. The parameters are average transaction size (T), average length of maximal patterns (I), number of transactions (D), number of patterns (L), correlation strength between patterns (C), and the number of different items in thousands (N). The real-world data streams are obtained from data sets from the UCI repository (Dua and Graff, 2019). Table 5.1 lists their key characteristics, i.e., number of instances, number of binary attributes, average transaction size, and density. As these data sets do not contain enough transactions for long data streams, we use data streams of length 5M obtained by random enlargement of the benchmark data sets for the experiments.

For the comparisons, we define a Batch algorithm. To do so, the first t transactions from the stream are added to a buffer and then Algorithm 5 with $\mathcal{C}_{\Delta,\mathcal{D}_t} = \emptyset$ is run on these transactions. Note that this is equivalent to removing cases (α) – (γ) from Algorithm 5. Hence, we effectively measure the benefit of these cases. The result is thus correct for these transactions and constitutes a ground truth against which the SCSM algorithm is compared to in terms of precision, recall, F-score, memory, runtime, and speed-up. For all experiments, we use $\Delta = \lceil \tilde{\Delta} t \rceil$ for the batch and $\Delta = \lceil \tilde{\Delta} s \rceil$ for our streaming algorithm, where s is the sample size.

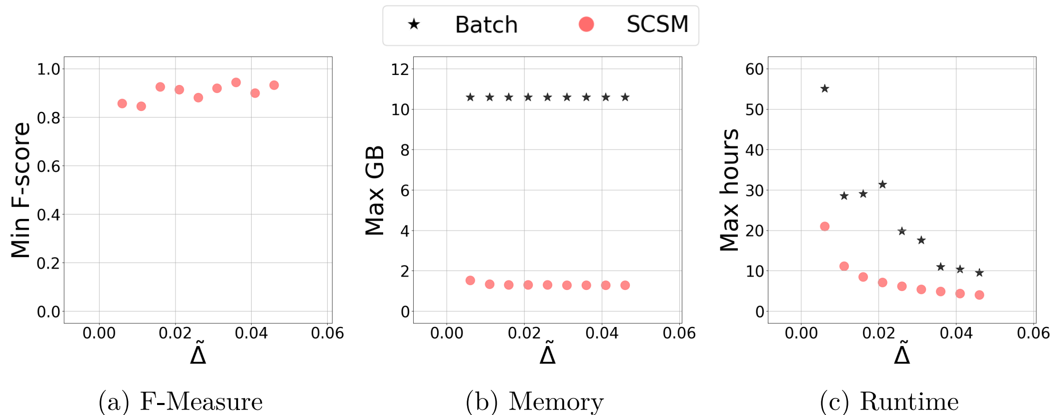
Data set	**Instances**	**Binary attributes**	**Average tx length**	**Density**
Kosarak	990,002	41,270	8	0.000196
Mushroom	8,124	119	23	0.193277
Poker-hand	1,025,010	95	11	0.115789
Retail	88,162	16,471	11	0.000626
T10I4D100k	100,000	870	10	0.011612
T40I10D100k	100,000	942	40	0.042044

Table 5.1.: Benchmark data sets used for the empirical evaluation of the SCSM algorithm.

	Confidence		
Error	0.1	0.01	0.001
0.1	149	264	380
0.01	14,978	26,491	38,004
0.001	1,497,866	2,649,158	3,800,451

Table 5.2.: Sample sizes for various error and confidence values of the SCSM algorithm.

The evaluation considers the effect of the four parameters of the algorithm, i.e., relative strength of the closure $\tilde{\Delta}$ (Section 5.3.1), error ϵ (Section 5.3.2), confidence δ (Section 5.3.3), and buffer size b (Section 5.3.4). The dimensions considered for these parameters are F-score, memory in GB, and runtime in hours. In particular, for $\tilde{\Delta}$ we consider the values $\tilde{\Delta} = 0.001 + 0.005i$ for $i = 0, 1, \ldots, 9$. For the buffer size b we compare the sizes 1k, 5k, 10k, 50k, 100k, and 500k. The confidence and error parameters δ and ϵ jointly affect the sample size. For both parameters, we consider the values 0.1, 0.01 and 0.001. The sample sizes for all nine combinations of these two parameters are given in Table 5.2.
Besides the above evaluation concerning the parameters of the algorithm, we separately investigate the mining quality measured in terms of precision and recall (Section 5.3.5) and the speed-up of SCSM compared to the Batch algorithm (Section 5.3.6).
For the first four experiments investigating the effects of $\tilde{\Delta}$, ϵ, δ, and b, we use the following setup: We take data streams with 5M transactions produced by random sampling from the data sets in Table 5.1. On these data streams, we run both the Batch algorithm and the SCSM algorithm requesting a new result at regular intervals after b transactions have been added. Except for the experiment evaluating the effect of the buffer size (i.e., b), we use $b = 500,000$. As the results in Section 5.3.4 show, the buffer size does not impact the F-measure and memory required by our streaming algorithm. It only impacts the runtime and large buffers reduce the overall processing time. The Batch algorithm runs always for the first n transactions and SCSM for the updated sample. As long as the sample is incomplete (because b is smaller than s), we simply add the transactions from the buffer to the sample and do not compute the strongly closed itemsets. We report the empirical results for a worst-case scenario, i.e., the smallest F-score, the maximum amount of memory, and the maximum runtime needed to process



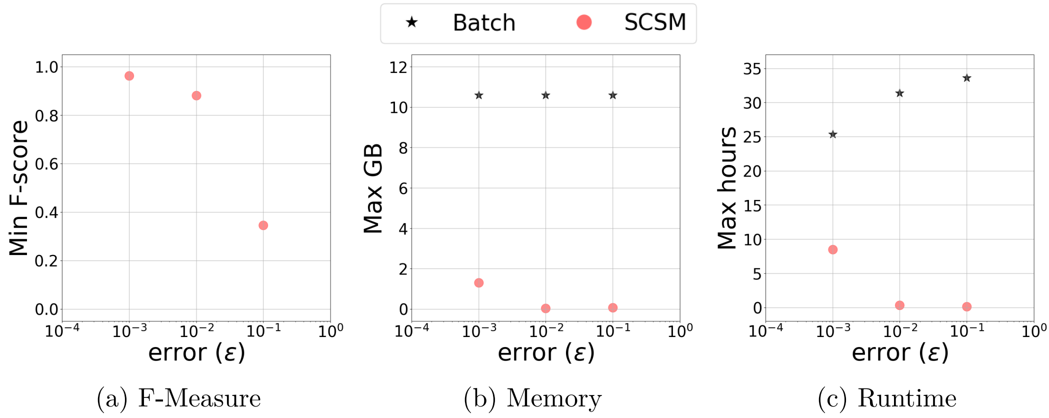
(a) F-Measure (b) Memory (c) Runtime

Figure 5.3.: Worst case effect of varying $\tilde{\Delta}$ on (a) F-Measure (showing SCSM only, because it is 1 for the Batch algorithm), (b) Memory and (c) Runtime for all data streams obtained by random enlargement of the data sets from Table 5.1 with 5M transactions.

the entire stream. We further condense the results for all data streams in the same fashion. The setup of the experiments for the mining quality and the speed-up differs from that of the first four experiments and will be described in the corresponding sections. All experiments were run on computers with Intel(R) Xeon(R) CPU E5-2650 @ 2.00GHz equipped with 64GB of memory running Debian GNU/Linux 9 with Kernel Version 4.18.10 and the OpenJDK Java version 1.8.0_181.

5.3.1. Relative Closure Strength

In this section, we investigate the effect of the choice of the relative strength of the closure $\tilde{\Delta}$ for $\epsilon < 0.1$ and for all values of δ as specified in Table 5.2. The largest value of ϵ is excluded for these experiments, as for very small samples, the approximation of the family of strongly closed itemsets is unreliable. The worst-case results for both algorithms and all steps of all data streams are shown in Figure 5.3 in terms of F-score, memory, and runtime. For $\tilde{\Delta} = 0.001$, the Batch algorithm required more than the available RAM for the data set T40I10D100K. The algorithm could therefore not complete the computation. We exclude the results for both algorithms for this case.
Figure 5.3a shows the F-score only for the SCSM algorithm because it is 1 for the Batch algorithm. The F-score decreases slightly with lower values of $\tilde{\Delta}$, as there are more strongly closed itemsets which need to be extracted from the fixed size sample. Overall, most F-scores are around 0.9; the lowest value is 0.84. These results are obtained with a fraction of the memory required by the Batch algorithm (Figure 5.3b). The parameter $\tilde{\Delta}$ shows no effect on the memory requirement for the Batch algorithm, except for the smallest value, where the experiment did not complete the execution. SCSM requires slightly more memory as $\tilde{\Delta}$ is lowered and the number of strongly closed sets increases. Despite the increase, it requires still far less memory than the Batch algorithm.



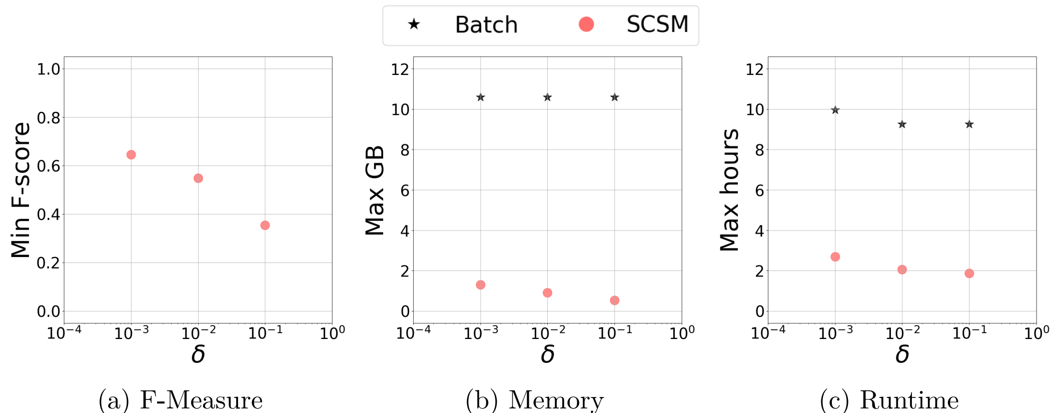
(a) F-Measure (b) Memory (c) Runtime

Figure 5.4.: Worst case effect of varying the error parameter ϵ on (a) F-Measure (showing SCSM only, because it is 1 for the BATCH algorithm), (b) Memory and (c) Runtime for all data streams obtained by random enlargement of the data sets from Table 5.1 with 5M transactions.

The effect of $\tilde{\Delta}$ on the runtime is presented in Figure 5.3c. The results show the runtime needed to process the entire data stream. One can observe that smaller values of $\tilde{\Delta}$ result in a higher runtime for both algorithms and that the SCSM algorithm is faster than the BATCH algorithm. Both results confirm our theoretical considerations. As $\tilde{\Delta}$ becomes smaller, there are more strongly closed sets. Since the complexity of our algorithm depends on the size of the output (cf. Theorem 2), the runtime increases with the number of strongly closed sets. A central goal of the streaming algorithm is to be faster than the BATCH algorithm, which is the case for all values of $\tilde{\Delta}$.

5.3.2. Error

Notice that the sample size depends both on ϵ and δ. Their effects are investigated separately. Because ϵ has a far stronger impact on the sample size than δ, we first study the effect of ϵ. The results are averaged over all values for δ. The experiments were run for $\tilde{\Delta} = 0.016 + 0.005i$ for $i = 0, 1, \ldots, 6$. The reason to start with $\tilde{\Delta} = 0.016$ instead of 0.001 is that some experiments with $\tilde{\Delta} \leq 0.011$ did not complete the computation for the BATCH algorithm. The worst-case results for both algorithms over all update steps of all data streams are shown in Figure 5.4. For smaller values of ϵ, the size of the sample increases. It is evident from Figure 5.4a that for smaller values of ϵ, the F-score increases. The same holds for the memory consumption (see Figure 5.4b). For $\epsilon = 0.001$, SCSM requires more memory than for $\epsilon = 0.1$, but overall far less than the BATCH algorithm. Both the increase in F-score and memory can be well explained by the larger sample size as a result of smaller values of ϵ.
Figure 5.4c shows the runtime required to process the entire stream. Overall, our SCSM algorithm is several times faster than the BATCH algorithm. The BATCH algorithm seemingly gets faster with smaller values of ϵ. The smaller ϵ, the larger the



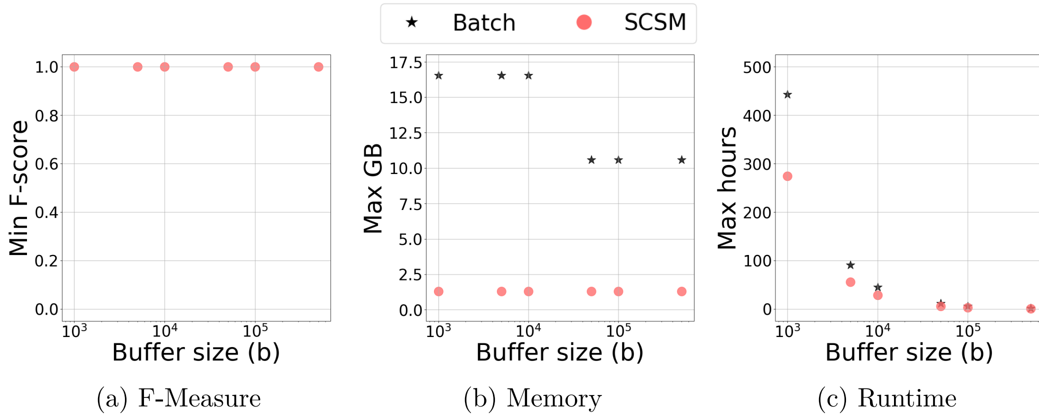
(a) F-Measure (b) Memory (c) Runtime

Figure 5.5.: Worst case effect of varying the confidence parameter δ on (a) F-Measure (showing SCSM only, because it is 1 for the BATCH algorithm), (b) Memory and (c) Runtime for all data streams obtained by random enlargement of the data sets from Table 5.1 with 5M transactions.

sample size. We note that as long as the sample of SCSM is incomplete, we run neither of the two algorithms, i.e., they run fewer times for small values of ϵ. Accordingly, the BATCH algorithm achieves an overall lower runtime. Interestingly, for very small values of ϵ the streaming algorithm gets slower. While some increase in runtime can be explained by the larger sample size, the algorithm runs a few iterations less. Its overall runtime remains far below that of the BATCH algorithm. We still have further investigated this behavior and found that the slower performance of our streaming algorithm is caused by the update steps that replace large parts of the sample. In this situation, two effects slow our algorithm down. First, most closures in Algorithm 5 will be computed with case (δ) which will result in the same runtime as the BATCH algorithm. However, there are several additional costs in this case for our algorithm. First, it needs to identify the set $\mathcal{D}_{\text{del}}$, which, in our implementation, requires one pass over the entire sample. Second, for every candidate closed set C_e, it needs to compute both the projections $\mathcal{D}_{\text{del}}[C_e]$ and $\mathcal{D}_{\text{ins}}[C_e]$. If none of the conditions for case (α) – (γ) holds, then these are simply additional costs. Of course, it would be possible to detect situations where large parts of the sample get replaced and always compute $\sigma_{\Delta,\mathcal{D}_{t'}}(C \cup \{e\})$ in this case.

5.3.3. Confidence

This section focuses on the effect of δ. The results are averaged over all values for ϵ as indicated in Table 5.2. The setup is otherwise identical to the one described in Section 5.3.2. In particular, the experiments were run for $\tilde{\Delta} = 0.016 + 0.005i$ for $i = 0, 1, \ldots, 6$. The worst-case results for both algorithms over all update steps for all data streams are shown in Figure 5.5. In particular, Figure 5.5a shows that for smaller values of δ, the F-score increases. The effect is, however, less pronounced than in case of ϵ (cf. Figure 5.4a). This is well explained by the fact that δ has a smaller impact on the



(a) F-Measure (b) Memory (c) Runtime

Figure 5.6.: Worst case effect of varying the buffer size on (a) F-Measure (showing SCSM only, because it is 1 for the BATCH algorithm), (b) Memory and (c) Runtime for kosarak data stream obtained by random enlargement with 5M transactions.

sample size, due to the logarithmic dependence (see (5.3)). Figure 5.5b shows the impact on the consumed memory. As δ becomes smaller, the memory consumption increases. This follows from the fact that a larger sample must be maintained. Again the impact is less strong than for the parameter ϵ. Overall, SCSM requires far less memory than the BATCH algorithm. Figure 5.5c shows the runtime required to process the entire stream. As δ gets smaller, the sample size increases and the streaming algorithm requires a little more time. Note that also the BATCH algorithm requires a little more time for the lowest confidence value. As we are interested in the streaming algorithm, we do not further consider this outlier. The reasons have been discussed in Section 5.3.2. For this specific experiment, the streaming algorithm is still more than three times faster than the BATCH algorithm.

5.3.4. Buffer Size

The impact of the choice of the buffer size b is investigated for $\tilde{\Delta} = 0.046$. We have chosen this value because the number of strongly closed sets is small for it. This allows the algorithm to compute the update fast. The parameters ϵ and δ are both fixed at 0.001, as these settings result in a large sample and provide the most accurate results. The worst-case results for both algorithms for this experiment over all update steps and all data streams are shown in Figure 5.6.
Figure 5.6a shows the detailed F-scores and Figure 5.6b the memory consumption. Neither the correctness of the result nor the memory required to produce the result seems to be correlated to the buffer size. SCSM reaches an F-score of 1 and requires only a fraction of the memory consumed by the BATCH algorithm. The parameter b affects only the runtime as shown in Figure 5.6c. The larger the buffer the smaller the total runtime required to process the entire stream. With larger buffers, the strongly closed sets are computed less frequently and hence the overall runtime decreases. As the family

of strongly closed itemsets is stable against small changes, it will change little without concept drifts in the data stream. This justifies the use of large buffers for many applications.

5.3.5. Mining Quality

In this section, we present empirical results demonstrating the high approximation quality of our algorithm measured in terms of precision and recall. For these experiments, we use data streams of length 5M obtained by random enlargement of the benchmark data sets listed in Table 5.1, as well as 10 artificial data streams (T10I4D5M, T40I10D5M, and 8 variations of T10I4D5M). For the two artificial data streams (T10I4D5M and T40I10D5M) we used the same parameters (except for the size) as for T10I4D100K and T40I10D100K. For the variations of T10I4D5M, we systematically modified the parameters, i.e., number of patterns (L), correlation strength between patterns (C), and the number of different items in thousands (N). In particular, we used L $\in$ {1k,10k,100k,1M}, C $\in \{0, 0.5\}$ and N $\in \{1, 10, 100\}$ in the data generation process. The patterns are independent at C $=0$, while there is some correlation among them for C $= 0.5$.

Similarly to the previous experiments, we run this experiment for the values $\tilde{\Delta} = 0.001 + 0.005i$ for $i = 0, 1, \ldots, 9$. We use $\epsilon = 0.005$ and $\delta = 0.001$ which gives us a sample size $s = 150$k (see Section 5.2.1), corresponding to around 3% of the 5M stream length. The buffer size is chosen arbitrarily to be $b = 25$k. The results in Figures 5.6a and 5.15 show that the choice of b is not critical.

The results are reported in Tables 5.3, 5.4 and 5.5 for the data sets from Table 5.1 and the variations of T10I4D5M in terms of precision (P) and recall (R), together with the number of $\tilde{\Delta}$-closed sets ($|\mathcal{C}_{\Delta,\mathcal{D}_t}|$). We note that for T40I10D5M, the Batch algorithm was unable to compute the result for $\tilde{\Delta} = 0.001$ in 24 hours. One can see that the precision and recall values are never below 0.80; in most cases, they are actually close or equal to 1. The results on the data streams obtained from the benchmark data sets might be favorable for our algorithm due to the repetition of transactions. The two artificial data streams T10I4D5M and T40I10D5M do not have such a bias. Still, we obtained very good results for these data streams as well. Thus, the repetition of transactions does not improve the results in favor of our algorithm.

