Title Methods for frequent pattern mining in data streams within the MOA system

Volume 1

Student Massimo Quadrana

Director Ricard Gavaldà Mestre

Department Llenguatges i Sistemes Informàtics

Date 27 june 2012

UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC)
BarcelonaTech
Facultat d'Informàtica de Barcelona (FIB)

Methods for frequent pattern mining in data streams within the MOA system

Departament de Llenguatges i Sistemes Informàtics

Director: Ricard Gavaldà Mestre

Final project of :
Massimo Quadrana

Academic year 2011-2012

A mio papà Franco,
a mia mamma Antonella,
alle mie sorelline Ilaria e Sara,
siete sempre con me,
ovunque io sia.

To my father Franco,
to my mother Antonella,
to my little sisters Ilaria and Sara,
you are always with me,
wherever I am.

Abstract

Data stream mining has been intensively investigated in the last decades, since researchers figure out the necessity to mine efficiently very huge quantities of data coming, e.g., from Internet.

Among the its several applications, a large effort has been made to find such way to mine efficiently Frequent Itemsets over a possibly infinite stream of data.

In this work we provide a robust, efficient, practical, usable and extendable solution to perform Frequent Itemset mining over data streams. We study the peculiar requirements of elaborating data online, we analyze different solutions that have been proposed so far by the scientific community, then we implement the IncMine algorithm due to Cheng et al. with several instructions. The solution is implemented in the Massive Online Analysis (MOA) framework for stream mining.

We perform several testings, with both synthetic and real data streams, to testify the quality of the proposed solution.

Acknowledgements

I am extremely grateful to my advisor, Ricard Gavaldà Mestre. His knowledge, his motivation, and his support were fundamental in every moment. Nothing of this work would have been done without him.

I would like thank my old and new friends, thank for being here at my side in the good and bad times. No words can express how I am really grateful to you. Your friendliness, words and support were essential in this amazing experience.

Most of all, I am grateful to my family, my mother Antonella and my sisters Ilaria and Sara. Thank for your patience and your endless support, without you I would never have been anyone.

Contents

Chapter 1

Introduction

In this section we present an overview of the entire work. We mention the problem we have studied and goals we wanted to achieve. We provide the intended schedule and an economic estimate of the work done.

1.1 The problem

Our work belongs to the area of Data Mining. A commonly accepted definition for data mining is the discovery of *models* from data. As computer scientists, we are also interested in the efficiency of the methods, their scalability to large data, and their integration with other software platforms. Machine learning, a field emerged from the computer science community, is together with statistics one of the main supports of data mining.

There are many approaches to modeling data, but most of them can be ascribed to one of the following characteristics:

- *summarization*, i.e., finding short but informative summaries of the data.

- *feature extraction*, that is, finding the most extreme examples of a phenomenon. A complex relationship among object in a model can be expressed by finding the dependencies among this examples and using only those in representing all the connections.

- *clustering*, that is, examine collections of 'points' and group them into 'clusters' accordingly to some distance measure. The goal is that points in the same cluster have a small distance from one another, while points in different clusters are at a large distance from one another.

- *classification* or *prediction*, that is, learning to predict one particular feature of the data as a function of the others. Models such as linear and nonlinear regression, Bayes nets, decision trees, hidden Markov models are among those proposed for prediction.

- Finally, *frequent pattern mining*, that is finding patterns that occur often (and especially, more often than one would expect by chance) in the data.

An important kind of frequent pattern mining is *Frequent Itemsets Mining*. Usually this problem is presented by introducing the *market-basket* model of data. This model is used to describe a common form of many-many relationship between *items* and *baskets* (or *transactions*). Each basket consists of a set of items, called *itemset*, and usually the number in a basket is much smaller that the total number of items. The number of baskets is assumed to be very large, bigger than what can fit in main memory. The data is assumed to be represented as a sequence of baskets.

It can be applied, e.g., in real marketing applications, or in plagiarism detection, and many others. Frequent itemset mining is also important for many other data mining applications, such as association rules, correlations, sequential patterns.

In more recent times, the explosion of the information coming from networks, such as social networks in the Internet or sensor networks, requires the introduction of a different paradigm of mining. Since now mining algorithms were studied to be applied of over static databases, that is, all the data is available at the time of processing. But when this data is not entirely available, or the database is to huge to be treated efficiently with the common approaches, it becomes necessary to elaborate data *online*.