We have carried out experiments on several other artificial data streams generated by the IBM Quest data generator using other parameters selected systematically (except for the size 5M). All results in Tables 5.4 and 5.5 are over a ground set of 1,000 items (i.e., for $N = 1$). For larger N, the results look similar, but with increasing N the number of strongly closed patterns decreases. The precision and recall values for the synthetic data sets are close to 1, in all considered settings they do not fall below 0.92. Thus, our algorithm provides a good approximation of the set of strongly closed itemsets in transactional data streams.

	Kosarak			Mushroom			Poker-hand			Retail			T10I4D5M			T40I10D5M		
$\tilde{\Delta}$	$\lvert\mathcal{C}_{\Delta,\mathcal{D}_t}\rvert$	P	R	$\lvert\mathcal{C}_{\Delta,\mathcal{D}_t}\rvert$	P	R	$\lvert\mathcal{C}_{\Delta,\mathcal{D}_t}\rvert$	P	R	$\lvert\mathcal{C}_{\Delta,\mathcal{D}_t}\rvert$	P	R	$\lvert\mathcal{C}_{\Delta,\mathcal{D}_t}\rvert$	P	R	$\lvert\mathcal{C}_{\Delta,\mathcal{D}_t}\rvert$	P	R
0.046	8	1	1	155	0.99	1	6	1	1	8	1	1	6	1	1	190	1	0.99
0.041	8	1	1	190	1	0.99	62	1	1	10	1	1	11	0.92	1	223	0.98	0.99
0.036	9	1	1	225	0.98	0.98	127	1	1	11	1	1	19	1	1	267	0.99	0.99
0.031	10	1	1	382	0.99	1	248	1	1	12	1	1	29	1	0.93	330	0.99	0.99
0.026	14	1	1	676	0.97	0.98	353	1	1	13	1	1	44	0.96	0.98	433	1	0.98
0.021	16	1	0.94	1112	0.99	1	578	1	1	13	0.93	1	82	0.99	0.98	650	1	0.99
0.016	24	1	1	1934	0.97	1	738	1	1	18	1	1	140	1	0.99	1137	0.98	0.98
0.011	40	0.98	1	4361	0.84	0.84	739	1	1	27	1	0.96	218	0.98	0.99	2785	0.98	0.98
0.006	86	0.98	0.97	9469	0.80	0.93	4343	0.96	0.94	66	0.98	0.98	390	0.99	1	11k	0.97	0.97
0.001	1153	0.93	0.96	76k	0.93	0.98	47k	1	1	1653	0.93	0.96	2591	0.96	0.94		—	

Table 5.3.: Number of $\tilde{\Delta}$-closed sets ($|\mathcal{C}_{\Delta,\mathcal{D}_t}|$), precision ($P$) and recall ($R$) after processing 5M transactions for various data sets and different values of $\tilde{\Delta}$.

	L = 1k						L = 10k					
	C = 0			C = 0.5			C = 0			C = 0.5		
$\tilde{\Delta}$	$\lvert\mathcal{C}_{\Delta,\mathcal{D}_t}\rvert$	P	R	$\lvert\mathcal{C}_{\Delta,\mathcal{D}_t}\rvert$	P	R	$\lvert\mathcal{C}_{\Delta,\mathcal{D}_t}\rvert$	P	R	$\lvert\mathcal{C}_{\Delta,\mathcal{D}_t}\rvert$	P	R
0.041	11	1	1	12	1	1	11	1	0.91	11	1	1
0.036	17	1	1	19	1	0.95	20	1	1	24	0.96	1
0.031	33	1	1	34	1	0.97	32	1	1	37	1	0.97
0.026	46	0.98	0.98	54	0.98	1	63	0.95	1	61	0.94	0.98
0.021	75	0.99	1	81	0.98	0.98	101	0.98	0.99	99	0.99	0.99
0.016	132	1	0.98	136	0.97	0.99	164	0.98	0.99	163	0.99	0.98
0.011	226	0.99	0.98	224	0.98	0.97	288	0.99	1	282	0.98	0.99
0.006	392	0.99	0.99	359	0.98	0.99	497	0.99	0.99	471	0.99	0.99
0.001	2618	0.96	0.95	2673	0.95	0.94	2455	0.93	0.94	2590	0.94	0.93

Table 5.4.: Number of $\tilde{\Delta}$-closed sets ($|\mathcal{C}_{\Delta,\mathcal{D}_t}|$), precision ($P$) and recall ($R$) after processing 5M transactions of synthetic quest data generated with IBMs Quest data generator. The parameters used in data generation are $T = 10$, $I = 4$, $D = 5M$, and $N = 1$, the remaining parameters are specified in the first two header rows.

5.3.6. Speed-up

In this section, we empirically study the speed-up obtained by our algorithm. For this purpose, we first sample 100k random transactions, replace then 10k, 1k, 100, 10, and 1 transaction in the sample, and run our SCSM algorithm as well as the Batch algorithm. Figure 5.7 shows the average runtime fraction of our algorithm in comparison to the Batch algorithm as a function of the number of changed transactions for all data sets from Table 5.1. The runtime results are reported in detail for one data set in Figure 5.8 by noting that we observed a similar speed-up for all other data sets. As the number of changes decreases, our streaming algorithm needs to evaluate considerably fewer database queries, implying that the smaller the change in the sample, the more the runtime of the two settings differs. In Table 5.6 we present the number of strongly closed itemsets ($|\mathcal{C}_{\Delta,\mathcal{D}_t}|$) and the speed-up ($S$) of our algorithm for various values of $\tilde{\Delta}$ for experiments when only a single transaction has been changed. In most cases, our algorithm is faster by at least one order of magnitude. Interestingly, the more $\tilde{\Delta}$-closed itemsets are calculated, the higher the speed-up. Recall that a transaction is added to the sample with probability s/k, where s is the size of the sample and k the current

	L = 100k						L = 1M1					
	C = 0			C = 0.5			C = 0			C = 0.5		
$\tilde{\Delta}$	$\lvert\mathcal{C}_{\Delta,\mathcal{D}_t}\rvert$	P	R	$\lvert\mathcal{C}_{\Delta,\mathcal{D}_t}\rvert$	P	R	$\lvert\mathcal{C}_{\Delta,\mathcal{D}_t}\rvert$	P	R	$\lvert\mathcal{C}_{\Delta,\mathcal{D}_t}\rvert$	P	R
0.041	9	0.9	1	8	1	0.88	7	0.88	1	7	1	1
0.036	21	1	0.95	20	1	1	20	1	0.9	20	1	0.95
0.031	34	0.97	0.97	33	0.97	0.97	33	0.97	1	36	1	0.97
0.026	60	0.98	1	59	0.98	1	57	1	0.98	57	0.98	1
0.021	100	1	0.97	104	0.96	0.97	100	0.97	1	100	0.95	0.99
0.016	171	0.98	1	173	0.99	0.98	175	0.98	0.98	175	1	0.99
0.011	311	0.98	0.97	317	0.99	0.98	310	1	1	314	0.98	0.99
0.006	530	0.99	1	535	0.99	0.99	534	0.99	0.99	534	0.99	0.99
0.001	3176	0.93	0.92	3153	0.93	0.93	3260	0.93	0.92	3279	0.92	0.92

Table 5.5.: Number of $\tilde{\Delta}$-closed sets ($|\mathcal{C}_{\Delta,\mathcal{D}_t}|$), precision ($P$) and recall ($R$) after processing 5M transactions of synthetic quest data generated with IBMs Quest data generator. The parameters used in data generation are $T = 10$, $I = 4$, $D = 5M$, and $N = 1$, the remaining parameters are specified in the first two header rows.

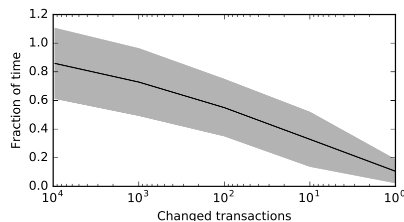


Figure 5.7.: Fraction of the runtime of our SCSM of the Batch algorithm as a function of the number of changes (log scale): black: mean, gray: SD.

#Changes	Stream time	Batch time
10,000	6.0	6.0
1,000	4.7	6.0
100	4.2	6.0
10	1.1	6.0
1	0.3	6.0

Figure 5.8.: Runtime in seconds of our SCSM and the Batch algorithm obtained for T10I4D100k for different number of changes and for $\tilde{\Delta} = 0.006$.

length of the data stream. The expected value to replace only one transaction is reached if the probability to replace each of the s transactions in the sample is at most $1/s$. This condition holds, whenever $k \geq s^2$.

5.4. Potential Applications

This section considers two practical application scenarios for the SCSM algorithm developed in this chapter. The first one is *concept drift detection*, a classical problem from data stream mining. A concept drift is an unforeseen change in the distribution of a data stream. For our example of fraud detection, such a concept drift is caused by new fraud patterns. The detection of such drifts is of high relevance for any stream mining algorithm. Indeed, a model that was learned on a data stream before a drift occurred might be incorrect after the drift. It is thus important to retrain the model after each drift. The second application scenario is computer-aided *product configuration recommendation*, a problem raised in a project with an industrial partner. The task in product

	Kosarak		Mushroom		Poker-hand		Retail		T10I4D100K		T40I10D100K	
$\tilde{\Delta}$	$\lvert\mathcal{C}_{\Delta,\mathcal{D}_t}\rvert$	S	$\lvert\mathcal{C}_{\Delta,\mathcal{D}_t}\rvert$	S	$\lvert\mathcal{C}_{\Delta,\mathcal{D}_t}\rvert$	S	$\lvert\mathcal{C}_{\Delta,\mathcal{D}_t}\rvert$	S	$\lvert\mathcal{C}_{\Delta,\mathcal{D}_t}\rvert$	S	$\lvert\mathcal{C}_{\Delta,\mathcal{D}_t}\rvert$	S
0.046	8	3.09	154	10.90	62	8.26	8	5.69	6	7.69	191	16.57
0.041	8	5.81	186	13.74	62	11.55	10	2.89	11	8.00	222	16.52
0.036	9	5.48	245	5.89	127	11.33	11	8.05	19	11.48	267	18.73
0.031	10	7.20	385	7.24	247	14.74	12	3.67	29	9.89	330	14.37
0.026	14	7.36	547	18.04	353	13.17	13	7.69	44	26.15	433	22.79
0.021	15	7.31	1105	19.44	578	19.45	13	9.28	83	24.57	649	23.40
0.016	24	11.13	2012	23.44	738	36.43	18	11.00	138	16.77	1146	43.30
0.011	38	14.87	4367	34.73	739	26.43	26	12.58	219	21.77	2780	96.39
0.006	86	23.61	11k	33.96	4238	39.35	67	24.63	391	50.05	11k	235.86
0.001	1148	99.79	82k	37.41	46k	59.66	1638	106.19	2574	148.09	346k	1893

Table 5.6.: Number of $\tilde{\Delta}$-closed sets and speed-up (S) for changing a single transaction.

configuration recommendation is to suggest to a user an individual configuration based on some initial choices from a very large set of potential options. The goal is to come up with the configuration desired by the customer by asking her as few questions as necessary and thus to eliminate the need to consider each option individually.

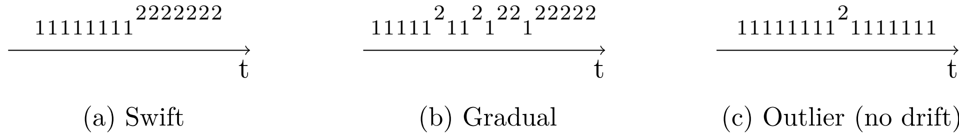
Strongly closed patterns have the characteristic feature that they correspond to patterns exhibiting a sharp drop in support count if extended by any item. For a user-defined value Δ, each strongly closed pattern occurs in at least Δ more transactions than any of its supersets. This property does not only reduce the number of patterns significantly with increasing Δ, but also guarantees that the patterns are sufficiently stable against changes in terms of support count (Boley et al., 2010). Additionally, Δ reduces the "redundancy" amongst the patterns. While all subsets of a frequent itemset are frequent themselves, subsets of Δ-closed itemsets are not necessary Δ-closed. For a fixed capacity of K itemsets, the set of Δ-closed itemsets is less redundant than the set of K frequent itemsets with high probability. This property is especially useful for applications that profit from an irredundant pattern set, such as fraud detection. As we empirically demonstrate, both *concept drift detection* and *product configuration recommendation* profit from this kind of non-redundancy.

As mentioned above, the goal of concept drift detection is to identify changes in the distribution of a data stream. Desired properties of algorithms for the detection of such drifts are: a timely detection of the true changes, few false positive detections, and limited resource consumption in terms of time and memory. For concept drift detection from transactional data streams, various pattern classes might be considered as candidates. Since the families of frequent, closed, crucial, and even maximal itemsets can be of enormous cardinality, they seem inappropriate for these tasks. Although their size can be controlled by the frequency threshold θ, such a control creates a language bias towards short patterns. However, they might be unable to capture the concept. In contrast, the size of the pattern class of strongly closed patterns can be controlled without this bias. It includes, therefore, patterns of various lengths. At the same time, they are not as numerous as ordinary closed or crucial patterns because Δ prunes very

effectively (cf. Boley et al. (2009a)). This allows faster updates of the set of patterns and requires less memory. These properties make strongly closed itemsets a good candidate for the detection of concept drifts from transactional data streams.
In product configuration recommendation the goal is to assist a customer in finding an individual configuration of a hyper-configurable system. Such systems provide a large variety of options, from which only a subset can be selected at any time. From the perspective of itemset mining, each option is an item and each sold configuration a transaction. A starting point for any such recommendation system could be frequent patterns, as they correspond to the combinations which have been chosen often. A system based purely on the descending frequency of the frequent itemsets has, however, some severe disadvantages. First, the number of patterns can be very large. Second, and more significant is the fact that there is a great redundancy among the patterns because they are closed under the subset relation. Extending a recommendation by a single item at any time is a naive solution, which does not solve the problem. Our experimental results in Section 5.4.2 show that strongly closed patterns can reduce the number of queries to a potential customer to less than half with respect to asking her for each item individually. This might make the difference between a customer buying a product and one being overwhelmed by the vast number of choices before abandoning the purchase altogether. Maximal patterns could be another alternative, but they can not be listed in output polynomial time (Boros et al., 2003), in contrast to strongly closed itemsets. Maximal patterns correspond to the maximal configurations that are frequent. A truly new configuration will differ from any existing configuration in at least one option. Thus, recommending sets of options in small chunks seems more likely to be successful than recommending the configurations from the border (see, e.g., Mannila and Toivonen (1997)) corresponding to the maximal frequent patterns. Strongly closed patterns provide starting points for such new configurations, in contrast to maximal patterns. Our experiments with different values of $\tilde{\Delta}$ clearly demonstrate that additional patterns are beneficial.
In summary, our working hypothesis is that strongly closed itemsets form a suitable synopsis for both practical applications.

5.4.1. Concept Drift Detection

This section is concerned with the application of strongly closed itemsets to concept drift detection. The detection of concept drifts is a classical problem in data stream analysis. An excellent survey on this subject has been written by Gama et al. (2014). Depending on the type of transactions in the stream, the definition of concept drifts is slightly different. We follow the one given in van Leeuwen and Siebes (2008). Consider a transactional data stream $\mathcal{S} = \mathcal{S}_1\mathcal{S}_2\mathcal{S}_3\ldots$ composed of sequences of transactions $\mathcal{S}_i$ such that $\mathcal{S}_i$ is drawn i.i.d. from distribution $\mathcal{D}_i$ such that $\mathcal{D}_i \neq \mathcal{D}_{i+1}$ for all $i > 0$ integer. In this setting the transition from the last transaction of $\mathcal{S}_i$ to the first of $\mathcal{S}_{i+1}$ constitutes a *concept drift*. If each transaction consists of a single item, statistical tests can be used to detect drifts (see, e.g., Kifer et al. (2004)). Gama et al. (2014) consider labeled transactions, where each transaction consists of descriptive variables, also called



(a) Swift (b) Gradual (c) Outlier (no drift)

Figure 5.9.: Illustration of the drift dimension pace over time for the two concepts $_1$ and ^{2}. Adapted from Gama et al. (2014).

independent variables, and a target variable also called the dependent variable. Data streams with independent and dependent variables are the primary subject of research on concept drift detection. For classification tasks, the target variable takes a finite number of categorical values, whereas, in regression, its value range is continuous. In the transactional streams considered in this thesis, the transactions are unlabeled and may contain more than one item. The focus lies on detecting drifts in this last setting, which is called a *virtual* concept drift in Gama et al. (2014).
Similarly to Gama et al. (2014), we assume that the drifts happen unexpectedly and are unpredictable. If the drifts were predictable, the problem would be trivial.
As an example, recommendation systems often rely on "typical" purchasing patterns extracted from the shopping baskets of other customers. Since these patterns are usually dynamic (i.e., change over time), such systems resorting to typical patterns must work with an up-to-date set of patterns corresponding to the (unknown) current distribution. As a second example consider topics in news feeds. They constantly change. A topic is characterized by a set of keywords that are most relevant to it. A stream of articles can be transformed into a transactional stream via keyword extraction. The keywords extracted from an article form a transaction, which might be used to identify topics and changes in the distribution of topics. A user who is interested in certain topics might be interested in reading only articles relevant to her. As new topics emerge, it might be necessary to ask her whether a new topic is relevant to her. Motivated by these and other scenarios we present an application of strongly closed itemsets to concept drift detection in transactional data streams.
Two major classes of concept drifts are distinguished in Li and Jea (2014) for frequent patterns in transactional data streams: *Isolated* and *successive* concept drifts. While isolated concept drifts are single drifts that do not necessarily have any preceding or successive drift, successive concept drifts are drifts within a drift sequence. The goal of this section is to demonstrate the suitability and effectiveness of our algorithm for the first drift type, i.e., for detecting *isolated* concept drifts.[6]
To characterize isolated concepts drifts, the following two dimensions are regarded in Li and Jea (2014):

(i) *pace*, i.e., the time required to completely replace the old distribution by the new one and

[6] Since our algorithm does not save the concept drifts detected in the past, it is not suited (in its present form) for successive concept drift detection.

(ii) *commonality*, i.e., the overlap of the two distributions.

For (i), we consider both *swift* and *gradual* replacements of distributions. In case of swift drifts, the distribution changes abruptly, i.e., from one transaction to the next. For illustration, consider a production plant. When a machine abruptly stops working, as might be the case e.g. when a drive belt breaks, then the change between the transaction when the machine was last working and the non-working state is abrupt. In contrast, gradual drifts have an elongated transition from one distribution to the other. Consider another production plant. A machine experiences wear out over time, such that for example drilled holes will be less and less precise. The transition from good holes to bad ones is very gradual as the drill slowly loses its sharpness. Figure 5.9 illustrates the two paces and an outlier, which does not constitute a drift, for transactions over the set $\{_1,^2\}$.
For (ii) above, we consider *separated* and *intersected* distributions. In case of separated distributions, there is no overlap in the distributions. For our production plant example, such a separated distribution occurs when the sensor readings for the state in which the machine is working are completely different from those when it stopped working. A straightforward way to obtain such distributions is to pick them at random from a pool over pairwise disjoint ground sets. In contrast, intersected distributions are defined over the same ground set and in a way that the individual and the joint probabilities over the ground set are identical for some of the elements and different for the others. For example, such a drift may occur in a production environment if some sensors report similar results, independent of the machines state, while others have different ranges depending on the state of the machine.
Combining (i) and (ii), we thus have four cases for isolated drifts (i.e., swift–separated, swift–intersected, gradual–separated, gradual–intersected). Gama et al. (2014) point out that most techniques "implicitly or explicitly assume and specialize in some subset of concept drifts. Many of them assume sudden non-recurring drifts. But in reality, often mixtures of many types can be observed."
The algorithms to detect drifts have been classified in Gama et al. (2014) into the four groups:

1. sequential analysis,
2. control charts,
3. monitoring two distributions, and
4. contextual.

Sequential analysis techniques use the entire stream to detect changes. They either conceptually divide a stream $\mathcal{S} = \langle T_1, \ldots, T_t \rangle$ in two successive substreams $\mathcal{S}_1^n = \langle T_1, \ldots, T_n \rangle$ and $\mathcal{S}_{n+1}^t = \langle T_{n+1}, \ldots, T_t \rangle$ and test the hypothesis that the substreams are generated by two different probability distributions (Wald, 1947) or they maintain some statistic over the stream in form of sums and signal a change if a threshold is exceeded (Page,

1954). Control charts consider labeled transactions. For such a transaction the prediction of a model can be correct or wrong. The errors in the prediction are random variables of Bernoulli trials. As long as the number of errors is less than a threshold, the stream is considered to be in-control. If the probability that the recent examples come from a different distribution is 95% or above, the stream is in a warning state. The stream is out-of-control if the recent examples came from another distribution with a probability of at least 99%. Algorithms monitoring two distributions use a reference window, which is often fixed, and a sliding window containing the most recent transactions. They compare the distributions of the transactions in the two windows with statistical tests. These approaches are more general than the previous ones and often more precise. However, they need to store the transactions of the two windows. This requires more memory than the techniques keeping only simple statistics. Finally, contextual algorithms are self-adapting to changes, rather than signaling the change to the user. This category includes, for example, the Splice-2 algorithm (Harries et al., 1998), a meta-learning technique for batch learning. This category will not be considered any further. Since methods comparing two distributions are the most general, we follow this approach to detect concept drifts with strongly closed patterns.

We omit a detailed overview of the literature on concept drift detection, as our primary goal is to demonstrate only the potential of strongly closed itemsets for this problem. The StreamKrimp (van Leeuwen and Siebes, 2008) seems to be the only algorithm specifically addressing the task of change detection from classical transactional data streams. It falls under the category of algorithms monitoring two distributions. Our approach based on strongly closed sets for the task of concept drift detection also relies on monitoring two distributions. For the sake of simplicity, the families of strongly closed sets will be computed for two windows and will be compared to each other. This simple approach could be improved by checking the closure of a subset of the closed sets from the first window on the new one. If the closure did not change for most of the checked itemsets, there is probably no drift. With this optimization, an algorithm specific to drift detection based on strongly closed sets can be developed. We stress that the focus in this thesis is only to demonstrate the suitability of strongly closed sets as a potential application. Hence, we do not include these optimizations.

Concept Drift Detection with Strongly Closed Sets

We briefly sketch our approach to detect concept drifts with strongly closed sets. As already mentioned, our goal is not to present a sophisticated algorithm for this problem, but rather to demonstrate how strongly closed sets can be used to detect such drifts. Our method follows the design of monitoring two distributions. The approach to detect concept drifts with strongly closed sets works as follows. Similarly to StreamKrimp (van Leeuwen and Siebes, 2008), it divides the stream into small batches and computes the set of relatively strongly closed itemsets for each batch for a user-defined value of $\tilde{\Delta}$. The sets of strongly closed patterns for two consecutive batches are compared by computing the Jaccard distance between them; the Jaccard distance between two sets A and B is

defined by

$$JD(A,B) = 1 - \frac{|A \cap B|}{|A \cup B|} \ .$$

Since the Jaccard distance is normalized, it provides an intuitive measure for the dissimilarity between two sets. We will use it as an indicator for drift, by comparing the two families of strongly closed itemsets. For a practical algorithm, the user would need to define a threshold on the Jaccard distance, which would cause the algorithm to signal a change if the threshold has been reached. We briefly mention two further aspects of interest for a practical algorithm. One concerns the choice of the parameter $\tilde{\Delta}$. Ideally, it would be somehow self-adjusting based on the characteristics of the stream. Further potential optimizations include a preliminary test to check if the set of strongly closed patterns has changed and skip the computation of a new set of strongly closed patterns in case it is unlikely that a drift occurred. However, such optimizations are beyond the scope of this thesis and left open for future work. In our empirical evaluation, we use the simple algorithm computing sets of strongly closed sets at regular intervals. The experimental results confirm that already this simple technique results in reasonable detection performance.

Empirical Evaluation

To asses the potential of strongly closed sets for concept drift detection under various parameter settings, we generate artificial data sets of swift and gradual replacements from the data sets in Table 5.1 (page 85) by repeatedly drawing transactions from the data sets as follows: We first create two data streams S_1 and S_2 generated with different distributions. For swift drifts, we then simply concatenate S_1 and S_2. For gradual replacements, we generate a data stream $S_1 \cdot S \cdot S_2$ by concatenating S_1, S, and S_2, where S consists of ℓ transactions from S_1 and S_2, corresponding to the "graduality" of the drift. In particular, transaction i in S is taken from S_1 at random with probability $1 - i/\ell$ and from S_2 with probability i/ℓ. In this way, we simulate a noisy "linear" transition of length ℓ from S_1 to S_2. Clearly, the longer the transition phase the less evident is the exact location of the drift.
To generate separated distributions, we simply replaced each item by a new one. Finally, for intersected distributions, some of the items were removed from the transactions independently and uniformly at random.
For each data stream, we generated three concept drifts with 2M transactions between any two consecutive drifts; 2M is a sufficiently large length enabling a careful investigation of different features (see below) of our algorithm.
To detect the concept drifts in these data streams, we started a new instance of our mining algorithm every 100k transactions, with parameter values $\epsilon = 0.01$ and $\delta = 0.02$. These values give a sample size of around 23k (cf. Section 5.2.1), corresponding to roughly 1% of the 2M transactions between the consecutive drifts. Recall from Section 5.3.5 that we obtained very accurate results for the sample size of 150k. The results in this section obtained for the much smaller sample size of 23k also demonstrate that reliable concept drift detection is possible by means of approximate results. A practical

implication of this property is that working with smaller sample sizes allows for faster update times. As the indicator for concept drifts, we used the Jaccard distance between the families of strongly closed sets returned by the two consecutive instances of our algorithm (having a delay of 100k); the impact of non-consecutive instances is discussed at the end of this section.
We investigate both the effect of different drift characteristics as well as that of different parameter settings of our algorithm. While our aim at considering different drift characteristics is to demonstrate that strongly closed sets can indeed detect a wide range of concept drifts, the analysis of different parameters of our algorithm serves to show that it can detect drifts for various choices of the parameters. That is, the stability of strongly closed itemsets ensures that it is not sensitive to the particular choice of the parameters. In particular, we empirically analyze the effects of drift characteristics for

drift type: the four drift types defined above,

drift length: the length of gradual drifts, and

drift intersection: the probability of overlap for intersected drifts

and those of the algorithm's parameter choices for

degree of closedness: the strength of closedness (i.e., $\tilde{\Delta}$),

delay: the delay after which a new instance of our algorithm is started, and

buffer: the buffer size b of our algorithm.

In order to make our experimental results clearly comparable, we present them in detail only for the Poker-hand data set. It was selected for this purpose at random out of the six data sets considered in Table 5.1; for all other five data sets we obtained very similar results, for all six characteristics above. Unless otherwise specified, the length of gradual drifts is 250k transactions, the probability for intersected distributions is 0.5, and $b = 25$k. We justify the particular choice of these values by noting that the length of gradual drifts is longer than the sample size, a probability of 0.5 results in a clear contrast between the distributions, and the buffer size was chosen at random close to the sample size.
For these experiments, we plot the Jaccard distance as a function of the transactions in the stream. These plots have two advantages over a numerical evaluation with performance measures: First, they provide more insights than pure numbers and second, their outcome does not depend on the choice of some threshold. In particular, the results can be inspected for arbitrary thresholds afterwards.

Experimental Results

This section contains the results of the experiments to demonstrate the suitability of strongly closed sets for concept drift detection for various types of drifts and different parameter settings. The six different aspects are presented in turn. The first three results are for different characteristics of the drift, the last three investigate the effect of different parameters of the algorithm.

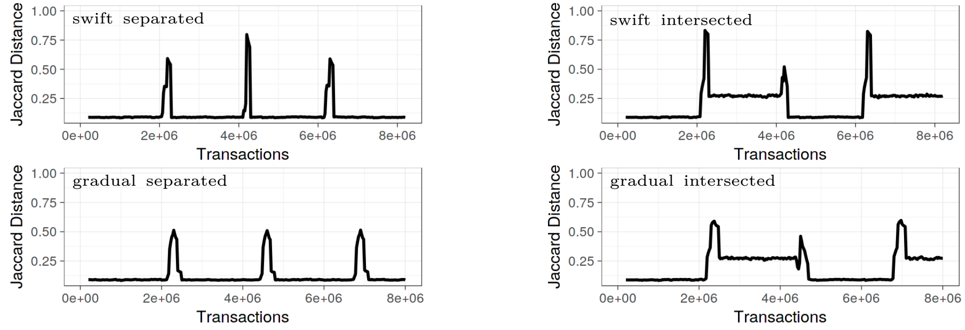


Figure 5.10.: Concept drift detection results for Poker-hand drift type $\in$ {swift–separated, swift–intersected, gradual–separated, gradual–intersected} at $\tilde{\Delta} = 0.001$.

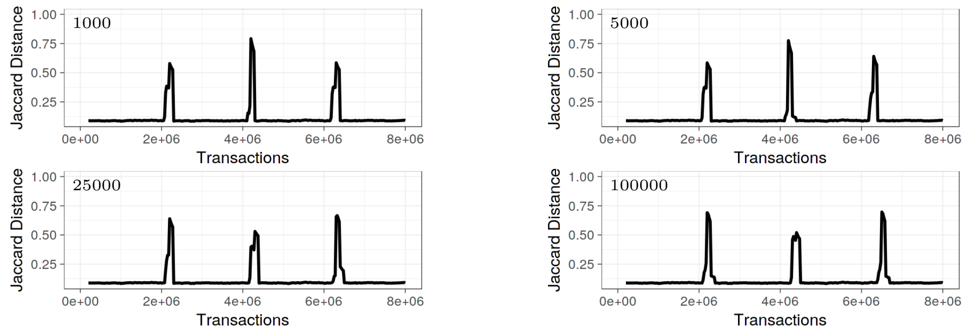


Figure 5.11.: The influence of the drift length $\in$ {1k, 5k, 25k, 100k} on concept drift detection. Results for Poker-hand with gradual–separated concept drifts at $\tilde{\Delta} = 0.001$. Note that drifts of length 0 correspond to the swift–separated case.

drift type: Figure 5.10 presents the results obtained by our algorithm for the four types of isolated drifts. The three drifts are clearly identifiable in all four cases. Notice that for the two swift drifts, the peaks are spikier than for the two gradual ones. This is due to the reason that in case of swift drifts, the transition from one distribution to the next is more abrupt compared to gradual drifts that spread over more transactions. Comparing the two separated drifts (LHS) with the two intersected ones (RHS), one can observe that the peaks stand more apart for separated drifts. This meets our expectations, as for separated drifts, there is (much) less overlap in the data distributions than for intersected ones.

drift length: Figure 5.11 is concerned with the influence of the drift's length for gradual–separated concept drifts. That is, we are interested in the ability to detect drifts for different times needed to completely replace a new concept with the previous one. We present our results for drift lengths of $1,000$, $5,000$, $25,000$, and $100,000$ transactions. The results clearly demonstrate that no matter how long the drift's length, as all three drifts are clearly identifiable for all four lengths. In particular, drifts of $1,000$ and $5,000$ transactions are clearly shorter, drifts of $25,000$ transac-

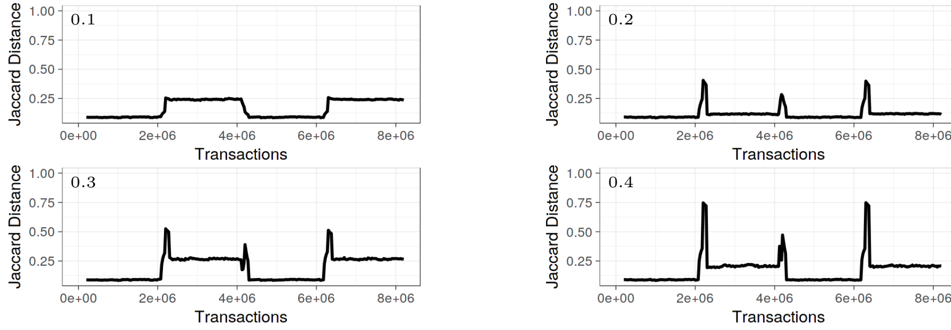


Figure 5.12.: Concept drift detection results for Poker-hand with swift–intersected concept drifts for probability of intersection $\in \{0.1, 0.2, 0.3, 0.4\}$ and $\tilde{\Delta} = 0.01$.

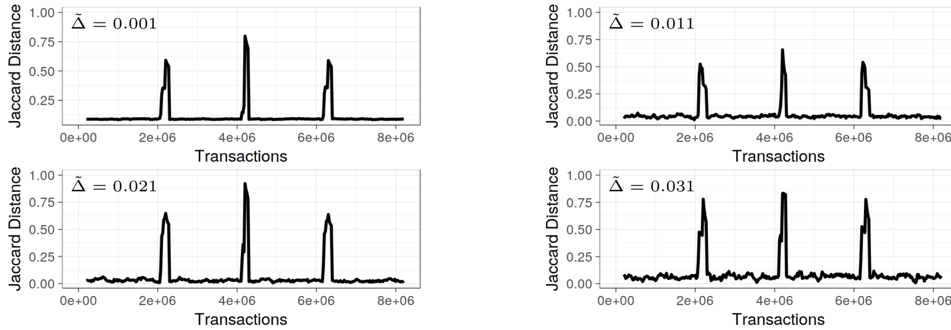


Figure 5.13.: Concept drift detection results for Poker-hand with swift–separated concept drifts for $\tilde{\Delta} \in \{0.001, 0.011, 0.021, 0.031\}$.

tions are a bit longer, and drifts of 100,000 transactions are clearly longer than the sample size. The results obtained for intersected drifts are similar. (Swift concept drifts are not presented, as their drift length is always 0.)

drift intersection: In all other experiments, intersected drifts are generated by taking the transactions one-by-one and removing each item from the transaction at hand independently and with probability $p = 0.5$. It is natural to ask how sensitive is our algorithm for other values of p. To answer this question, we generated intersected drifts for $p = 0.1, 0.2, 0.3$, and 0.4. The results are presented in Figure 5.12. One can see that the drifts are clearly recognizable for all cases, i.e., even for $p = 0.1$, although there is no peak for this value (in contrast to the three other values). The figure shows a clear correlation between p and the height of the peaks. The results for gradual drifts look very similar with slightly wider peaks (cf. Figure 5.10 for the difference between swift and gradual).[7]

[7] Note that a similar experiment is meaningless for separated drifts because they do not share any common items before and after the drift.

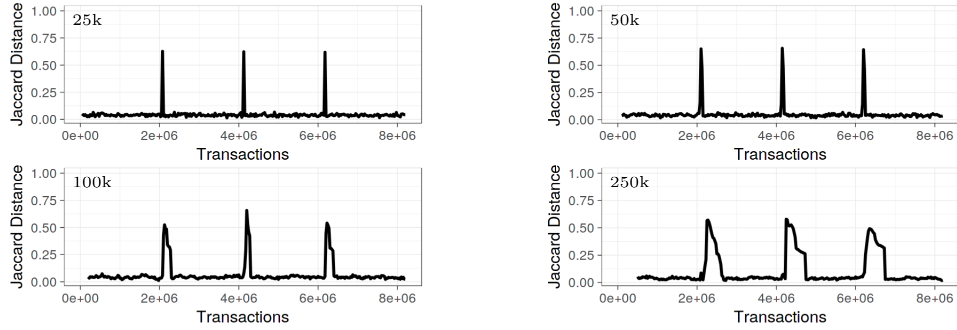


Figure 5.14.: Influence of the delay between miners $\in \{25k, 50k, 100k, 250k\}$ on concept drift detection illustrated for the Poker-hand data set with swift–separated drift and stongly closed itemsets for $\tilde{\Delta} = 0.011$.

degree of closedness: In Figure 5.13 we investigate the influence of $\tilde{\Delta}$ ranging from 0.001 to 0.031, corresponding to $\Delta = 23$ and $\Delta = 714$, respectively. The upper limit 0.031 is chosen based on the values in Table 5.6. In case of Poker-hand for instance, this choice of $\tilde{\Delta}$ results in around 250 (i.e., about 0.5%) strongly closed itemsets out of 46,000 ordinary ones. For all values of $\tilde{\Delta} \in \{0.001, 0.011, 0.021, 0.031\}$ the drifts are clearly visible. While they are smoother and more indicative for lower values of $\tilde{\Delta}$ (i.e., for larger subsets of ordinary closed itemsets), already as few as 250 strongly closed itemsets ($\tilde{\Delta} = 0.031$) suffice to detect the drifts, demonstrating the appropriateness of strongly closed itemsets to concept drift detection.

delay: In Figure 5.14 we investigate the effect of the delay, i.e., the number of transactions after which we start a new instance of our mining algorithm. Recall that the Jaccard distance is computed for the output of two consecutively started instances of the algorithm. Clearly, there is some trade-off in choosing the number of transactions between the two miners. On the one hand, the more transactions are processed before the next instance of the algorithm is started, the lower the overall runtime. On the other hand, a concept drift can only be located within the interval of transactions between two consecutively started miners. We investigate delays of 25k, 50k, 100k, and 250k transactions. Four observations can be made as the delay increases from 25k to 250k. First, for small delays, the drifts are detected on the spot, i.e., right when they happen. Second, the peaks, which are very spiky for a delay of 25k transactions become wider for larger delays. Third, the height of the peaks decreases with increasing delay. Forth, as there is more delay between the miners, it takes more transactions after the drift, before it is detected (i.e., the first peak moved from 2.075M for a delay of 25k to 2.25M for a delay of 250k). Still, the drifts are clearly visible in all cases, regardless of the choice of this parameter.

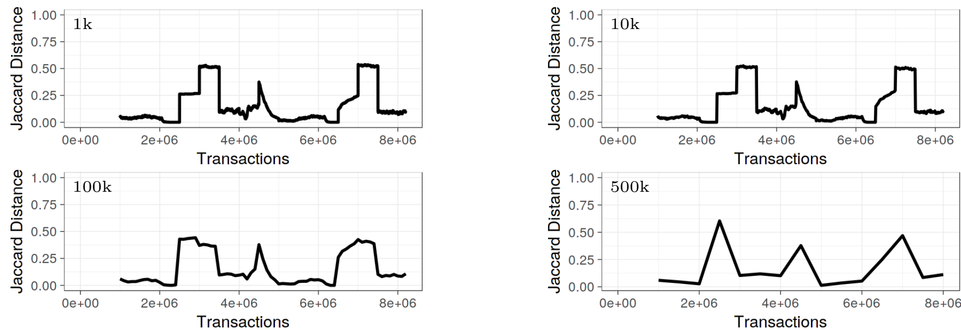


Figure 5.15.: The effect of the buffer size $\in \{1k, 10k, 100k, 500k\}$ on the concept drift detection with strongly closed sets. Results for poker-hand with swift–intersected drifts and $\tilde{\Delta} = 0.011$.

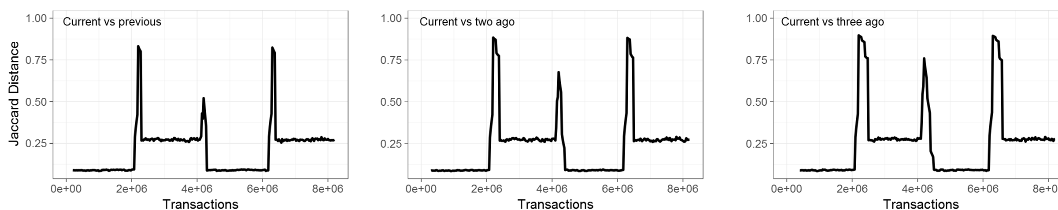


Figure 5.16.: Concept drift detection results for Poker-hand with swift–intersected concept drifts for varying detection delay at $\tilde{\Delta} = 0.001$.

buffer: The effect of the buffer size is shown in Figure 5.15. In particular, we consider buffers of size 1k, 10k, 100k, and 500k. On the one hand, with a larger buffer, there are less frequent updates of the family of strongly closed sets, resulting in a better runtime. On the other hand, however, less frequent updates of strongly closed sets reduce the ability to detect drifts close to the time they happen. Our experiments clearly confirm this trade-off. The larger the buffer size, the smoother the plot of the Jaccard distance. (The reason that there is no apparent difference between a buffer of size 1k and 10k is that both are smaller than the sample size.) For buffers of 100k and especially of 500k, the peaks lose their individual shape but are still clearly visible. In summary, smaller buffers can capture more details of the data stream, while larger buffers result in shorter computation time, as expected.