From this considerations comes the *data stream mining* paradigm. Data arrives in one or more streams, it is processed and then deleted. No data is stored, a part from the modeling information we are interested in. Data stream mining requires high processing speeds, since data have to be read and elaborated before the coming of the successive information. Differently from static data processing, it is not possible to perform multiple scans on data.

We examine here the problem of mining Frequent Itemsets over data streams. We will see what are the principal solutions that are used in static datasets mining, such as the famous Apriori principle and different others. We discuss over the different kinds of itemsets and what advantages they offers. Then we analyze what are the requirements of online processing, and,

after a deep analysis over several stream algorithms, we are able to define the main characteristics of our solution.

1.2 Goals of the project and results obtained

We noticed that there is no really usable solution to mine frequent itemset over data streams. So we decide to cover this hole providing a robust, efficient, practical, usable and extendable solution to perform Frequent Itemset mining over data streams. From the beginning, and as a requirement from the advisor, we wanted to integrate our solution into the Massive Online Analysis framework [4], a data stream mining framework developed by the University of Waikato, New Zeland. We use its capabilities to guarantee the generality, portability and usability of the program.

After an intense search and analysis of several proposals about frequent itemset mining over data streams, and for reasons that we explain in Chapter 3, we decided to implement the INCMINE algorithm of Cheng et al. [6]. We also added an efficient implementation of the CHARM algorithm [31, 32], necessary for the correct working of the algorithm, and we used efficient inverted indexing, as we explain in Chapter 4.

We obtain very good performance in processing several synthetic data streams. We obtain a really competitive solution, that clearly outperforms MOMENT algorithm [13], a state-of-the-art algorithm for frequent closed itemset mining over data streams. This is even though the MOMENT implementation we tried was coded in C++, which is known to be faster that Java.

We also study the effects of concept drift over the mining procedure. To do this, we introduce different synthetic drifts into data streams. We study the influence of the parameters of the algorithm over the processing. We also test our solution over a data stream generated from real data. This intensive testing phase is intended to verify the quality of the final system. We explain and comment the results of the testing phase in Chapter 5.

1.3 Work planning

The project has been subdivided into the following phases:

Required knowledge acquisition Before any immersion into the real topic, it was necessary to acquire the knowledge necessary to understand the data stream mining problem. In this phase we also familiarized with the MOA framework and its A.P.I.

Paper analysis In this phase we analyze and compare several works (about 10-15 different papers) about frequent itemset mining and frequent closed itemsets over data streams. Doing this we became conscious of functionalities that our proposal should have and we were able to guide all the subsequent phases.

Design and implementation In this phase we design and code the program, implementing all the functionalities of the selected solution.

Testing I In this phase we test the program in order to identify errors in the implementation. It includes the successive recoding.

Testing II In this phase we perform tests over synthetic and real data streams. We evaluate the performance of the program and we study the effects of concept drift.

Reporting In this phase we write down the report.

The estimation of hours per stage, schedule, and cost is as follows:

Phase	Deadline	Hours	Cost	Total
Required knowledge acquisition	15/02/2012	75	15 €/h	1125 €
Paper analysis	15/03/2012	150	15 €/h	2250 €
Design/Implementation	30/04/2012	225	20 €/h	4500 €
Testing I	15/05/2012	75	15 €/h	1125 €
Testing II	31/05/2012	75	15 €/h	1125 €
Reporting	20/06/2012	100	15 €/h	1500 €
TOTAL	-	**600**	-	**10125 €**

Table 1.1: Time schedule and economic cost of the project.

The cost is fictitious as it has not been developed commercially. Table 1.1 intents to estimate the economic cost of each of the phases of the project. The cost per hour is intended as an approximation of the current cost per work hour of young analysts and developers in our environment.

Chapter 2

Background knowledge

In this chapter we present the background related to problem to be addressed in this work. We present the problem of *Frequent Itemset Mining* and *Frequent Closed Itemset Mining*. We present some applications of this branch of Data Mining in the batch case. We present the *Data Stream Mining* problem and the peculiar issues that arise by mining data online. Finally we present the **MOA** framework and we detail our contribution to its functionalities. The discussion in these sections is to some extent taken from [28, 29, 38].

2.1 The Frequent Itemset Mining problem

The discovery of frequent itemsets is one of the major families of techniques for characterizing data. It aims to find correlations among data, and it is often viewed as the discovery of association rules, although this is a more complex characterization of data that can be derived from frequent itemsets analysis.