In all of the experiments so far, we considered Jaccard distances between the outputs of consecutive miners started at equidistant intervals. Even for swift drifts, the drift takes place with high probability during the run of one of the miners. In such cases, the output of the miner might thus not fully reflect the state prior to the drift but capture already a part of the drift. This is especially true for long intersected drifts. It can thus be favorable to allow for some gap between the outputs of the miners. Figure 5.16 shows the influence of the gap between the two miners. In particular, we compare the current miner with the previously started one (left), with the one started two intervals before

(middle), and with the one started three intervals before (right). The three drifts are clearly visible in all settings. With an increasing gap between the miners, the Jaccard distance reaches higher values in case of drifts, confirming our expectations.
In summary, drift type (Figure 5.10) and intersection (Figure 5.12) are two drift characteristics having the strongest influence on the Jaccard distance and thus the ability to detect concept drifts by using strongly closed itemsets. In contrast, the particular choice of the parameters of our algorithm seems to show less effect, indicating that our algorithm is not sensitive to them. In particular, for appropriately chosen delays (Figure 5.14) between miners, drifts are detected on the spot. The parameters $\tilde{\Delta}$ (Figure 5.13) and b (Figure 5.15) show the effect that drifts are a little more prominent for lower values. The length (Figure 5.11) of the drift seems to have no effect on the ability to detect drifts. An additional gap between the miners can improve the Jaccard distance (Figure 5.16). Having investigated different drift types and the sensitivity of the parameters of our algorithm on various data sets, we finally conclude that strongly closed itemsets are excellent indicators for concept drift detection in data streams.

5.4.2. Product Configuration Recommendation

As another potential practical application of strongly closed itemsets, in this section we empirically demonstrate their suitability for computer-aided *product configuration* (Falkner et al., 2011; Ricci et al., 2003), a problem raised by an industrial project. In particular, we propose an algorithm based on strongly closed itemsets that supports the customer in selecting a set of options (items) from a given pool that together constitute her desired product to be purchased (e.g., an individual composition of a pizza's topping, an individually customized prefabricated house/modular home, etc.). Depending on the number of possible options, finding the most appropriate configuration can be a time-consuming and tedious task.
The above kind of configuration problems can be regarded as the following computational problem: Suppose the goal is to identify a product, i.e., an *unknown* transaction $T \subseteq I$ for some finite set I of items. To achieve this goal, the learning algorithm is assumed to have access to a database $\mathcal{D}$ of transactions over I (e.g., pizza toppings, prefabricated houses/modular homes etc. ordered by other customers) and to an *oracle* (i.e., the customer) and it may ask *queries* of the form

$$\text{“Is } Y \subseteq T\text{?”}$$

for some $Y \subseteq I$. In case $Y = T$ (resp. $Y \subsetneq T$) the answer is “Equal” (resp. “Subset”); otherwise, the oracle returns a counterexample $x \in Y \setminus T$. The aim of the learning algorithm is to identify T with as few queries as possible.
Notice that the problem above is in fact *concept learning* with queries. Indeed, just regard I as the instance space and transactions as concepts. For the case that the transaction database $\mathcal{D}$ is not part of the problem setting, exact identification of concepts has systematically been discussed in the pioneering work by Angluin (Angluin, 1987) for various types of queries and concept classes. The query defined in this section can be considered as a combination of equivalence and subset queries (cf. Angluin (1987)). The

Algorithm 9 EXACT TRANSACTION IDENTIFICATION WITH QUERIES

input: database $\mathcal{D}$ over I and $\mathcal{C}_{\Delta,\mathcal{D}}$ for some $\Delta \in \mathbb{N}$
require: subset query oracle and an unknown set $T \subseteq I$
output: T

1: $\mathcal{S} := \mathcal{C}_{\Delta,\mathcal{D}} \cup \{\{x\} : x \in I\}$
2: $X := \emptyset$
3: $Y := \underset{Z \in \mathcal{S}}{\operatorname{argmax}} |Z \setminus X| \cdot |\mathcal{D}[Z]|$
4: call the oracle with query $X \cup Y$
5: **if** ANSWER = "EQUAL" **then return** $T = X \cup Y$
6: **else if** ANSWER = "SUBSET" **then**
7: $\quad X := X \cup Y$
8: $\quad$ remove all sets Z from $\mathcal{S}$ with $Z \subseteq X$
9: **else**
10: $\quad$ let x be the counterexample returned by the oracle
11: $\quad$ remove all sets Z from $\mathcal{S}$ with $x \in Z$
12: **go to** 3

rationale behind considering this type of queries is that most customers have typically some constraint defined in advance for the product to be purchased (e.g., an upper bound on the number of components of the pizza's topping, some fixed budget for the prefab house/modular home, etc.) that must be fulfilled by the product. Once the set of items recommended is appropriate for the customer and any further extension would violate the constraint, she might be interested in completing the process (answer "EQUAL"), without considering the remaining options (items) that have not been shown/recommended by the algorithm yet. This is an important requirement especially for such situations where the number of all options or items (i.e., $|I|$) is too large compared to that of the finally selected ones (i.e., $|T|$).

Another difference to the problem settings in Angluin (1987) is that the algorithm has access also to $\mathcal{D}$ containing a set of already purchased configurations. The underlying idea of our approach is that some of the "typical patterns" in $\mathcal{D}$ are likely to be selected also for the unknown target configuration T. It is not difficult to see that for the case that $\mathcal{D}$ is not part of the problem or the transactions in $\mathcal{D}$ have been generated with an entirely different process as the unknown set T, the number of subset queries required to identify T exactly is $|I|-1$ if all non-empty subsets of I can be a potential transaction (or concept). In real-world situations, both of these assumptions are, however, unnecessarily strong. In fact, as we show empirically using real-world product configuration data sets, the above number can be reduced to $0.5 \cdot |I|$ on average by using strongly closed itemsets.

The Algorithm

The algorithm exactly identifying an unknown transaction T over a ground set I of items with queries is given in Algorithm 9. Its input is a database $\mathcal{D}$ of transactions over I and the family $\mathcal{C}_{\Delta,\mathcal{D}}$ of Δ-closed itemsets of $\mathcal{D}$ for some positive integer Δ. Although the algorithm considers $\mathcal{D}$ and $\mathcal{C}_{\Delta,\mathcal{D}}$ as *static* inputs, it can effectively be applied in practice in the data stream setting as well. For data streams $\mathcal{D}$ and $\mathcal{C}_{\Delta,\mathcal{D}}$ are continuously updated as described in Section 5.2 based on a reservoir sample of the data stream. This follows from the properties that $\mathcal{C}_{\Delta,\mathcal{D}}$ contains typically only a few thousands of Δ-closed sets for appropriately chosen Δ and that the time complexity of the algorithm is linear in the combined size of I and $\mathcal{C}_{\Delta,\mathcal{D}}$.
Algorithm 9 starts by initializing the set variable $\mathcal{S}$ with the union of $\mathcal{C}_{\Delta,\mathcal{D}}$ and the family of singleton sets formed by the items in I (line 1). Some of the singleton sets will be needed for exact identification, e.g., in such cases when the unknown transaction T to be identified is not Δ-closed. In the set variable X, we store the set of items of T to be identified that have already been detected by the algorithm. It is initialized by the empty set (line 2). At this point, the uncertainty as to T is $|I|$ bits. The goal of the learning algorithm is to reduce this amount of uncertainty to zero. To achieve this, on the one hand, we prefer queries that reduce the amount of uncertainty with as many bits as possible, i.e., we are interested in selecting a set $Y \in \mathcal{S}$ maximizing $|Y \setminus X|$. On the other hand, however, the larger the cardinality of the query the smaller is the chance that it is a subset of T. Therefore, we need to take into account the absolute frequency (or support count) of the set inquired as well. Since cardinality is at odds with frequency, we control the trade-off between them by the product of the potential information gain with the absolute frequency and select the set Y from $\mathcal{S}$ maximizing this heuristic (cf. line 3). As we will see shortly, each set in $\mathcal{S}$ will be queried at most once. We then call the oracle with the union of the already learned subset X of T with this candidate set Y (line 4) and, depending on its answer, proceed as follows: We stop the algorithm by returning $X \cup Y$ if it is equal to T (line 5). If Y is a subset of T (line 6), we add it to X (line 7) and remove all sets from $\mathcal{S}$ that are contained by X (line 8), as none of them can further contribute to the reduction of the uncertainty as to T. Note that by definition, the set Y used in the query will also be removed. Finally, if $X \cup Y \not\subseteq T$, the oracle returns a counterexample $x \in Y \setminus T$ (line 10). As $x \notin T$, we remove all sets in $\mathcal{S}$ that contain x. Note that in this case the amount of uncertainty is reduced by 1 bit only, in contrast to the case that $X \cup Y \subsetneq T$ (lines 6–8).
We will compare the performance of our algorithm described above to the following less sophisticated algorithm, called Algorithm Baseline, obtained from Algorithm 9 by replacing line 1 with

$$1' : \mathcal{S} := \{\{x\} : x \in I\} \ .$$

That is, this algorithm ignores all Δ-closed sets and uses only singletons in the queries, preferring them by their absolute frequency. The brute-force solution to this problem would be to ask a membership query for all items in some arbitrary order. The difference between this brute-force strategy, referred to as Algorithm Naive and Algorithm Baseline is that Algorithm Baseline asks the membership queries for the items in the order

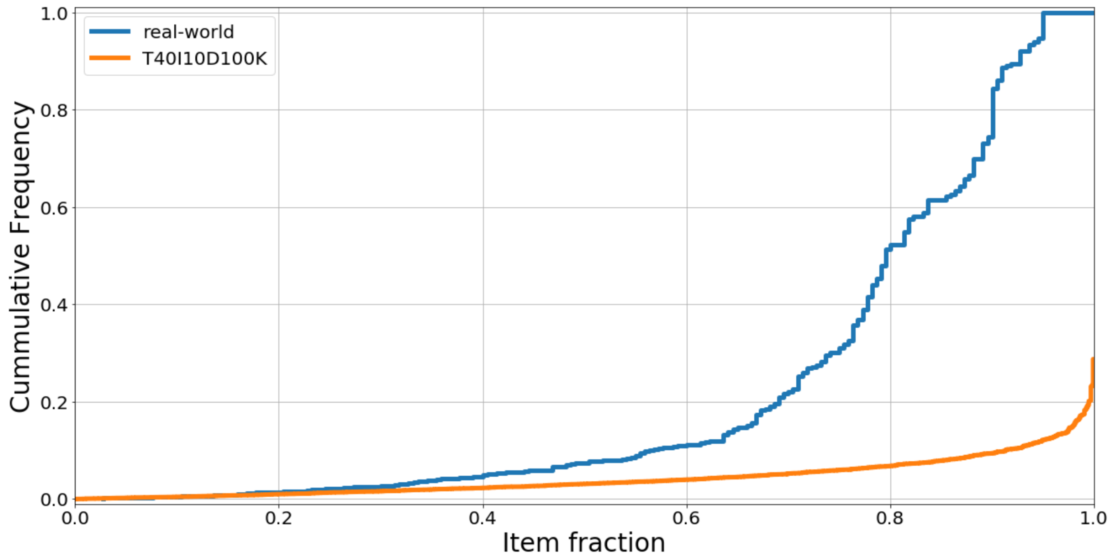


Figure 5.17.: Cumulative item frequency distribution: real-world vs. benchmark.

of their frequencies and can stop the algorithm as soon as the target transaction has been identified. One can easily see that all three algorithms are correct and require $|I|$ queries in the worst-case.[8] Below we show on real-world data sets that Algorithm 9 requires much fewer queries than Algorithm Baseline.

Real-World Data Characteristics

To understand the characteristics of the real-world industrial data, we analyze their properties and compare them to those of the classical artificial benchmark data set T10I4D100k. More precisely, we look at (i) the frequency distribution of single items, (ii) the transaction histogram, and (iii) the co-occurrences of item pairs.
First, we analyze the frequency distribution of single items for our real-world data and the benchmark data set. The cumulative frequency distributions for single items are shown for both data sets in Figure 5.17. There are some apparent differences between the real-world data and the benchmark data. The real-world data contains both very infrequent and highly frequent items. In particular, several items occur in all transactions. On the artificial benchmark data, the frequency distribution of items is more uniform. In particular, high frequent items are completely missing in the artificial benchmark data. The most frequent items in the real-world data occur in each transaction whereas the most frequent item in the artificial data occurs in only 28.7% of all transactions. Only for the least frequent 25% of all items, the relative frequency in the real-world and the artificial data set are close to each other. A consequence of this observation is that the distribution of items in the real-world data differs largely from that in the artificial

[8] Assuming that transactions are non-empty subsets of I, this worst-case bound can be reduced to $|I|-1$ by querying finally the set containing the two items left.

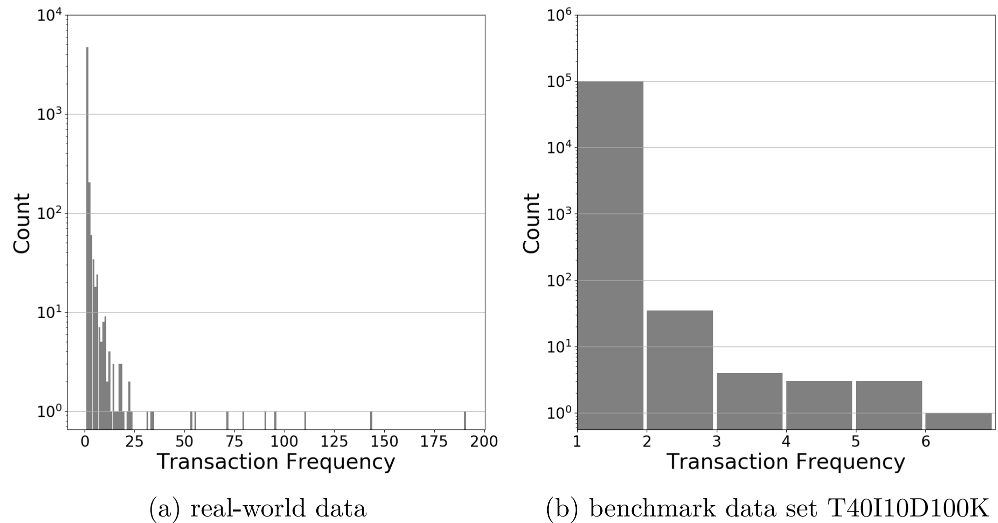
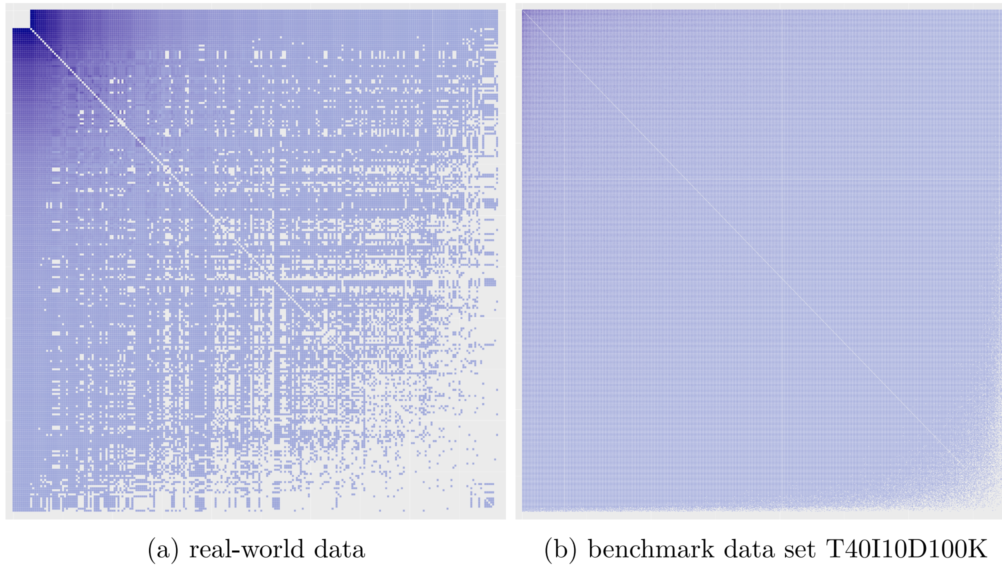


Figure 5.18.: Transaction histograms: real-world vs. benchmark.

benchmark data. This will affect the family of strongly closed patterns, as, for example, the highly frequent items from the real-world data will occur in several strongly closed patterns.
The transaction histograms in Figure 5.18 report the number of transactions as a function of the transaction frequency. We observe that in the real-world data (Figure 5.18a) 64.86% of all transactions are unique. The most frequent transaction occurs 190 times and the second most frequent 143 times. On average a transaction occurs 1.4203 times. On the benchmark data (Figure 5.18b) 99.86% of all transactions are unique. That is, there is way less similarity in the transactions compared to the real-world data. The most frequent transaction occurs only 6 times, the second most frequent 5 times. While the real-world data contains a few very frequent transactions, such transactions are missing in the artificial data. On average, a transaction occurs 1.0007 times in the benchmark data set, which is very different from the 1.4203 times average transaction occurrence on the real-world data. We observe that the item frequencies are more uniform for the synthetic data. In summary, there is a less diverse structure in the artificial data.
Finally, we investigate the co-occurrences of two items. They are shown in Figure 5.19a (real-world) and Figure 5.19b (artificial). Each plot represents a matrix with items on both axes. The cell (i, j) is colored as a function of the joint occurrence of items i and j. The darker the color, the more frequent the two items occur together.
For the real-world data we observe:

1. The first few lines in the co-occurrence plot have an almost identical color scale and show a dark color in the top left corner which is fading to the right (top of Figure 5.19a).

2. There seem to be a few highly frequent co-occurring items (top of Figure 5.19a).



(a) real-world data (b) benchmark data set T40I10D100K

Figure 5.19.: Co-ocurrences of items: real-world vs. benchmark.

3. Many combinations, in particular with low frequent items, never occur (lower right of Figure 5.19a).

On the benchmark data we note the following differences:

1. The prominent dark area in the top left corner is missing.

2. The color fading is much less evident.

3. The number of items that do not occur together is much lower than on the real-world data.

To quantify the last difference, we counted the number of co-occurring items, i.e. 2-itemsets, in the real-world and synthetic data. We observe, in the real-world data 35.91% of all possible co-occurrences actually occur versus 48.69% in the synthetic data. This is yet another indication that there is less diversity in randomly generated benchmark data sets than in the real-world data. Some important constraints, e.g., some items increase the likelihood of other items to occur or prohibit some items altogether are not well-captured in the synthetic data. However, such constraints are typical for real-world data. Consider the choice of the equipment of a bathroom as an example. A certain shower design might influence the choice of the sink, while at the same time it excludes all other showers. While there is some variation on the showers and sinks, each bathroom needs both straight and curved tubes. Thus, they will occur in any bathroom configuration. Such dependencies are missing in the artificial data sets, which show a nearly uniform distribution. Results from the real-world data are therefore not comparable to such benchmark data sets. Designing a process to generate artificial data sets that mimic the characteristics of our real-world data is an interesting open problem for future work.

ID	1	2	3	4	5	6	7	8	9	10	11
$\|I\|$	246	239	251	262	334	331	232	237	171	168	154
$\|\mathcal{D}\|$	8,341	15,844	19,310	28,239	19,550	50,134	27,078	33,933	9,149	17,935	5,902
k	52.28	55.29	51.25	43.98	59.95	60.72	48.94	67.14	48.86	47.14	51.62
density	0.2125	0.2314	0.2042	0.1679	0.1795	0.1835	0.2110	0.2833	0.2857	0.2806	0.3352

Table 5.7.: Real-world product configuration data set characteristics. The four rows correspond to the cardinality of the ground set ($|I|$), number of transactions ($|\mathcal{D}|$), average transaction size (k), and density.