Now we provide a formal definition of the market-basket model and of the frequent itemset mining problem.

Let $\mathcal{I} = \{x_1, x_2, \ldots, x_m\}$ be a set of binary-valued attributes called *items*. A set $X \subseteq \mathcal{I}$ is called an *itemset*. An itemset of size k is called a k-itemset. We denote by $\mathcal{I}_k$ the set of all k-itemsets, i.e, subsets of $\mathcal{I}$ of size k.

Let $\mathcal{T} = \{t_1, t_2, ..., t_m\}$ be a set of transaction identifiers, also called *tids*. A set $T \subseteq \mathcal{T}$ is called *tidset*. We assume that itemsets and tidsets are kept sorted in lexicographic order.

A *transaction* is a tuple of the form $\langle t, X \rangle$, where $t \in \mathcal{T}$ is a unique transaction identifier, and X is an itemset. We refer to a transaction $\langle t, X \rangle$ by its identifier t.

A binary database $\mathcal{D}$ is a binary relation on the set of items and tids, i.e., $\mathcal{D} \subseteq \mathcal{I} \times \mathcal{T}$. We say that tid $t \in \mathcal{T}$ *contains* item $x \in \mathcal{I}$ if and only if $(x, t) \in \mathcal{D}$.

For a set X, we denote by 2^X the powerset of X, i.e., the set of all subsets of X. Let $\boldsymbol{i} : 2^{\mathcal{T}} \to 2^{\mathcal{I}}$ be a function, defined as follows:

$$\boldsymbol{i}(T) = \{x | \forall t \in T, t \text{ contains } x\} \tag{2.1}$$

That is, $\boldsymbol{i}(T)$ is the set of items that are contained in *all* the tids in the tidset T. In particular, $\boldsymbol{i}(t)$ is the set of items contained in tid $t \in \mathcal{T}$.
Let $\boldsymbol{t} : 2^{\mathcal{I}} \to 2^{\mathcal{T}}$ be a function, defined as follows:

$$\boldsymbol{t}(X) = \{t | X \subseteq \boldsymbol{i}(t)\} \tag{2.2}$$

That is, $\boldsymbol{t}$(X) is the set of transactions that contain *all* the items in the itemset X. In particular, $\boldsymbol{t}$(x) is the set of tids that contain the single item $x \in \mathcal{I}$.

The *support* of an itemset X in a dataset $\mathcal{D}$, denoted $sup(X, \mathcal{D})$, is the number of transactions in $\mathcal{D}$ that contain X, i.e.,

$$sup(X, \mathcal{D}) = |\{t_i | \langle t_i, \boldsymbol{i}(t_i)\rangle \in \mathcal{D} \text{ and } X \subseteq \boldsymbol{i}(t_i)\}| \tag{2.3}$$

That be easily defined in terms of the cardinality of its corresponding tidset $\boldsymbol{t}$(X):

$$sup(X, \mathcal{D}) = |\boldsymbol{t}(X)| \tag{2.4}$$

Let us fix some user-defined minimum support threshold $minsup$. Then X is said to be *frequent* in $\mathcal{D}$ if $sup(X, \mathcal{D}) \geq minsup$. When there is no confusion about D and $minsup$ we will drop then and simply say "X is frequent" and write its support as $sup(X)$.

We use the set $\mathcal{F}$ to denote the set of all frequent itemsets, defined as

$$\mathcal{F} = \{X | X \subseteq \mathcal{I} \text{ and } sup(X) \geq minsup\} \tag{2.5}$$

Table 2.1 shows an example of a transactions database and the relative frequent set $\mathcal{F}$ for minimum support threshold $minsup = 3$.

The formal definition above can be easily explained in terms of basket case analysis. For example, consider an itemset X as a subset of products that a department stored sells. Every receipt can be seen as a transaction, which reports the time of the buy and the products bought by a client. Thus, the transaction database D is the stack of all receipts of purchase of

Tid t	$\boldsymbol{i}$(t)
1	abde
2	bce
3	abde
4	abce
5	abcde
6	bcd

sup	itemsets
6	b
5	e,be
4	a,c,d,ab,ae,bc,bd,abe
3	ad,ce,de,abd,ade,bce,bde,abde

Table 2.1: Example of a Transactions Database and Frequent Itemsets with $minsup = 3$

the store. Discover frequent itemsets corresponds to identify products that clients frequently buy together.

An important property of transactional databases is their *density*, i.e. of how much the transactions inside the dataset resemble one with another. The more patterns and correlations are contained in such data, the more resources are needed to extract them. Conversely, in *sparse* datasets transactions differ a lot one from another. A formal definition of density of datasets is still not available in literature, and its analysis escapes from the scope of our work. Although, is important to consider that dataset density has an important influence over the resources needed to extract frequent itemsets and on their number. That is, the more dense is, the bigger will be the number of frequent itemsets that can be mined from it. An interesting statistical analysis over this aspect is provided by Palmerini et al. in [44].

2.1.1 The Apriori principle

One possible way to get the frequent set of a transactions database is to enumerate all possible itemsets and then determine the frequent ones. This *brute force* approach typically results in huge waste of computations since many of the candidates may not be frequent.

The candidate itemset search space is clearly exponential, since there are $2^{|\mathcal{I}|}$ potentially frequent itemsets, which is also referred as the *exponential explosion* of the number of subsets. Enumerating all itemsets thus has a $O(2^{|\mathcal{I}|})$ computational complexity.

This search space forms a *lattice* structure (shown in Figure 2.1), where two itemsets X and Y are connected by a link if and only if X is direct subsets of Y. The itemsets can be enumerated using either a breadth-first (BFS) or a depth-first (DFS) search on the *prefix tree* (shown in Figure 2.2). In a prefix tree structure two itemsets X, Y are connected by a link if and only if

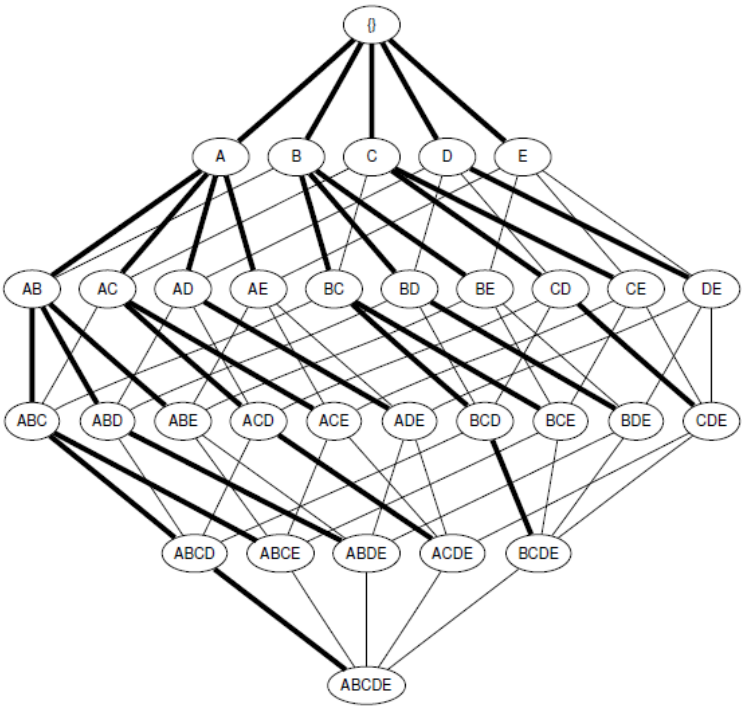


Figure 2.1: Itemset lattice and Prefix-Based search tree (in bold).

X is a direct subset and prefix of Y. We refer to this kind of representation in different points of our work.

Then for each enumerated subset $X \subseteq \mathcal{I}$, we have to check whether X is a subset of a transaction $\langle t, \boldsymbol{i}(t)\rangle \in \mathcal{D}$ and consequently increase its support, which cost at most $O(|\mathcal{I}| \times |\mathcal{D}|)$ with the direct implementation. So the overall computational complexity of a brute force approach is $O(|\mathcal{I}| \times |\mathcal{D}| \times 2^{|\mathcal{I}|})$. It is clearly unfeasible to treat with this complexity, so several solutions have been found to reduce the effort needed to compute frequent itemsets.

Consider any two itemsets $X, Y \subseteq \mathcal{I}$. If $X \subseteq Y$, then $sup(X) \geq sup(Y)$, which leads to the following corollaries

- If X is frequent, then any subset $Y \subseteq X$ is also frequent.
- If X is not frequent, then any superset $Y \supseteq X$ cannot be frequent.