Empirical Evaluation

To assess the potential of strongly closed sets for the product recommendation problem, we ran both Algorithms 9 and BASELINE on 11 *real-world* product configuration data sets from a single real-world product configuration database provided by our industrial partner. The data sets are all from the same domain but have non-overlapping ground sets I. Let I_i denote the set of all items that occur in any transaction of data set $\mathcal{D}_i$, then for every two data sets $\mathcal{D}_i$ and $\mathcal{D}_j$, $I_i \cap I_j = \emptyset$. Table 5.7 contains the cardinality of the ground set ($|I|$), the number of transactions ($|\mathcal{D}|$), the average transaction size (k), and the density for each of the 11 data sets. A final observation concerning the characteristics of these data sets is that k is very large compared to I. In other words, the density is high. The lowest density is 0.17 and the highest 0.34. For both algorithms, we measure the fraction of queries they require compared to Algorithm NAIVE. Recall that Algorithm NAIVE would ask a membership query in some ad hoc order for every element of I to identify the unknown target transaction T.
Using five-fold cross-validation, we computed the family of strongly closed itemsets for each of the data sets, used them in Algorithm 9 to identify the transactions in the test set, and calculated the fraction of queries required in comparison to Algorithm NAIVE. We did so for values of $\tilde{\Delta}$ from 0.005 to 0.1, corresponding to $504,302$ respectively 1 strongly closed sets on average for all data sets. For smaller values of $\tilde{\Delta}$, the families of strongly closed sets become very large. We ran algorithm BASELINE on the same folds and evaluated it against Algorithm NAIVE.

Experimental Results

Figure 5.20 shows the average fraction of queries over all data sets required by our Algorithm 9 for various values of $\tilde{\Delta}$ and by Algorithm BASELINE. The gain over Algorithm NAIVE is high for both algorithms. One can see that the number of queries monotonically increases with $\tilde{\Delta}$ in the observed interval, motivating the choice of small values for $\tilde{\Delta}$. This is not surprising, as smaller values of $\tilde{\Delta}$ result in larger families of strongly closed itemsets, allowing for an ultra-fine grade of queries. In particular, for $\tilde{\Delta} = 0.005$ our approach requires on average only a fraction of 0.49 of the queries needed by Algorithm NAIVE, compared to the fraction of 0.64 needed by Algorithm BASELINE. This results in a saving of 23.4% on average and 37.61% in the best case for our Algorithm 9 over Algorithm BASELINE. Note, however, that there is a trade-off between

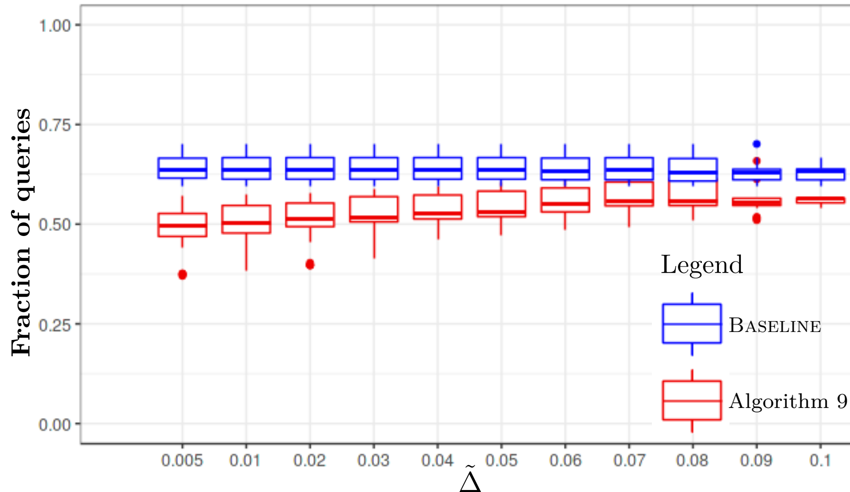


Figure 5.20.: Empirical results for the task of product configuration recommendation. We show the fraction of queries of Algorithm BASELINE and our Algorithm 9 in comparison to Algorithm NAIVE. The box-plots display the distribution of the averages over all input data sets for varying $\tilde{\Delta}$.

the choice of $\tilde{\Delta}$ and the time of updating the family of strongly closed sets. It is also very likely that the improvement will not continue forever with decreasing $\tilde{\Delta}$: As the number of closed sets increases with decreasing $\tilde{\Delta}$, they lose their characteristic sharp drop in frequency and will hence become similar to the frequent itemsets used by Algorithm BASELINE.

To illustrate the effect of $\tilde{\Delta}$ on the number of strongly closed sets, in Figure 5.21 we report the number of strongly closed sets for the values of $\tilde{\Delta}$ used in this experiment. The number of strongly closed sets increases fast with decreasing $\tilde{\Delta}$. In fact, it increases faster than the saving increases. It might thus be concluded that the contribution of each additional closed set becomes less and less, as the number of closed sets increases. Mining strongly closed patterns for lower values of $\tilde{\Delta}$ takes longer and requires more memory. The parameter $\tilde{\Delta}$ must, therefore, be chosen carefully for a specific application scenario (see Section 5.3.1 for the effect of $\tilde{\Delta}$ on both runtime and memory). The time to compute the recommendation given a family of strongly closed sets is negligible in all our experiments.

The experimental results presented in this section clearly demonstrate the potential of strongly closed itemsets for the computer-aided product configuration task.

5.5. Discussion

The goal of this chapter was the presentation of a general-purpose algorithm for mining strongly closed itemsets from transactional data streams under the landmark model. In contrast to frequent itemsets, strongly closed patterns do not have a language bias towards short patterns. Furthermore, their number can be controlled effectively by a

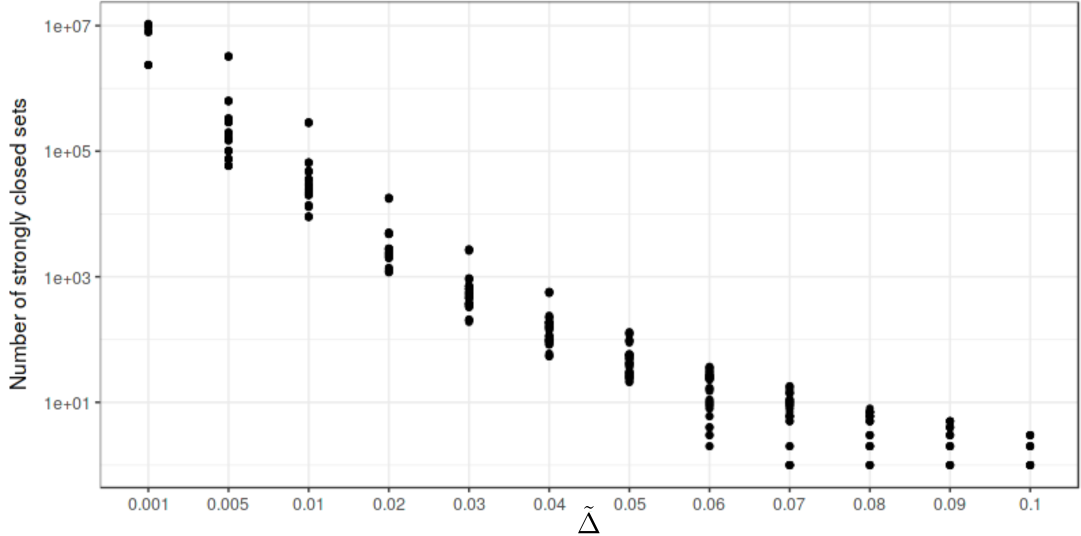


Figure 5.21.: Number of strongly closed sets for the product configuration data sets for varying $\tilde{\Delta}$.

parameter Δ that has the clear semantic interpretation that any superset must have a support count of at least Δ transactions less. These two advantageous properties motivated our choice of this particular pattern class. Our algorithm relies on reservoir sampling to obtain a fixed size data set from a growing data stream. The sample size is chosen by Hoeffding's inequality providing the following probabilistic guarantee: With probability at least $1 - \delta$, the discrepancy between the relative frequency of an itemset X in the stream $\mathcal{S}_t$ and that in the sample $\mathcal{D}_t$ is at most ϵ for all $X \subseteq I$. The fixed sample size enables our algorithm to heavily utilize some of the nice algebraic and algorithmic properties of this kind of itemsets. The careful investigation of the impact of the parameters of our algorithm presented in Section 5.3 has shown that it achieves high F-scores, requires only a small fraction of the memory required by the Batch algorithm, and is in most settings clearly faster than the Batch algorithm. The approximation and speed-up results clearly indicate the suitability of our algorithm for mining strongly closed itemsets even from *massive* transactional data streams.

In Section 5.4 we have demonstrated the suitability of strongly closed sets for two practical applications, the classical problem of concept drift detection and the task of computer aided product configuration recommendation for hyper-configurable systems.

The potential of strongly closed sets for concept drift detection has been demonstrated with a basic algorithm that computes families of strongly closed sets from a stream divided into consecutive mini-batches. The Jaccard distance was used to compare the families of strongly closed sets from two successive windows. The plots of the Jaccard distance as a function of the stream length clearly showed that a simple threshold can be used to signal a drift. The algorithm has been evaluated over diverse settings, varying both the characteristics of the data streams as well as the parameters of the algorithm. We considered data streams with different pace of drift and varying commonality for the

distributions before and after the drift. For the algorithm, the relative strength of the closure $\tilde{\Delta}$, the delay between consecutive miners, and the buffer size have been varied. In all experiments, the drifts are clearly visible in the plots of the Jaccard distance, demonstrating that strongly closed sets are a good indicator of concept drifts for a wide range of drifts and algorithmic settings. Both characteristics are important. The first to detect drifts of different types, the second to detect drifts even when the algorithmic parameters are not perfectly tuned.

The algorithm to detect concept drifts with strongly closed sets serves mainly as a proof of concept. It is obviously less sophisticated than the StreamKrimp algorithm (van Leeuwen and Siebes, 2008), which is tailored to the specific problem of concept drift detection. However, our algorithm can further be improved and turned into a sophisticated algorithm that does not need to compute a new set of strongly closed itemsets for each batch.

As a second practical application, the task of product configuration recommendation has been introduced. This problem was coined by an industrial project. Since the data is not publicly available and differs significantly from classical benchmark data sets, the data characteristics have been reported and compared to those of a classical benchmark data set. The essential difference is that the real-world data contains a much more intrinsic structure, whereas the benchmark data is rather uniform. In particular, the real-world data contains items that imply others and also items that exclude others. Strongly closed itemsets mined on historical data contain only item combinations that were sold together, which implies that they are valid solutions. To solve the problem of product configuration recommendation, an algorithm querying strongly closed sets has been proposed. It has been empirically compared to a naive and a baseline algorithm and turned out to be superior to both algorithms requiring up to 23.4% fewer user queries than the baseline algorithm, which in turn is superior to the naive algorithm. We conclude that strongly closed sets provide a clear algorithmic advantage over the other two algorithms and are indeed suitable for the problem of computer-aided product configuration recommendation. Additionally, they implicitly capture inclusion and exclusions of items that must be modeled explicitly with other machine learning approaches. This is a huge benefit, as it eliminates any manual work to define these concepts.

We close this section with with two problems for future research. The speed-up results reported in Section 5.3.6 can further be improved by utilizing that $|\mathcal{C}_{\Delta,\mathcal{D}_t}|$ is typically (much) smaller than the sample size s calculated by Hoeffding's inequality (for a detailed discussion on the size of $\mathcal{C}_{\Delta,\mathcal{D}_t}$ see Boley et al. (2009b)). In such cases, the closure $\sigma_{\Delta,\mathcal{D}_{t'}}(C \cup \{e\})$ can be computed from $\mathcal{C}_{\Delta,\mathcal{D}_t}$ without any database access to $\mathcal{D}_{t'}$, even when the closure of $C \cup \{e\}$ has not been calculated for $\mathcal{D}_t$. For example, instead of computing $\sigma_{\Delta,\mathcal{D}_t}(C \cup \{e\})$ in line 2 of Algorithm 6, we can return

$$\bigcap \{Y \in \mathcal{C}_{\Delta,\mathcal{D}_t} : C \cup \{e\} \subseteq Y\} \ ,$$

as $\mathcal{C}_{\Delta,\mathcal{D}_t}$ is a closure system.

Besides the landmark model considered in this work, the problem of mining strongly closed itemsets under the *sliding window* model would be another interesting related problem. Since our algorithm handles both insertion and deletion of transactions, it might be applicable to this setting if only the sampling is replaced by a sliding window. It might be however possible to design a more sophisticated algorithm for this setting. For practical applications, more sophisticated algorithms tailored to the specific problems would be beneficial over our proof-of-concept solutions.

5.6. Summary

In this chapter, we have presented a general-purpose algorithm, called SCSM, for the task of mining strongly closed itemset from transactional data streams. Our algorithm maintains a fixed size sample such that the relative frequency of any itemset in the sample deviates from that in the data stream at most by some small user-defined error. The size of the sample is derived from Hoeffding's bound based on a user-defined level of confidence. We have proven the formal correctness of our algorithm with respect to the sample. Our experiments confirm both the compactness of the set of strongly closed patterns and that our incremental algorithm achieves a high approximation quality in significantly less time than the Batch algorithm. Similarly to our other algorithms presented in Chapter 4, SCSM can be queried at any time. Additionally, we have demonstrated the suitability of strongly closed itemsets for two practical applications, for concept drift detection and for product configuration recommendation. All in all, our new algorithm allows mining long frequent patterns without the usual language bias introduced by frequent patterns with high accuracy from data streams.

6. Conclusion

This chapter contains a high-level summary of this dissertation thesis. It discusses the value and significance of the main results (cf. Section 1.3 for a complete list of all contributions). Finally, it shows directions for future research in a concluding section.

Summary

Our research on frequent itemset mining from data streams under the landmark model lead us to the design of new algorithms that can be queried for new patters *anytime* and still *guarantee* probabilistic error bounds. In contrast, all existing state-of-the-art algorithms with error guarantees need to process transactions in blocks of fixed size, where the block size is derived from an error bound. This enforces update intervals to be determined by the block size and limits, therefore, the flexibility. Additionally, for a fixed capacity of patterns, we can provide *longer* and *less redundant* patterns which improve the interpretability and are of high practical use. We achieve this by mining strongly closed patterns.

In the first chapter, we identified the limitation caused by static, fixed size updates of state-of-the-art algorithms. We motivated the need for more flexible designs with the exemplary use case of fraud discovery. This inspired us to consider three problems related to frequent itemset mining or subsets of them for which we can provide bounds that are independent of the update interval. Chapter 2 contained an introduction of the theoretical background of both itemset and data stream mining and the formulation of the problems considered in this thesis. Chapter 3 gave an overview of the most prominent as well as recent algorithms mining frequent itemsets from data streams for different window models. The subsequent chapters focused on the landmark model. Chapter 4 introduced two new algorithms with error guarantees for the task of frequent itemset mining from data streams under the landmark model. In contrast to most state-of-the-art approaches, the guarantees hold independently of the block size in which transactions are processed as mini-batches, or equivalently, the guarantee is independent of the update cycle. This is a great advantage for data streams with *varying* transaction arrival rate, as the results can be obtained at any point in time without the need to wait until the blocks will be filled. Extensive experiments investigated the effects of different characteristics of the data streams, compared the proposed approaches to the state-of-the-art methods, and demonstrated the strength of the new algorithms. In a subsequent chapter, the first algorithm to mine relatively strongly closed itemsets from data streams under the landmark model has been proposed. Strongly closed patterns have several advantages compared to frequent patterns. First, they represent a smaller subset of patterns and thus require less memory and are easier to understand. Second, they do not share the

language bias of frequent itemsets towards short patterns. Third, they correspond to itemsets with a sharp drop in frequency and thus form a stable subset. To exploit some of their algebraic properties, strongly closed sets are not generated from the entire stream, but instead from a reservoir sample; the sample size is derived from a probabilistic error bound. The quality and speed of our algorithm have been demonstrated empirically. Finally, Section 5.4 showed the potential of strongly closed sets for the classical problem of *concept drift detection* and the problem of *product configuration recommendation* for hyper-configurable systems. The problem of product configuration recommendation originates from an industrial project. It differs from classical recommendation systems (Ricci et al., 2011) in the sense that it recommends a configuration composed of many items, i.e., itemsets, instead of single items as in classical recommendation systems.

Discussion

The results summarized above show that the focus of the thesis lies on the *design of algorithms* and not on new data structures. In this respect, the work follows the majority of literature on pattern mining. The empirical evaluation of the properties of the algorithms is far more extensive than usual in this field. In particular, the effect of various data stream characteristics have been investigated in great detail.

Existing algorithms mining frequent itemsets with formal guarantees rely on *fixed* size update cycles. In case of very small update cycles, i.e., single transactions, these algorithms are, however, slow. In case of long update cycles, they are not flexible. The algorithms proposed in this thesis eliminate this requirement. The benefits are obvious. Consider some production plant that interrupts the production. In this scenario, classical algorithms can not provide an up-to-date result, unless by coincidence the stop happens to match the update cycle or the user is willing to sacrifice the error bound. In contrast, our algorithms can provide the most up-to-date result without any loss for future results. Assume the result is to be used to understand the cause of the interruption. In this setting, it is very important that it is reliable. This reliability can be only guaranteed by the error bound.

We point out the significance of mining strongly closed itemsets from data streams instead of classical frequent itemsets. Notably, strongly closed itemsets have several important advantages. The popular frequency-based itemset mining paradigm suffers from the language bias favoring short patterns. This is an often undesirable side effect of frequency-based pruning. Strongly closed sets do *not* have this bias. Moreover, they correspond to stable patterns with a sharp drop in frequency. This sharp frequency drop ensures that these patterns stand apart amongst all. In addition, the set of strongly closed patterns is typically much smaller and shrinks fast in size with increasing strength of the closure (Boley et al., 2009b; Trabold and Horváth, 2017). This is advantageous in terms of a smaller output. A central goal in pattern discovery is to provide *understandable* patterns (Mannila, 2002). Providing a large set of patterns, as usually generated by frequency-based pruning approaches, is a nuisance for domain experts trying to understand the entire collection of those patterns. In contrast, our choice of the smaller

pattern sets of strongly closed itemsets helps the practitioners to better understand the characteristics of their data streams. This motivates further research mining compact understandable pattern sets from data streams, which are often desired, yet hard to find. While there are very strong benefits on the application level both in terms of stability and compactness, there is a cost associated with mining such concise patterns. Computing the closure is expensive in comparison to frequency-based pruning. Thus, mining strongly closed itemsets from high-speed transactional data streams might not be an option. In such cases, algorithms mining frequent itemsets are more beneficial.