By applying the above observations we can reduce the number of candidates we generate, and so significantly improve our mining performances. We can *prune* the search space by, firstly, stop generating supersets of an unfrequent candidate, since none of its supersets can be frequent, and secondly, we can avoid any candidate that has an unfrequent subset.

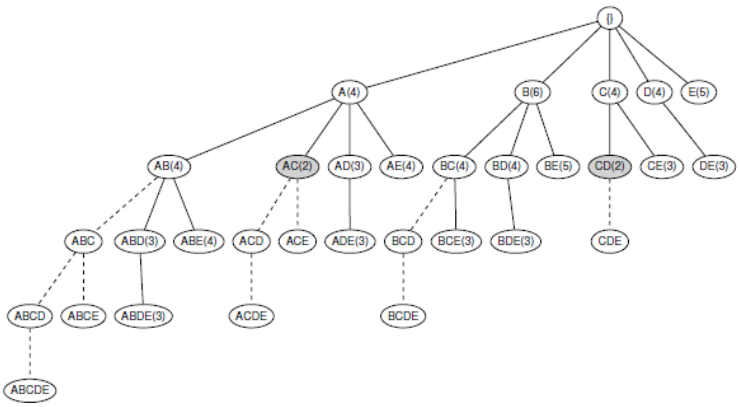


Figure 2.2: Prefix search tree and effect of pruning of the Apriori algorithm.

In Algorithm 2.1 is shown a pseudo-code version of the APRIORI algorithm. We denote as C_k and F_k the set of candidate and frequent k-itemsets respectively. Once identified the set of all frequent 1-itemsets, F_1, the algorithm iteratively generate new candidate k-itemsets using the frequent $(k-1)$-itemsets found in the previous iteration. Candidate generation is implemented using apriori-gen function, which uses the apriori properties explained above to establish a policy of candidate generation and pruning. Then the support of the subsets of these candidate is updated and unfrequent candidates are pruned. The algorithm terminates once no new frequent itemset is generated.

The worst case complexity of the APRIORI algorithm is still $O(|\mathcal{I}| \times |\mathcal{D}| \times 2^{|\mathcal{I}|})$, since all itemsets may be frequent. But in practices the cost is much smaller, due to the pruning effects, but it is difficult to characterize.

Other well known proposals in frequent itemset mining algorithms are ECLAT [30] and FP-GROWTH [3]. Both tries to *index* the transactions database to perform fast support counting.

The former uses a depth-first approach to explore the prefix tree, using the tids information of each item (tidset) for doing an efficient support counting. It achieves a computational complexity of $O(|\mathcal{D}| \times 2^{|\mathcal{I}|})$.
The latter uses an augmented prefix tree called the *frequent pattern tree* (FP-tree) that stores the support information for the itemsets prefixes over all transactions. We do not provide a detailed analysis of these algorithms, but their pseudo-codes can be found in Appendix A.1 and A.2.

Algorithm 2.1 Apriori($\mathcal{D}$,$\mathcal{I}$,$minsup$)

1: $k = 1$
2: $F_k = \{X | X \in \mathcal{I} \wedge sup(X) \geq minsup\}$ /* Find all frequent 1-itemsets */
3: **repeat**
4: $k = k + 1$
5: $C_k =$ apriori-gen(F_{k-1}) /* Generate candidate itemsets */
6: **foreach** transaction $\langle t, X \rangle \in \mathcal{D}$ **do**
7: $C_t =$ subset(C_k,X) /* Identify all candidates in $\langle t, X \rangle$ */
8: **foreach** candidate itemset $C \in C_t$ **do**
9: $sup(C) = sup(C) + 1$
10: $F_k = \{X | X \in C_k \wedge sup(X) \geq minsup\}$ /* Extract all frequent k-itemsets */
11: **until** $F_k = \emptyset$
12: **return** $\bigcup F_k$

2.1.2 Summarizing Itemsets

The search space for frequent itemsets is usually very large and grows exponentially with the number of items. A way to reduce this cost is to determine a representation of the frequent itemsets that summarize the characteristics of the set of itemsets totally. By removing all redundant information in the frequent itemsets we get a much smaller representation that the original one, reducing the computational and storage demands, but also it eases any further analysis on the correlations found. We examine two possible compact representations for frequent itemsets over a database: *maximal frequent itemsets* and *frequent closed itemsets*.

A frequent itemset $X \in \mathcal{F}$ is called *maximal* if it has no frequent superset. The set $\mathcal{M}$ of all maximal frequent itemsets is given as

$$\mathcal{M} = \{X | X \in \mathcal{F} \text{ and } \nexists Y \supset X, \text{ such that } Y \in \mathcal{F}\} \tag{2.6}$$

In the example in Table 2.1, itemsets 'abde' and 'bce' are the only two maximal frequent itemsets.

A frequent itemset $X \in \mathcal{F}$ is *closed* if it has no frequent superset *with the same support*. The set $\mathcal{C}$ of all frequent closed itemsets can be defined as

$$\mathcal{C} = \{X | X \in \mathcal{F} \text{ and } \nexists Y \supset X \text{ with } sup(X) = sup(Y)\} \tag{2.7}$$

Another definition, more abstract but algorithmically more useful, is based on the *closure operator* $\boldsymbol{c} : 2^{\mathcal{I}} \rightarrow 2^{\mathcal{I}}$, which is defined as

$$\boldsymbol{c}(X) = \boldsymbol{i} \circ \boldsymbol{t}(X) = \boldsymbol{i}(\boldsymbol{t}(X)) \tag{2.8}$$

Algorithm 2.2 SupportCounting($\mathcal{C}$)

1: k_{max} = maximum size of itemsets in $\mathcal{C}$
2: $\mathcal{F}_{k_{max}} = \{X | X \in \mathcal{C}, |X| = k_{max}\}$ /* All frequent itemsets of size k_{max} */
3: **for** $k = k_{max} - 1$ downto 1 **do**
4: **foreach** $X \in \mathcal{F}_k$ **do**
5: **if** $X \notin \mathcal{C}$ **then**
6: $X.support = \max \{X'.support | X' \in \mathcal{F}_{k+1}, X \subset X'\}$

An itemset X is called *closed* if $\boldsymbol{c}(X) = X$, i.e., if X is a fixed-point of the closure operator $\boldsymbol{c}$. We detail the properties of this operator in Appendix A.3.

In the in Table 2.1, itemsets 'b', 'bc', 'bd', 'be', 'abe', 'bce' and 'abde' are the closed ones. Usually the number of frequent closed itemsets $\mathcal{C}$ is much smaller than that of frequent itemsets $\mathcal{F}$.

Furthermore, the most important property is that we can derive $\mathcal{F}$ from $\mathcal{C}$, because a frequent itemset I must be a subset of one or more frequent closed itemsets, and its support is equal to the maximal support of those closed itemsets that contain I. Thus, if we know the closed itemsets and their respective supports, we are able to reconstruct the support information of every possible subset. This assumption is still true if work only with frequent itemsets.

In summary, the relation among $\mathcal{F}$, $\mathcal{C}$ and $\mathcal{M}$ is $\mathcal{M} \subseteq \mathcal{C} \subseteq \mathcal{F}$. Since $\mathcal{C}$ is smaller than $\mathcal{F}$ and preserve all the information about any frequent itemset, we focus our work on mining frequent closed itemset, because they provide a compact representation but sufficient to maintain the information to determine all frequent itemsets as well as their support.

If cleverly used, closed itemset mining can reduce the execution time and the memory consumption in frequent itemset mining. We can look first for frequent closed itemsets in a transaction database, extract and store them efficiently. Then, the set of frequent itemsets can be quickly obtained from the closed set.

Algorithm 2.2 is a simple procedure to get support counting of the frequent itemsets set $\mathcal{F}$ starting from a frequent closed itemset set $\mathcal{C}$.

2.2 Frequent Closed Itemset mining

In Section 2.1.1 we provide a detail explanation of the APRIORI algorithm, since it contains the main principles used in every frequent itemset mining

algorithm. We now explain CHARM, a method for frequent closed itemset mining, which is directly used in our solution, and treats with closed itemsets, differently form APRIORI. We will not explain in this details many of the other algorithms we studied, especially in Chapter 3.

Mining frequent closed itemsets requires that we perform closure checks, i.e., $X = \boldsymbol{c}(X)$. Perform these checks directly can be very expansive, since we have to verify that X is the largest itemset common to all the tids in $\boldsymbol{t}(X)$, i.e., $X = \bigcap_{t \in \boldsymbol{t}(X)} \boldsymbol{i}(t)$.