Outlook

The results of this thesis have solved some problems and at the same time raised some new questions for future research. Besides the individual questions related to each chapter that are mentioned in the respective chapter, we have identified more general questions for potential future research. As this work focused on frequent, respectively strongly closed patterns mined under the landmark model, two very natural questions to ask are: (i) How can the proposed methods be adapted for mining the top-k patterns with the same sort of probabilistic guarantee independent of the update cycle and (ii) how can the probabilistic algorithms be fit to other window models? For the case of frequent patterns, algorithms for top-k pattern mining and other window models have already been proposed. However, they all do not use the probabilistic reasoning introduced in this work and lack the ability to be queried anytime. Adapting our reasoning model to (i) and (ii) should not be too complicated. The development of additional algorithms, that provide bounds independent of the size of the update cycles might lead to another direction of future research: Is there an entire class of algorithms for which a probabilistic bound like ours could be proven in general?
While strongly closed itemsets can be enumerated with polynomial delay (Boley et al., 2009b), no efficient algorithm for the enumeration of the top-k strongly closed patterns is known so far. The naive solution building on the existing algorithm computing the closure for a very small Δ and filtering the result does not yield an efficient algorithm, as its delay would not be output-polynomial. The problem is interesting because it might be easier for an expert to define the number of patterns she is interested in, instead of a strength threshold Δ. The problem is however non-trivial. It might even have no efficient exact solution. In case of approximate solutions questions about the quality of the approximation arise. Adapting strongly closed patterns to further window models, on the other hand, should be easier. Our algorithm already handles both the addition and deletion of transactions from the sample. It suffices thus to modify the sampling mechanism, to obtain an algorithm for the sliding window case. The adaptation to the time fading window model would require an additional weight for each transaction.
We have cast the problem of mining relatively $\tilde{\Delta}$-closed sets into the problem of mining absolutely Δ-closed sets for a *fixed* size sample. For the case of absolutely Δ-closed sets it holds that the addition of a transaction can only add further Δ-closed sets to those that are already Δ-closed, but can not remove any Δ-closed set; vice versa the deletion of

a transaction can only remove Δ-closed itemsets, but can not induce new Δ-closed sets. The two properties do not, however, hold for the case of relatively $\tilde{\Delta}$-closed itemsets[1]. Hence for the enumeration of relatively $\tilde{\Delta}$-closed sets, update mechanisms different from those considered in this thesis are required. How to maintain $\tilde{\Delta}$-closed itemsets efficiently is an open question for potential future research. It is both of theoretical and practical importance, as the number of absolutely closed sets for any fixed Δ increases with additional transactions, unless restricted to a fixed size sample. Relatively strongly closed sets on the other hand would scale naturally with increasing stream length.
Other directions for future research relate to *practical applications*. We have only considered two potential application scenarios. Strongly closed patterns might be useful in further scenarios such as root-cause analysis, where they could be used to identify common characteristics as well as in many other domains, such as biology or medicine. Another theoretically interesting direction is the transfer of the concept of strong closedness to other pattern classes, for example, *sequences* (Agrawal and Srikant, 1995). Strongly closed sequences could reduce the set of sequences in a controllable and semantically meaningful way. A very similar direction would be to extend the concept of *relevant sequences* (Grosskreutz et al., 2013) to strongly relevant sequences.
Finally, future research could build upon the problem of recommending *entire* itemsets instead of single items, just like multi-label classification extends the concept of assigning a single class to an instance to the assignment of multiple classes to each instance. State-of-the-art solutions for the multi-label classification problem exploit inter-class dependencies and achieve thus better solutions than simply running several dedicated single-class classifiers. Itemset recommendation might be solvable both more efficiently and with better results if addressed directly in comparison to a combination of many single item recommender systems.
To summarize, many inspiring directions emerge from this work. Some of these are directly related to frequent pattern mining, others concern the pattern class of strongly closed itemsets, and some consider further potential applications.

[1] To illustrate this consider the following example of a data stream with the three transactions `ac`, `ab`, `ab`. Let $\tilde{\Delta} = 0.6$, then `ab` is $\tilde{\Delta}$-closed. Assume that the next transaction is `d`. This makes `ab` 0.5-closed, i.e., even though a transaction has been added, `ab` lost its state of relative closedness. If in the next step transaction `ac` will be deleted, then `ab` is 0.67-closed again.

Bibliography

Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Procedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*, pages 487–499. VLDB Endowment, 1994.

Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering (ICDE '95)*, pages 3–14. IEEE Computer Society, 1995.

Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data (SIGMOD '93)*, pages 207–216. ACM, 1993.

Salha M. Alzahrani, Naomie Salim, and Ajith Abraham. Understanding plagiarism linguistic patterns, textual features, and detection methods. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(2):133–149, 2012.

Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1987.

Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '02)*, pages 1–16. ACM, 2002.

Siddhartha Bhattacharyya, Sanjeev Jha, Kurian Tharakunnel, and J. Christopher Westland. Data mining for credit card fraud: A comparative study. *Decision Support Systems*, 50(3):602 – 613, 2011.

Mario Boley, Tamás Horváth, and Stefan Wrobel. Efficient discovery of interesting patterns based on strong closedness. In *Proceedings of the 2009 SIAM International Conference on Data Mining (SDM 2009)*, pages 1002–1013. SIAM, 2009a.

Mario Boley, Tamás Horváth, and Stefan Wrobel. Efficient discovery of interesting patterns based on strong closedness. *Statistical Analysis and Data Mining*, 2(5-6): 346–360, 2009b.

Mario Boley, Tamás Horváth, Axel Poigné, and Stefan Wrobel. Listing closed sets of strongly accessible set systems with applications to data mining. *Theoretical Computer Science*, 411(3):691–700, 2010.

Christian Borgelt. An implementation of the fp-growth algorithm. In *Proceedings of the 1st international workshop on open source data mining (OSDM '05)*, pages 1–5. ACM, 2005.

E. Boros, V. Gurvich, L. Khachiyan, and K. Makino. On maximal frequent and minimal infrequent sets in binary matrices. *Annals of Mathematics and Artificial Intelligence*, 39(3):211–221, 2003.

Joong Hyuk Chang and Won Suk Lee. Finding recent frequent itemsets adaptively over online data streams. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '03)*, pages 487–492. ACM, 2003.

James Cheng, Yiping Ke, and Wilfred Ng. A survey on algorithms for mining frequent itemsets over data streams. *Knowledge and Information Systems*, 16(1):1–27, 2008.

Yun Chi, Haixun Wang, Philip S. Yu, and Richard R. Muntz. Moment: maintaining closed frequent itemsets over a stream sliding window. In *Proceedings of the 4th IEEE International Conference on Data Mining (ICDM '04)*, pages 59–66. IEEE Computer Society, 2004.

Chuck Cranor, Yuan Gao, Theodore Johnson, Vlaidslav Shkapenyuk, and Oliver Spatscheck. Gigascope: High performance network monitoring with an sql interface. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*, pages 623–623. ACM, 2002.

Xuan Hong Dang, Wee-Keong Ng, and Kok-Leong Ong. Online mining of frequent sets in data streams with error guarantee. *Knowledge and Information Systems*, 16(2): 245–258, 2008.

Ariyam Das and Carlo Zaniolo. Fast lossless frequent itemset mining in data streams using crucial patterns. In *Proceedings of the 2016 SIAM International Conference on Data Mining (SDM '16)*, pages 576–584. SIAM, 2016.

Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows: (extended abstract). In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '02)*, pages 635–644. SIAM, 2002.

Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order, Second Edition*. Cambridge University Press, 2002.

James Davidson, Benjamin Liebald, Junning Liu, Palash Nandy, Taylor Van Vleet, Ullas Gargi, Sujoy Gupta, Yu He, Mike Lambert, Blake Livingston, and Dasarathi Sampath. The youtube video recommendation system. In *Proceedings of the 4th ACM Conference on Recommender Systems (RecSys '10)*, pages 293–296. ACM, 2010.

Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh AAAI National Conference on Artificial Intelligence*, AAAI'88, pages 49–54. AAAI Press, 1988.

L Delamaire, HAH Abdou, and J Pointon. Credit card fraud and detection techniques: a review. *Banks and Bank Systems*, 4(2):57–68, 2009.

Norman R. Draper and Harry Smith. *Applied Regression Analysis*. Wiley, 1998.

Dheeru Dua and Casey Graff. UCI machine learning repository. URL: http://archive.ics.uci.edu/ml, 2019.

Richard O. Duda, Peter E.Hart, and David G. Stork. *Pattern classification*. Wiley, 2000.

European Central Bank ECB. Fifth report on card fraud. URL: https://www.ecb.europa.eu/pub/cardfraud/html/ecb.cardfraudreport201809.en.htm, 2018.

Opher Etzion, Fabiana Fournier, Inna Skarbovsky, and Barbara von Halle. A model driven approach for event processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems (DEBS '16)*, pages 81–92. ACM, 2016.

Andreas Falkner, Alexander Felfernig, and Albert Haag. Recommendation technologies for configurable products. *AI Magazine*, 32(3):99–108, 2011.

Usama M. Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery: an overview. In *Advances in knowledge discovery and data mining*, pages 1–34. American Association for Artificial Intelligence, 1996.

Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

Philippe Fournier-Viger, Jerry Chun-Wei Lin, Bay Vo, Tin Truong Chi, Ji Zhang, and Hoai Bac Le. A survey of itemset mining. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 7(4), 2017.

Alex S. Fraser and Donald Burnell. *Computer Models in Genetics*. McGraw-Hill, 1970.

Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. Mining data streams: A review. *ACM SIGMOD Record*, 34(2):18–26, 2005.

João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM Computing Surveys*, 46 (4):44:1–44:37, 2014.

João Gama. *Knowledge Discovery from Data Streams*. Chapman & Hall, 2010.

João Gama. A survey on learning from data streams: current and future trends. *Progress in Artificial Intelligence*, 1(1):45–55, 2012.

Bernhard Ganter and Klaus Reuter. Finding all closed sets: A general approach. *Order*, 8(3):283–290, 1991.

Alain Gély. A generic algorithm for generating closed sets of a binary relation. In *Proceedings of the 3rd International Conference on Formal Concept Analysis (ICFCA '05)*, pages 223–234. Springer, 2005.

Chris Giannella, Jiawei Han, Jian Pei, Xifeng Yan, and Philip S. Yu. Mining frequent patterns in data streams at multiple time granularities. In H. Kargupta, A. Joshi, K. Sivakumar, and Y. Yesha, editors, *Next generation data mining*, pages 191–212. AAAI/MIT, 2003.

Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14, 2003.

Yihong Gong and Xin Liu. Generic text summarization using relevance measure and latent semantic analysis. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '01)*, pages 19–25. ACM, 2001.

Henrik Grosskreutz, Bastian Lang, and Daniel Trabold. A relevance criterion for sequential patterns. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD 2013)*, volume 8188 of *Lecture Notes in Computer Science*, pages 369–384. Springer, 2013.

Sudipto Guha, Nick Koudas, and Kyuseok Shim. Data-streams and histograms. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC '01)*, pages 471–475. ACM, 2001.

Anamika Gupta, Vasudha Bhatnagar, and Naveen Kumar. Mining closed itemsets in data stream using formal concept analysis. In Torben Bach Pedersen, Mukesh K. Mohania, and A. Min Tjoa, editors, *Proceedings of the 12th International Conference on Data Warehousing and Knowledge Discovery (DAWAK '10)*, pages 285–296. Springer, 2010.

Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data (SIGMOD '00)*, pages 1–12. ACM, 2000.

Michael Bonnell Harries, Claude Sammut, and Kim Horn. Extracting hidden context. *Machine Learning*, 32(2):101–126, 1998.

Simon Haykin. *Neural Networks. A Comprehensive Foundation.* Prentice-Hall, 1999.

Christian Hidber. Online association rule mining. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD '99)*, pages 145–156. ACM, 1999.

Jochen Hipp, Ulrich Güntzer, and Gholamreza Nakhaeizadeh. Algorithms for association rule mining - a general survey and comparison. *SIGKDD Explorations*, 2(1):58–64, 2000.

Victoria Hodge and Jim Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004.

Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, 1999.

Kuen-Fang Jea and Chao-Wei Li. Discovering frequent itemsets over transactional data streams through an efficient and stable approximate approach. *Expert Systems with Applications*, 36(10):12323–12331, 2009.

Cheqing Jin, Weining Qian, Chaofeng Sha, Jeffrey X. Yu, and Aoying Zhou. Dynamically maintaining frequent items over a data stream. In *Proceedings of the 12th International Conference on Information and Knowledge Management (CIKM '03)*, CIKM '03, pages 287–294. ACM, 2003.

Ruoming Jin and G. Agrawal. An algorithm for in-core frequent itemset mining on streaming data. In *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM '05)*, pages 8 pp.–. IEEE Computer Society, 2005.

Daniel Kifer, Shai Ben-David, and Johannes Gehrke. Detecting change in data streams. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB '04)*, pages 180–191. VLDB Endowment, 2004.

Willi Klösgen. Efficient discovery of intertesting statements in databases. *Journal of Intelligent Information Systems*, 4(1):53–69, 1995.

Donald E. Knuth. *The Art of Computer Programming. Vol. 2: Seminumerical Algorithms*. Addison-Wesley, 1997.

Moshe Koppel, Jonathan Schler, and Kfir Zigdon. Determining an author's native language by mining a text for errors. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD '05)*, pages 624–628. ACM, 2005.

Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*, pages 591–600. ACM, 2010.

Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2014.

Chao-Wei Li and Kuen-Fang Jea. An approach of support approximation to discover frequent patterns from concept-drifting data streams based on concept learning. *Knowledge and Information Systems*, 40(3):639–671, 2014.

Haifeng Li, Ning Zhang, Jianming Zhu, Huaihu Cao, and Yue Wang. Efficient frequent itemset mining methods over time-sensitive streams. *Knowledge-Based Systems*, 56: 281 – 298, 2014.

Hua-Fu Li and Suh-Yin Lee. Mining frequent itemsets over data streams using efficient window sliding techniques. *Expert Systems with Applications*, 36(2):1466 – 1477, 2009.

Hua-Fu Li, Chin-Chuan Ho, Man-Kwan Shan, and Suh-Yin Lee. Efficient maintenance and mining of frequent itemsets over online data streams with a sliding window. In *Proceedings of the 2006 IEEE International Conference on Systems, Man and Cybernetics (SMC'06)*, volume 3, pages 2672–2677. IEEE Computer Society, 2006.

Hua-Fu Li, Man-Kwan Shan, and Suh-Yin Lee. Dsm-fi: an efficient algorithm for mining frequent itemsets in data streams. *Knowledge and Information Systems*, 17(1):79–97, 2008a.

Kun Li, Yong-yan Wang, Manzoor Ellahi, and Hong-an Wang. Mining recent frequent itemsets in data streams. In *Proceedings of the 5th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD'08)*, volume 4, pages 353–358. IEEE Computer Society, 2008b.

Chih-Hsiang Lin, Ding-Ying Chiu, Yi-Hung Wu, and Arbee L. P. Chen. Mining frequent itemsets from data streams with a time-sensitive sliding window. In *Proceedings of the 2005 SIAM International Conference on Data Mining (SDM '05)*, pages 68–79. SIAM, 2005.

Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA '02)*, pages 88–97. ACM, 2002.

Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*, pages 346–357. VLDB Endowment, 2002.

Heikki Mannila. Local and global methods in data mining: Basic techniques and open problems. In Peter Widmayer, Stephan Eidenbenz, Francisco Triguero, Rafael Morales, Ricardo Conejo, and Matthew Hennessy, editors, *Automata, Languages and Programming*, pages 57–68. Springer, 2002.

Heikki Mannila and Hannu Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997.

Michael Mitzenmacher and Eli Upfal. *Probability and Computing. Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, 2005.

B. Mozafari, H. Thakkar, and C. Zaniolo. Verifying and mining frequent patterns from large windows over data streams. In *Proceedings of the 24th IEEE International Conference on Data Engineering (ICDE' 08)*, pages 179–188. IEEE Computer Society, 2008.

E. S. Page. Continuous inspection schemes. *Biometrika*, 41(1/2):100–115, 1954.

Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Closed set based discovery of small covers for association rules. In *Proceedings of 15emes Journees Bases de Donnees Avancees (BDA '99)*, pages 361–381, 1999a.

Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Efficient mining of association rules using closed itemset lattices. *Information Systems*, 24(1):25–46, 1999b.

Qinghua Zou, W. W. Chu, and Baojing Lu. Smartminer: a depth first algorithm guided by tail information for mining maximal frequent itemsets. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM '02)*, pages 570–577, 2002.

Francesco Ricci, Adriano Venturini, Dario Cavada, Nader Mirzadeh, Dennis Blaas, and Marisa Nones. Product recommendation with interactive query management and twofold similarity. In Kevin D. Ashley and Derek G. Bridge, editors, *Case-Based Reasoning Research and Development*, pages 479–493. Springer, 2003.

Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor, editors. *Recommender Systems Handbook*. Springer, 2011.

Saharon Rosset, Uzi Murad, Einat Neumann, Yizhak Idan, and Gadi Pinkas. Discovery of fraud rules for telecommunications-challenges and solutions. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '99, pages 409–413. ACM, 1999.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

R. J. Serfling. Probability inequalities for the sum in sampling without replacement. *The Annals of Statistics*, 2(1):39–48, 1974.

Claudio Silvestri and Salvatore Orlando. Approximate mining of frequent patterns on streams. *Intelligent Data Analysis*, 11(1):49–73, 2007.

Xingzhi Sun, Maria E. Orlowska, and Xue Li. Finding frequent itemsets in high-speed data streams. In *Proceedings of the 2006 SIAM International Conference on Data Mining (SDM '06)*. SIAM, 2006.

Syed Khairuzzaman Tanbeer, Chowdhury Farhan Ahmed, Byeong-Soo Jeong, and Young-Koo Lee. Sliding window-based frequent pattern mining over data streams. *Information Sciences*, 179(22):3843 – 3865, 2009.

Keming Tang, Caiyan Dai, and Ling Chen. A novel strategy for mining frequent closed itemsets in data streams. *Journal of Computers*, 7(7):1564–1573, 2012.

Daniel Trabold and Tamás Horváth. Mining data streams with dynamic confidence intervals. In *Proceedings of the 18th International Conference on Big Data Analytics and Knowledge Discovery (DaWaK '16)*, pages 99–113. Springer, 2016.

Daniel Trabold and Tamás Horváth. Mining strongly closed itemsets from data streams. In *Proceedings of the 20th International Conference on Discovery Science (DS '17)*, pages 251–266. Springer, 2017.

Daniel Trabold, Mario Boley, Michael Mock, and Tamás Horváth. In-stream frequent itemset mining with output proportional memory footprint. In *Proceedings of the LWA 2015 Workshops: KDML, FGWM, IR, and FGDB (LWDA '15)*, pages 93–104. Springer, 2015.

Daniel Trabold, Tamás Horváth, and Stefan Wrobel. Effective approximation of parametrized closure systems over transactional data streams. *Machine Learning*, 2019.

Pauray S.M. Tsai. Mining frequent itemsets in data streams using the weighted sliding window model. *Expert Systems with Applications*, 36(9):11617 – 11625, 2009.

Vincent S. Tseng, Cheng-Wei Wu, Bai-En Shie, and Philip S. Yu. Up-growth: an efficient algorithm for high utility itemset mining. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '10)*, pages 253–262. ACM, 2010.

Christian Ullenboom. *Java ist auch eine Insel: Programmieren lernen mit dem Standardwerk für Java-Entwickler. Ausgabe 2019, aktuell zu Java 11.* Rheinwerk Computing, 2018.

Matthijs van Leeuwen and Arno Siebes. Streamkrimp: Detecting change in data streams. In *Proceedings of the 2008 European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD 2008)*, pages 672–687. Springer, 2008.

V. N. Vapnik and A. Ya. Chervonenkis. *Theory of Pattern Recognition.* Nauka, USSR, 1974.

Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

Abraham Wald. *Sequential Analysis.* John Wiley and Sons, 1947.

En Tzu Wang and Arbee L.P. Chen. A novel hash-based approach for mining frequent itemsets over data streams requiring less memory space. *Data Mining and Knowledge Discovery*, 19(1):132–172, 2009.

Jianyong Wang, Jiawei Han, and Jian Pei. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '03)*, pages 236–245. ACM, 2003.

Show-Jane Yen, Yue-Shi Lee, and Chiu-Kuang Wang. An efficient algorithm for incrementally mining frequent closed itemsets. *Applied Intelligence*, 40(4):649–668, 2014.

Jeffrey Xu Yu, Zhihong Chong, Hongjun Lu, and Aoying Zhou. False positive or false negative: Mining frequent itemsets from high speed transactional data streams. In *Proceedings of the 30th international conference on Very large data bases (VLDB '04)*, pages 204–215. VLDB Endowment, 2004.

Jeffrey Xu Yu, Zhihong Chong, Hongjun Lu, Zhenjie Zhang, and Aoying Zhou. A false negative approach to mining frequent itemsets from high speed transactional data streams. *Information Sciences*, 176(14):1986–2015, 2006.

M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining (KDD '97)*, pages 283–286. AAAI Press, 1997.

Dongsong Zhang and Lina Zhou. Discovering golden nuggets: data mining in financial application. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 34(4):513 –522, 2004.

F. Zhu, X. Yan, J. Han, P. S. Yu, and H. Cheng. Mining colossal frequent patterns by core pattern fusion. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE '07)*, pages 706–715. IEEE Computer Society, 2007.

Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*, pages 358–369. VLDB Endowment, 2002.