CHARM [31] is a more efficient method which performs vertical tidset intersections for closure checking. The Properties 1,2,3 exploited by the algorithm, which justify its correctness, are detailed in Appendix A.4. Algorithm 2.3 is a pseudo-code for CHARM. It initially takes as input the set of all frequent single items, along with their tidsets, and the set of all closed itemsets $\mathcal{C}$ is initially empty. Any IT-paid set $P = \{\langle X_i, \boldsymbol{t}(X_i)\}$ is sorted by increasing support ($|\boldsymbol{t}(X_i)|$). Then we try to extend each itemset X_i with all other items X_j in sorted order, applying these three properties to prune the branches where possible.

2.3 Frequent Itemset Mining applications

Frequent itemset mining is fundamental to many important data mining tasks, such as association rules, correlations, sequential patterns, classification and clustering.

As suggested by the market-basket model, the original application was in the analysis of true market baskets. Supermarkets stores record the contents of every market basket brought, where each item is different product that the store sells and the baskets are the sets of products in a single market basket. By mining frequent itemsets, a retailer can learn what items are commonly bought together. One can find pairs of supermarket products that are bought frequently more than would be expected if such products were bought independently. This can help the retailer to decide product placement in shelves, pricing policies, offers for customers who buy several products together, etc.

This concept can be applied to many other different contexts and mine different kind of data. For example frequent itemset analysis can help in finding:

- *Related concepts* in documents, where the items are words and baskets are documents.

Algorithm 2.3 Charm$(P, minsup, \mathcal{C})$

1: $\mathcal{C} \leftarrow \emptyset$
2: **foreach** $\langle X_i, \boldsymbol{t}(X_i)\rangle \in P$ **do**
3: $\quad P_i \leftarrow \emptyset$
4: $\quad$ **foreach** $\langle X_j, \boldsymbol{t}(X_j)\rangle \in P$, with $j > i$ **do**
5: $\quad\quad N_{ij} = X_i \cup X_j$
6: $\quad\quad \boldsymbol{t}(N_{ij}) = \boldsymbol{t}(X_i) \cap \boldsymbol{t}(X_j)$
7: $\quad\quad$ **if** $sup(N_{ij} \geq minsup$ **then**
8: $\quad\quad\quad$ **if** $\boldsymbol{t}(X_i) = \boldsymbol{t}(X_j)$ **then**
/* Property 1 */
9: $\quad\quad\quad\quad$ Replace X_i with N_{ij} in P and P_i
10: $\quad\quad\quad\quad$ Remove $\langle X_j, \boldsymbol{t}(X_j)\rangle$ from P
11: $\quad\quad\quad$ **else if** $\boldsymbol{t}(X_i) \subset \boldsymbol{t}(X_j)$ **then**
/* Property 2 */
12: $\quad\quad\quad\quad$ Replace X_i with N_{ij} in P and P_i
13: $\quad\quad\quad$ **else**
/* Property 3: $\boldsymbol{t}(X_i) \neq \boldsymbol{t}(X_j)$ */
14: $\quad\quad\quad\quad$ $P_i \leftarrow P_i \cup \{\langle N_{ij}, \boldsymbol{t}(N_{ij})\rangle\}$
15: $\quad$ **if** $P_i \neq \emptyset$ **then**
16: $\quad\quad$ Charm$(P_i, minsup, \mathcal{C})$
17: $\quad$ **if** $\nexists Z \in \mathcal{C}, X_i \subseteq Z and \boldsymbol{t}(X_i) = \boldsymbol{t}(Z)$ **then**
18: $\quad\quad \mathcal{C} = \mathcal{C} \cup X_i$ /* Add X_i to closed set */

- *Plagiarism* by considering documents as items and sentences as baskets.
- Consider *biomarkers*, such as genes or blood proteins, and *diseases* as items. Consider patients as items. Every frequent itemset containing both biomarkers and diseases can suggest a correlation between them and suggest a test for the disease.

A common way of representing the information extracted is in form of if-then rules named *association rules*. We mention here how to extract association rules given the collection of frequent itemsets $\mathcal{F}$.

An *association rule* is an expression $A \overset{s,c}{\Rightarrow} B$, where A and B are itemsets, s and c are real numbers, $s, c \in [0, 1]$, and $A \cap B = \emptyset$. The support s of a rule is defined as $s = sup(A \Rightarrow B) = sup(A \cup B)$.