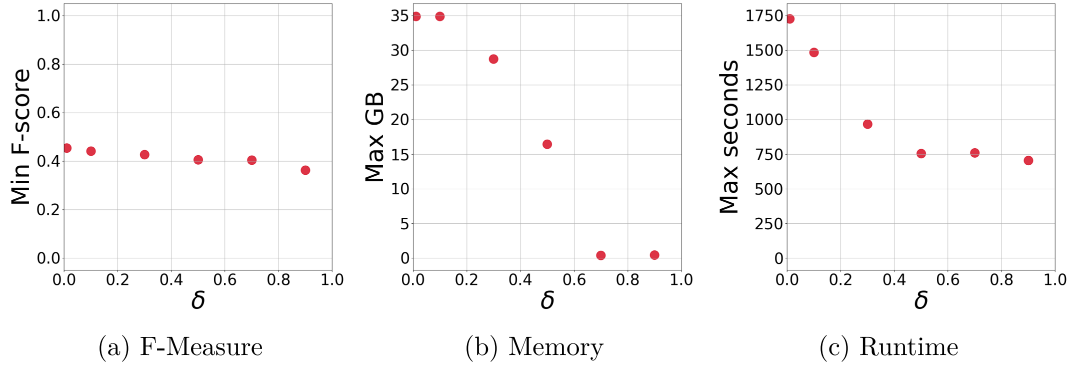
A. Parameter Tuning

This section describes the tuning of the parameters for the algorithms in Chapter 4. The data streams used and the design of the experiment follows the description of Section 4.4. We refer to this section for the details. The streams obtained from the six real-world benchmark data sets of Table 4.2 (page 58) do not have enough transactions for some combinations of the algorithms and parameters. However, their characteristic differs from that of artificial benchmark data streams. It is therefore interesting to see how the algorithms perform on the real-world and the 33 artificial benchmark data streams. The artificial data streams are obtained by the IBM Quest data generator with the parameters specified in Table 4.3 (page 59). All algorithms are hence tuned both on the classical benchmark data streams. We refer to the real-world data as UCI and the artificial data as QUEST.

Each data set is mined at five data set specific thresholds corresponding to 1k, 5k, 10k, 50k, and 100k frequent itemsets. These thresholds cover a broad range of applications. We use these thresholds for both types of data (i.e., real-world and artificial). For the six real-world data sets and the five thresholds, this results in a total of 30 experiments for each set of parameters and each algorithm. Whereas for the 32 artificial data streams and the five frequency thresholds we obtain a total of 160 experiments. Each stream is mined at regular intervals producing 10 results. The results for each data stream are aggregated to a single value corresponding to a worst-case scenario, i.e., the F-score for the entire stream is the minimal F-score out of the 10 F-scores whereas for memory and runtime the maximal value is considered. To aggregate the results from the 30 versus 160 experiments, for each set of parameters we report either a worst-case aggregation corresponding to the minimal F-score and maximal memory and runtime or the average result of all data streams and frequency thresholds. Whenever the worst-case scenario reveals the effect of parameters, results will be provided for this scenario. However, for some algorithms, the average results are more suitable to illustrate the effect of a parameter. We always present the result that shows the effect best and for DTM even both.

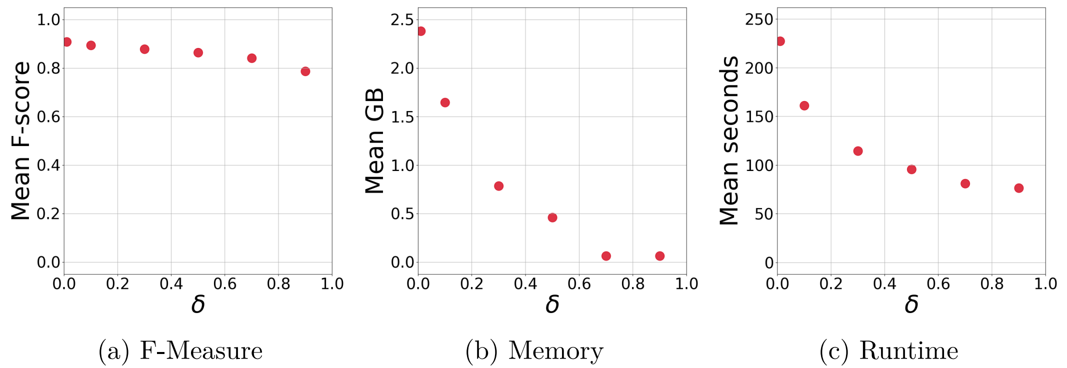
The parameter tuning will be described for each algorithm in turn, in alphabetic order. The evaluation of the specific parameters will include the parameter range used by the authors of the respective algorithm, as well as lower and larger values. This choice is based on the assumption that the authors have chosen the parameters in ranges that are suitable for similar data streams. The optimal parameters are expected to vary for different data streams but should be on the same order of magnitude. Otherwise, setting them to reasonable values for an (in advance) unknown stream is impossible.

DTM specific parameter δ Our DTM algorithm has a single parameter δ. It follows from theory that values of δ closer to 0 should provide better approximations. The parameter was evaluated at 0.0001, 0.001, 0.01, 0.1, 0.3, 0.5, 0.7 and 0.9. We only show results for the range 0.01 – 0.9. These are representative and lower values of δ do not



(a) F-Measure (b) Memory (c) Runtime

Figure A.1.: DTM worst-case effect of δ on (a) F-Measure, (b) Memory and (c) Runtime for the UCI benchmark data streams.



(a) F-Measure (b) Memory (c) Runtime

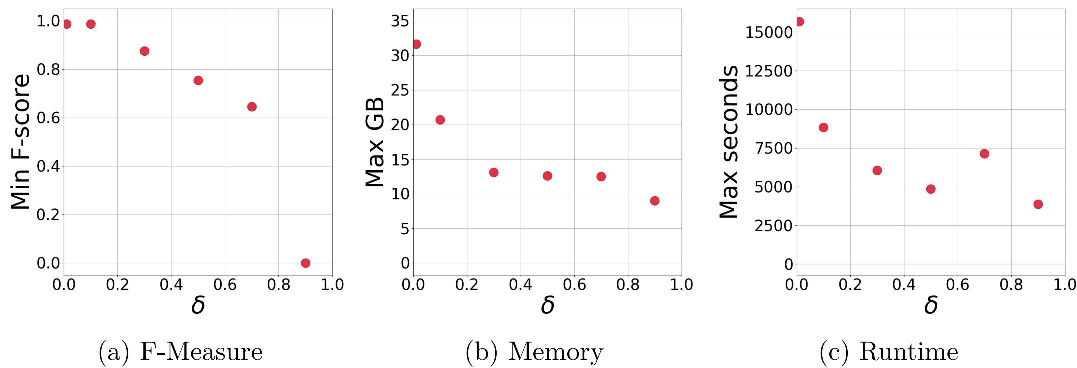
Figure A.2.: DTM average effect of varying δ on (a) F-Measure, (b) Memory and (c) Runtime for the UCI benchmark data streams.

change the overall picture. Results for the UCI data streams are given in Figures A.1 and A.2, showing the worst-case and the average case aggregations respectively. In a similar manner, Figures A.3 and A.4 report the results on the QUEST data streams.

The theoretical result, that smaller values for δ lead to an increasing F-score, is visible for both scenarios and the two data sources (i.e., real-world and artificial). The worst-case (Figures A.1a and A.3a) and the average case (Figures A.2a and A.4a) consistently show an improved F-score for lower values of δ. The improvement is best visible in Figure A.3a.

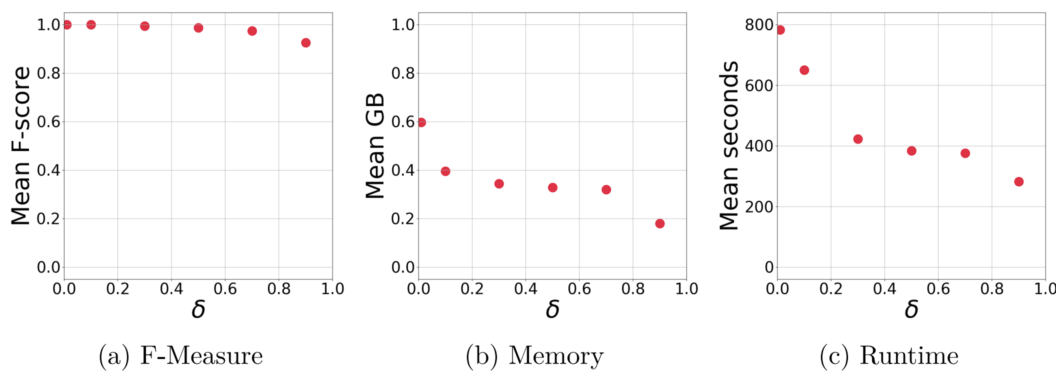
Smaller values of δ require more memory for both the real-world (Figure A.1b and A.2b) and artificial (Figure A.3b and A.4b) data streams. This confirms the theoretical considerations. The increase is visible for the worst-case and the average case scenario.

With respect to the runtime the results on both real-world (Figures A.1c and A.2c) and artificial (Figures A.3c and A.4c) data streams show that for lower values of δ the processing of the streams takes longer. This holds for the two aggregation scenarios. Note that these runtime results are very consistent with the results for memory consumption.



(a) F-Measure (b) Memory (c) Runtime

Figure A.3.: DTM worst-case effect of δ on (a) F-Measure, (b) Memory and (c) Runtime for the QUEST benchmark data streams.

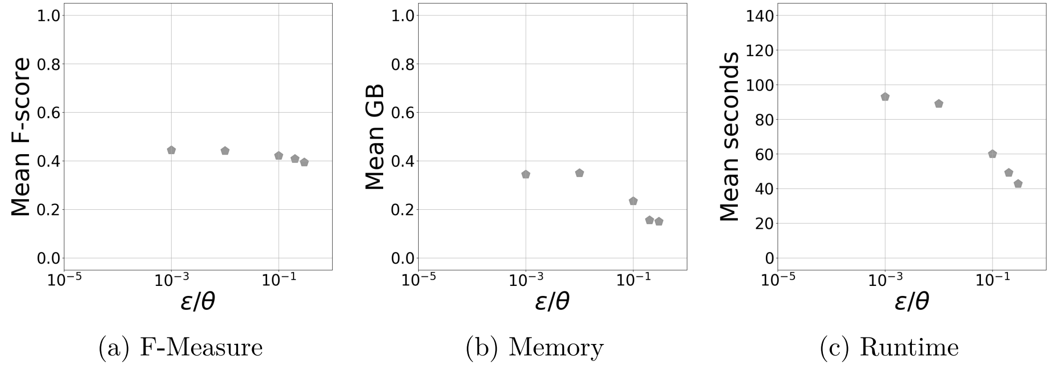


(a) F-Measure (b) Memory (c) Runtime

Figure A.4.: DTM average effect of varying δ on (a) F-Measure, (b) Memory and (c) Runtime for the QUEST benchmark data streams.

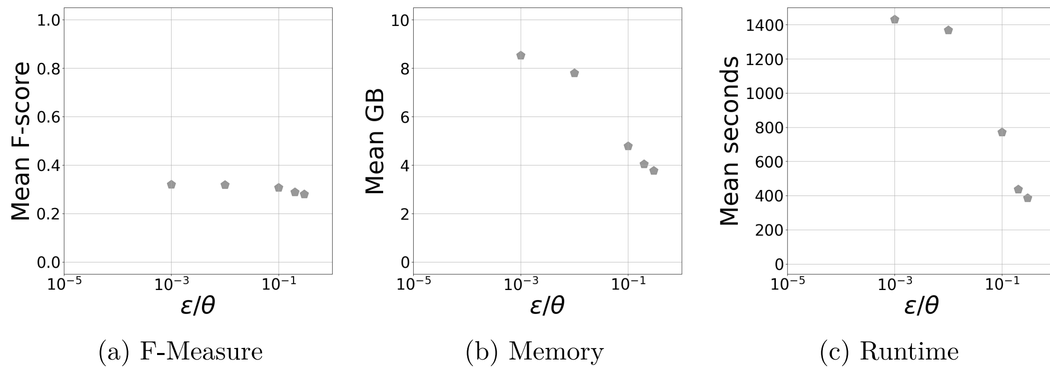
We chose the value $\delta = 0.1$ for all experiments. This value produces very good F-score results. The results improve only little with lower values. However, lower values of δ require more memory and time to process the streams.

EStream specific parameter error The EStream algorithm (Dang et al., 2008) has two parameters, one is the maximal pattern length, the other an error parameter ϵ. The maximal pattern length is set to the correct length of the longest pattern obtained by the non-streaming algorithm computing the ground truth for the entire stream. As the value for the first parameter is provided to the algorithm based on the ground truth, this parameter does not need to be optimized and only the error parameter is considered further. Dang et al. (2008) fix the value of ϵ to 0.1 of θ. We evaluate the following multiples of θ for this parameter: 0.001, 0.01, 0.1, 0.2 and 0.3. Figures A.5 and A.6 show the average results on the UCI and QUEST data streams respectively.



(a) F-Measure (b) Memory (c) Runtime

Figure A.5.: Effect of the error parameter used by EStream on (a) F-Measure, (b) Memory and (c) Runtime for all UCI benchmark data streams.

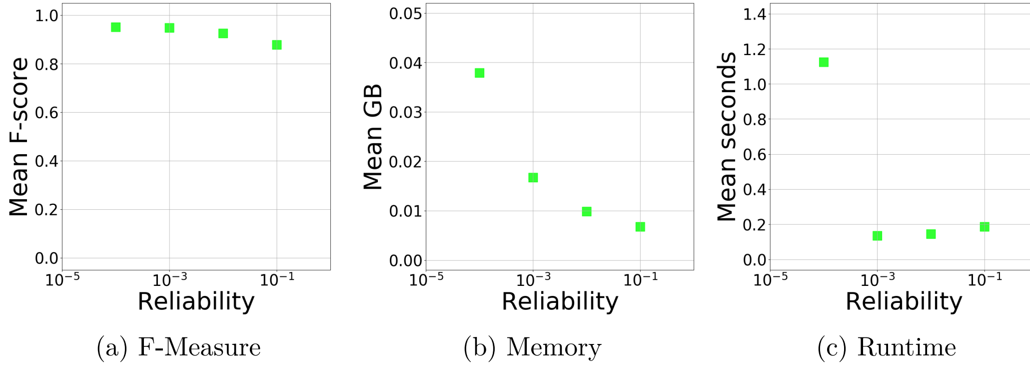


(a) F-Measure (b) Memory (c) Runtime

Figure A.6.: Effect of the error parameter used by EStream on (a) F-Measure, (b) Memory and (c) Runtime for all QUEST benchmark data streams.

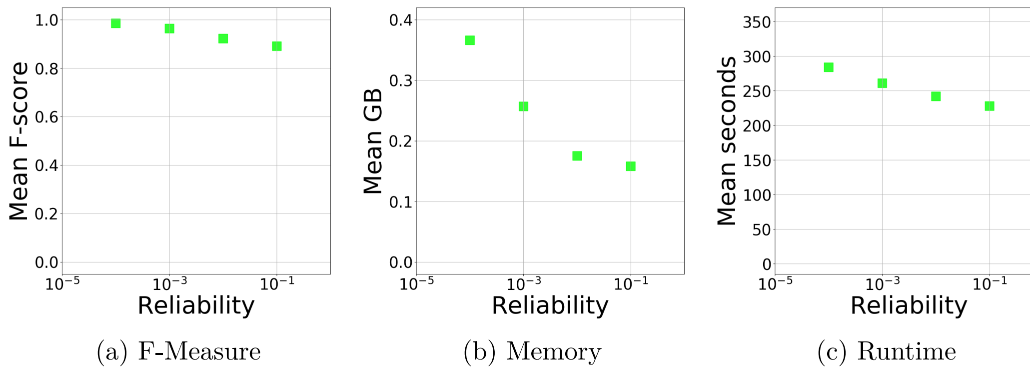
Smaller values of the error parameter result in better F-scores (Figures A.5a and A.6a), as expected. However, the effect is very small. In particular, there is very little gain from $\epsilon/\theta = 0.1$ to $\epsilon/\theta = 0.001$. See Figures A.5b and A.6b for the effect on memory and Figures A.5c and A.6c for impact on the runtime. Across all data streams, small error values seem to require both more memory and longer runtimes than larger values. The choice of ϵ seems to affect the memory and runtime far more than the F-score. While at $\epsilon/\theta = 0.1$ the algorithm needs only 2/3rd of the runtime required at $\epsilon/\theta = 0.001$ the F-scores at both values of ϵ are very close across the data sources. Based on these results, the error $\epsilon = 0.1 \cdot \theta$ will be used as the best parameter for EStream.

FDPM specific parameters reliability and k The authors of FDPM evaluate the parameter reliability δ at 0.0001, 0.001, 0.01, and 0.1 and adjust k such that $50,000$ transactions constitute one batch (Yu et al., 2004).



(a) F-Measure (b) Memory (c) Runtime

Figure A.7.: Effect of the reliability parameter used by FDPM on (a) F-Measure, (b) Memory and (c) Runtime for the UCI benchmark data streams.

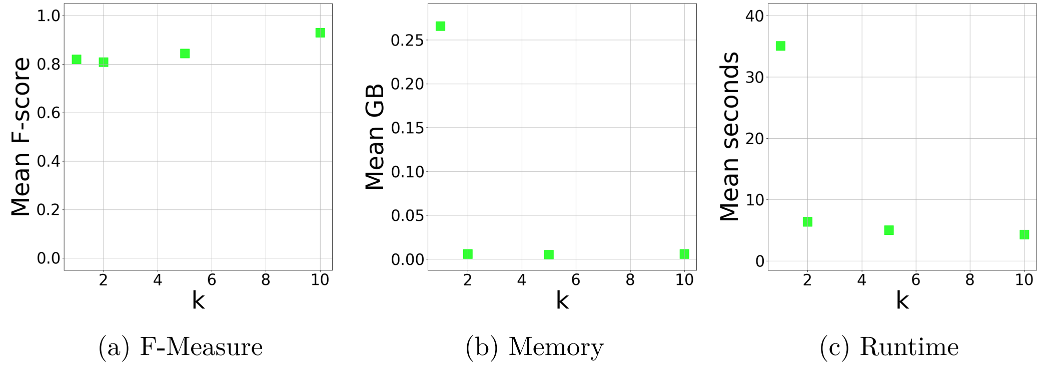


(a) F-Measure (b) Memory (c) Runtime

Figure A.8.: Effect of the reliability parameter used by FDPM on (a) F-Measure, (b) Memory and (c) Runtime for the QUEST benchmark data streams.

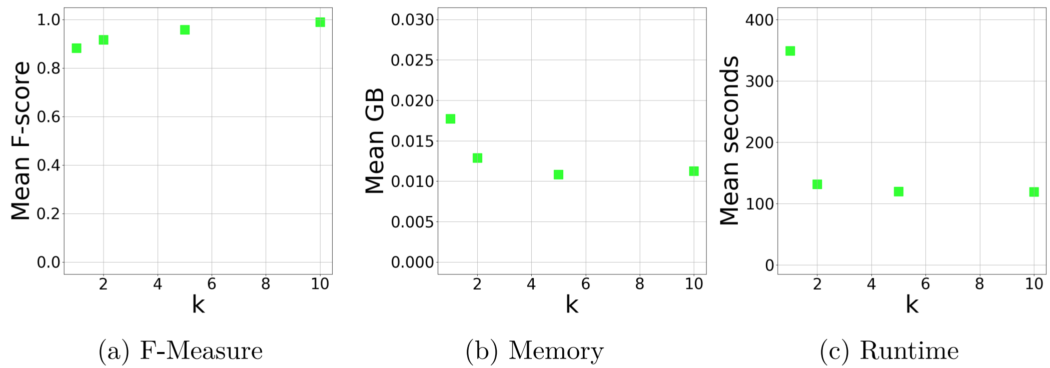
We optimize the two parameters in turn, starting with reliability. Figures A.7 and A.8 show the average effect for the UCI and QUEST data streams, respectively. In case the reliability parameter requires a buffer size larger than one-tenth of a data stream, the experiment is not run. Otherwise, k is set to the largest possible value such that no more than 1/10-th of the stream is processed at once. The situation in which the experiment can not run because the buffer size exceeds 1/10-th of the stream is more likely to occur for smaller values of the reliability parameter.

Both the F-score and the memory seem to be affected consistently by the choice of the reliability parameter. Smaller values for this parameter result in higher F-scores (Figures A.7a and A.8a), but require more memory (Figures A.7b and A.8b). The runtime result for the QUEST data follows this general pattern (Figure A.8c). Only on the UCI data, the pattern is less clear (Figure A.7c). This might be a side effect of different values for k. Note that the very fast runtime is due to the fact that we



(a) F-Measure (b) Memory (c) Runtime

Figure A.9.: Effect of the parameter k used by FDPM on (a) F-Measure, (b) Memory and (c) Runtime for the UCI benchmark data streams.



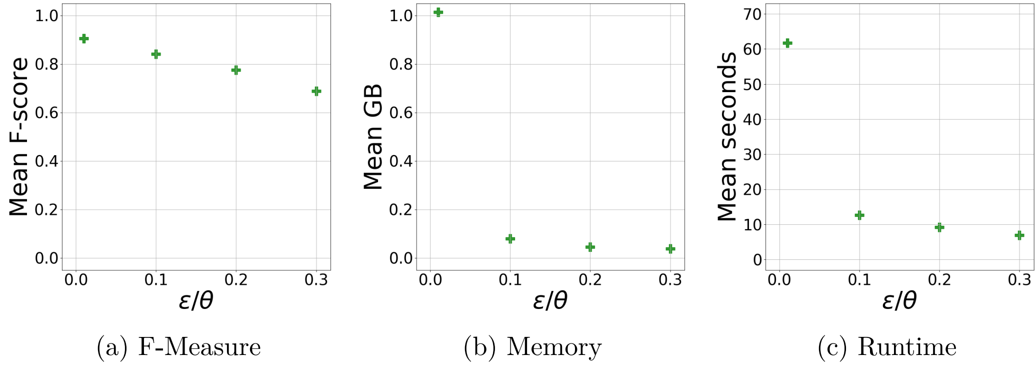
(a) F-Measure (b) Memory (c) Runtime

Figure A.10.: Effect of the parameter k used by FDPM on (a) F-Measure, (b) Memory and (c) Runtime for the QUEST benchmark data streams.

implemented the algorithm by using an FP-Tree and produce the result for each batch simply by an enumeration of the FP-Tree, without the intermediate pruning step. This optimization improves the runtime tremendously without affecting the F-score.

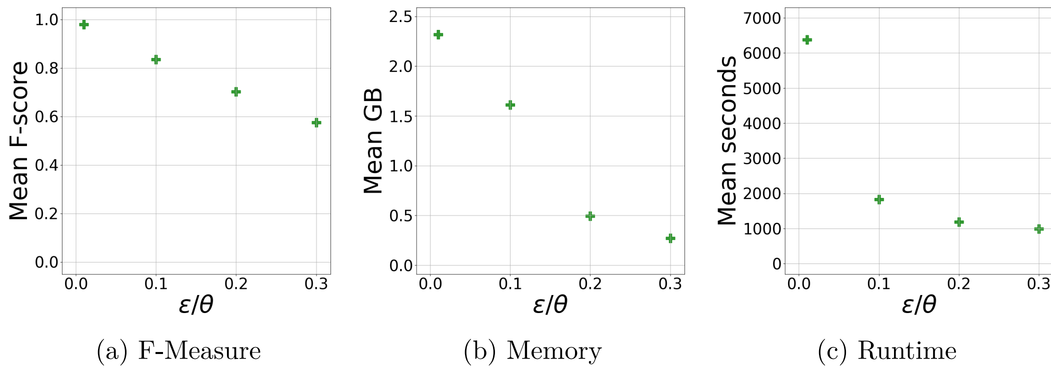
Based on these results, the value for the reliability parameter δ is set to 0.0001. For even lower values the buffer size increases further and the experiment can not be run on many data streams because the buffer size exceeds one-tenth of a data stream.

We now turn to the discussion of the parameter k which is evaluated for $\delta = 0.0001$ for the reasons above. We tested the values of $k = 1$, 2, 5, and 10. Figures A.9 and A.10 show the results for the UCI and QUEST data streams respectively. For both data sources the F-score increases with increasing k (Figures A.9a and A.10a). At the same time, the algorithm requires less memory for larger k and runs faster. Increasing k from 1 to 2 has a far larger impact than increasing k from 2 to 5 or 10. This observation holds for both memory (Figures A.9b and A.10b) and runtime (Figures A.9c and A.10c). The runtime result is expected, as the output is computed less frequent. The memory result is less obvious and an artifact of our FP-Tree-based implementation of the algorithm. In the



(a) F-Measure (b) Memory (c) Runtime

Figure A.11.: Effect of the error parameter used by LOSSY COUNTING on (a) F-Measure, (b) Memory and (c) Runtime for the UCI benchmark data streams.



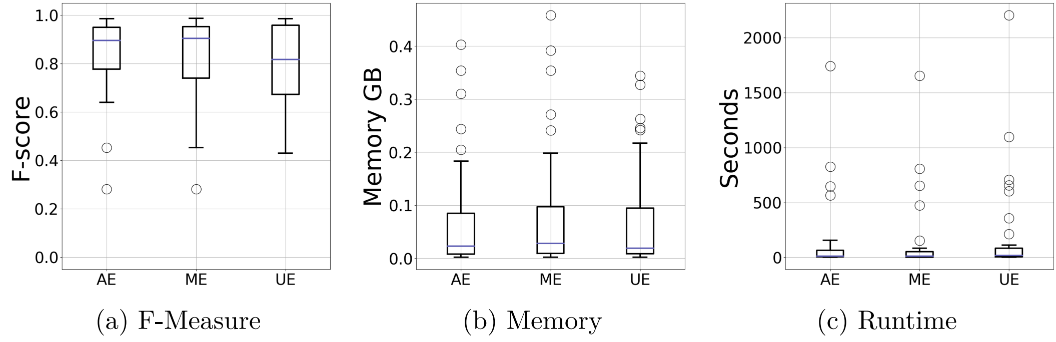
(a) F-Measure (b) Memory (c) Runtime

Figure A.12.: Effect of the error parameter used by LOSSY COUNTING on (a) F-Measure, (b) Memory and (c) Runtime for the QUEST benchmark data streams.

original version, the algorithm prunes more frequently with lower values of k and hence requires less memory. As we buffer transactions and convert them into an FP-Tree, the tree can be more compact for larger k, as more items can be pruned as infrequent from the buffer. We will use $k = 5$ for all further experiments, as for $k = 10$ some streams are not long enough at the frequency thresholds corresponding to 100k frequent itemsets.

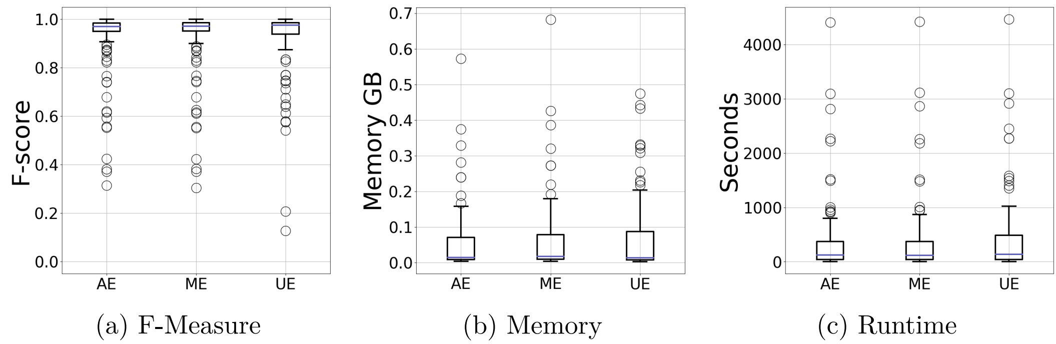
Lossy Counting specific parameter error Manku and Motwani (2002) fix the value of the error parameter ϵ to 0.1θ. We will evaluate the values 0.01θ, 0.1θ, 0.2θ and 0.3θ for the parameter ϵ. Figures A.11 and A.12 illustrate the effect of the parameter for the real-world and synthetic benchmark data streams respectively.

For all data streams, there is a clear trend that lower values of ϵ correspond to higher F-scores (Figures A.11a and A.12a). At the same time, we observe an increase in both memory and runtime. The effect on memory is better visible for the artificial data (Figure A.12b) then for the real-world benchmark data (Figure A.11b). For the former, there is a very clear increase for every change in ϵ. The changes are smaller for the UCI



(a) F-Measure (b) Memory (c) Runtime

Figure A.13.: Effect of the strategy used by Partial Counting on (a) F-Measure, (b) Memory and (c) Runtime for the UCI benchmark data streams.



(a) F-Measure (b) Memory (c) Runtime

Figure A.14.: Effect of the strategy used by Partial Counting on (a) F-Measure, (b) Memory and (c) Runtime for all QUEST benchmark data streams.

data for $\epsilon/\theta \geq 0.1$. Concerning the runtime, the effect is clearly visible for both data sources (Figures A.11c and A.12c). Note that memory and runtime do not scale together. Consider the decrease of ϵ/θ from 0.2 to 0.1 for the QUEST data streams. While the average memory required increases by a factor of 3, the average runtime increases by a factor less than 2.

Obviously, smaller values of ϵ provide better results, as expected. They require larger blocks of transactions to be processed together, which, in turn, increases the number of transactions to be processed before the next result can be obtained. The value $\epsilon = 0.01\theta$ provides the best results in terms of the F-score and will hence be used for all further experiments.

Partial Counting specific parameter estimation strategy For Partial Counting the three estimation strategies will be evaluated. Results for the UCI and QUEST data streams are shown in Figures A.13 and A.14, respectively. For each strategy, we generate

a boxplot from all the results of all data streams and all thresholds as they illustrate the distribution of the results over all configurations. Overall, the results look very similar for both real-world and artificial data.

In terms of the F-score (Figures A.13a and A.14a), the strategies behave very similarly. The average estimation strategy (AE) performs best for both data sources, followed by the minimum estimation (ME). The upper bound estimation (UE) seems to provide the worst results on average.

Regarding the memory (Figures A.13b and A.14b), the strategies behave overall very similar. The upper bound estimation requires the least memory across all data streams, average estimation comes second, and minimum estimation is last.

For the runtime (Figures A.13c and A.14c), there is only a little difference between the strategies. Surprisingly, the computation of the upper bound and minimum estimation strategies are not really faster than the average estimation.

Based on these results, the average estimation strategy will be used for the experiments. In terms of F-score, it is as good as the minimum estimation. Compared to the upper bound estimation, they are both better. Concerning memory, it is slightly better than the minimum estimation strategy. This makes the strategy the best choice.

sApriori specific parameters confidence and error For SApriori the parameters chosen by the authors for all experiments were fixed to confidence $\delta = 0.1$ and error $\epsilon = 0.002$ (Sun et al., 2006). We evaluate the parameters $\delta = 0.0001, 0.001, 0.01, 0.1, 1$ and $\epsilon \in [0.001, 0.005]$ at 0.001 increments, and optimize them sequentially. Starting with the initial value of $\epsilon = 0.002$ as used by the authors, δ is evaluated first and fixed to the best value and only after is ϵ tuned, given the best value of δ.

Figures A.15 and A.16 show the impact of the confidence parameter δ for the real-world and artificial data streams, respectively. As δ is lowered, the F-score increases for the QUEST data (Figure A.16a). The effect is less evident for the UCI data where the maximal F-score is obtained for $\delta = 0.1$ (Figure A.15a).

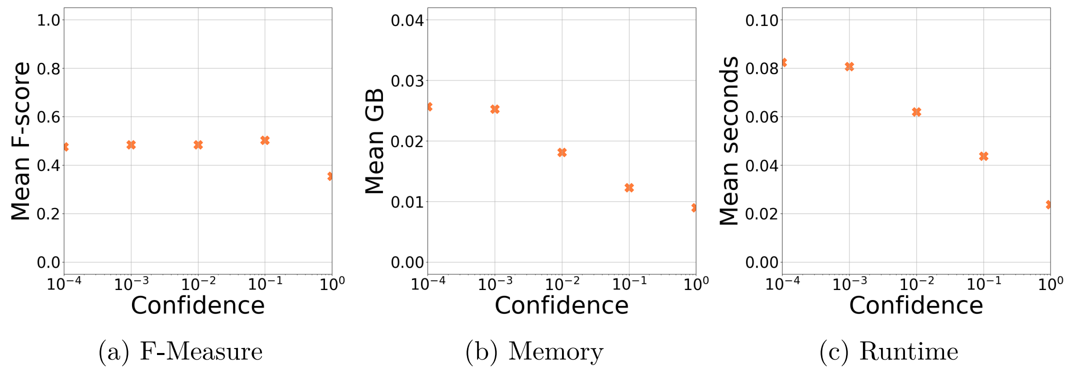
The memory consumption increases with lower confidence for the real-world data (Figure A.15b), but not necessarily for the artificial one (Figure A.16b). As the algorithm uses larger buffers for lower confidence values, it seems reasonable that it requires more memory as the confidence value is lowered.

Concerning the runtime, the results are similar to those obtained for memory. The UCI data shows a clear correlation between the confidence parameter and the runtime for lower confidence values (Figure A.15c), whereas the runtime on the artificial benchmark data is surprisingly highest for the largest confidence (Figure A.16c).

Because the SApriori algorithm is really fast, it is optimized for the F-score which is the best for low values of the confidence parameter. Therefore, δ is fixed to 0.0001.

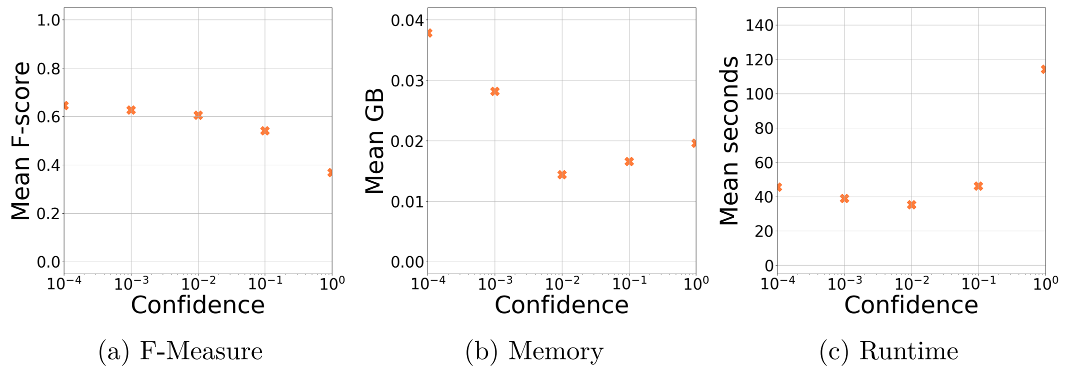
Results for varying ϵ are given in Figures A.17 and A.18 for the UCI and QUEST data streams.

The results are not consistent across the data sources. The F-scores increase for both scenarios with lower values of ϵ (see Figures A.17a and A.18a). For the UCI data, both memory and runtime follow this trend (see Figure A.17b for memory and A.17c



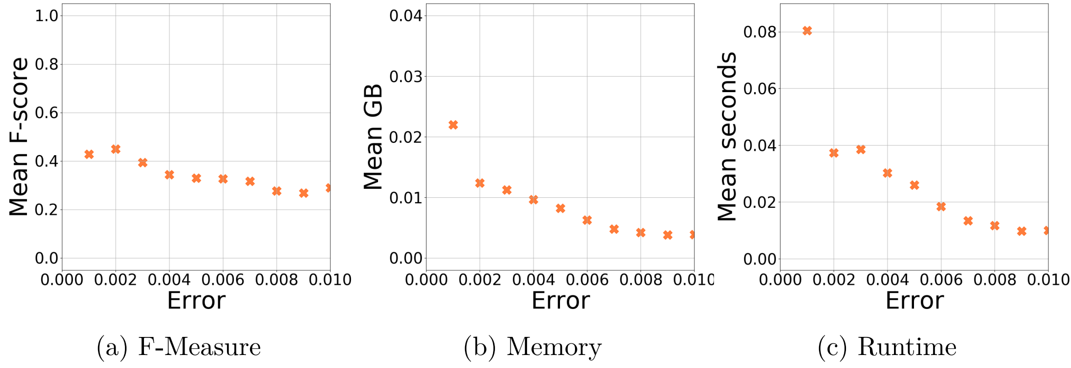
(a) F-Measure (b) Memory (c) Runtime

Figure A.15.: Effect of the confidence parameter used by SApriori on (a) F-Measure, (b) Memory and (c) Runtime for the UCI benchmark data streams.



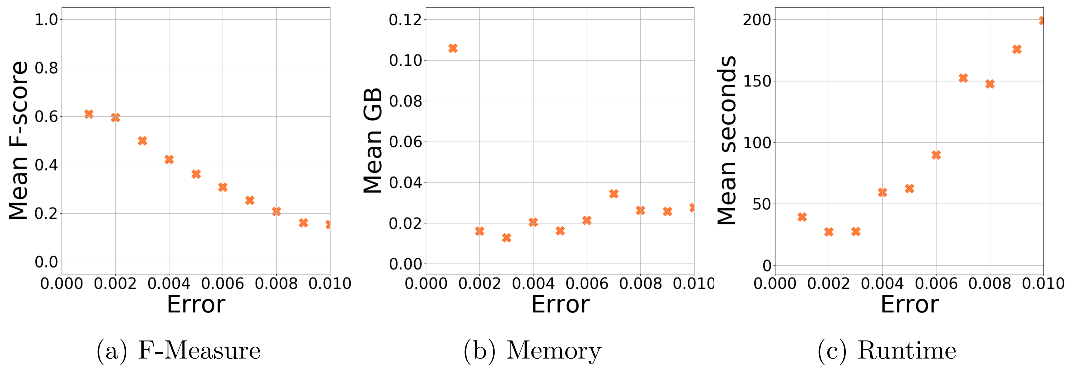
(a) F-Measure (b) Memory (c) Runtime

Figure A.16.: Effect of the confidence parameter used by SApriori on (a) F-Measure, (b) Memory and (c) Runtime for the QUEST benchmark data streams.



(a) F-Measure (b) Memory (c) Runtime

Figure A.17.: Effect of the error parameter used by SAPRIORI on (a) F-Measure, (b) Memory and (c) Runtime for the UCI benchmark data streams.



(a) F-Measure (b) Memory (c) Runtime

Figure A.18.: Effect of the error parameter used by SAPRIORI on (a) F-Measure, (b) Memory and (c) Runtime for the QUEST benchmark data streams.

for the runtime). For the QUEST data, there is a peak in memory for low values of ϵ (Figure A.18b), but in general, there is no clear trend in the memory consumption associated with the value of ϵ. Considering the runtime, the trend is that larger values of ϵ require more time (Figure A.18c). Thus, especially the runtime results for the two data sources seem to follow opposing patterns when we compare Figure A.17c to A.18c. Based on these results, the value $\epsilon = 0.001$ seems to be the best choice. The difference in terms of F-score to the value $\epsilon = 0.002$ is, however, very small. For θ set such that 100k frequent itemsets can found and $\epsilon = 0.001$, the block size exceeds the size of one-tenth of most data streams and thus the result can not be computed based on the design of the experiment. We will, therefore, fix $\epsilon = 0.002$ for the experiments.

Summary In this appendix, we have carefully tuned the algorithmic parameters of the algorithms mining frequent itemsets from data streams. We reported the detailed results in terms of F-score, memory, and runtime for each algorithm for a range of parameters. We have chosen the optimal parameters for each algorithm based on results

Algorithm	Parameter(s)
DTM	$\delta = 0.1$
EStream	$\epsilon = 0.1\theta; k =$ as from ground truth
Lossy Counting	$\epsilon = 0.01\theta$
Partial Counting	strategy = average estimation
FDPM	$k = 5; \delta = 0.0001$
SApriori	$\epsilon = 0.002\theta; \delta = 0.0001$

Table A.1.: Optimal parameters for frequent itemset mining algorithms.

on both real-world and artificial data streams. We finally summarize the optimal parameter values of the algorithms for the set of data streams and thresholds used in our experiments in Table A.1. The table serves as a reference within this work. For different applications or data streams, other parameters could be more suitable. For our purpose, these parameters provide good results for each algorithm and a foundation for a fair comparison.