The *confidence* of a rule is defined as $c = conf(A \Rightarrow B) = \frac{sup(A \cup B)}{sup(A)}$. Observe that if we take A and B as boolean predicates, the strict implication of classical logic corresponds to the case $c = 1$.

The extraction of association rules simply requires, for a given $X \in \mathcal{F}$, to look at all proper subsets $A \subset X$ to compute rules of the form

$$A \overset{s,c}{\Rightarrow} X - A \tag{2.9}$$

Which is surely frequent, since $s = sup(A \cup (X - A)) = sup(X) \geq minsup$. Its confidence is computed as $c = \frac{A \cup (X-A))}{sup(A)} = \frac{sup(X)}{sup(A)}$.

2.4 Data Stream mining

In a digital world that is continuously incrementing the amount of information created day by day, new ways of dealing with this huge information are needed to extract knowledge efficiently. In *machine learning*, and in particular in *data mining* field, this quantities of data were typically processed as large but static datasets. If data cannot fit into memory, smaller training sets were used or algorithms may resort to temporal external storage. But in every case the usual data mining approach does not address the problem of a *continuous* supply of data.

The *data stream* paradigm has recently emerged in response to this problem. The core assumption of this paradigm is that training samples, that arrives in a high speed stream, can be inspected only once and then must be discarded. No control over the data arrival order is possible, and such algorithm must update its model incrementally as each example is inspected.

Data stream mining leads to many new challenges, in particular in terms of

Memory it is unrealistic to keep the entire stream in main memory or in a secondary storage, since a data stream comes continuously and the amount of data is unbounded.

Scans the traditional method of mining with multiple scans is unfeasible. Stream data is processed only once and in a fixed order, determined by the stream arrival pattern.

Speed mining requires fast, real-time processing due to the high data arrival rate and mining results are expected to be available in a very short response time.

Concept-drift online analysis also subsumes the ability to react to concept changes that may occur during the evolution of the data stream.

These points identify which are the key parameters for query processing over data streams, i.e., the amount of *memory* made available to the online

algorithm and the *per-item processing time* required. In order to respect these constraints, we need algorithms that can summarize data streams in a concise and accurate *synopsis* that can be stored in a small amount of memory and can be used to provide *approximate answers* to user queries along with some reasonable guarantees on the quality of the approximation.

As mentioned in the previous sections, frequent itemset mining has been intensively studied on static databases an many efficient algorithms have been proposed.

Recently several emerging application domains needs data to be processed continuously *online*. Examples of *continuous data streams* are Call-Detail-Records in Telecom networks, transactions in retail chains, ATM and credit card operations in banks, sensor networks. Such critical operations like fraud-detection, trend analysis, market analysis, require an efficient *online* analysis to capture interesting trends, patterns and exceptions.

Some interesting examples are shown in [28]. Consider the case of Internet and web traffic. A switch in the middle of the Internet receives streams of IP packets from many inputs and routes them to its outputs. Normally, the job of the switch is to transmit data and not to retain or query it. But switches may be used , e.g, to detect denial-of-service attacks or to reroute packets based on the information about congestion in the network.

Web sites receives different kinds of streams, e.g., the several millions of queries that Google receive daily, or the billions of clicks onto Yahoo! web sites. Many things can be learned from these streams. For example, an increase of queries 'over flu' or 'sore throat' can help tracking the spread of viruses. Or a sudden increase in the click rate for a link could indicate some news connected to that page.

A detailed survey of the state of the art solutions in frequent itemset mining over data streams is provided in Chapter 3 and in the Appendix.

2.4.1 Concept Drift

According to [45], an important problem in mining real data streams is that the concept of interest may change in function of a hidden context. For example, the sets of products that are frequently bought together in a department store may change with time, depending on the current day of the week, the seasons, the availability of alternatives, and many others. Often the cause of the change is hidden, not known a priori.

The kinds of concept drifts that appears in the real word are commonly classified in:

- *sudden*, or *abrupt*, concept drift.

- *gradual* concept drift.

For example, someone graduating from college might suddenly have completely different monetary concerns, whereas a slowly wearing piece of factory equipment might cause a gradual change in the quality of output parts [46].

In Section 5.2 we study in depth the influence of concept drift in frequent itemset mining over data streams, considering both the above types of change.

2.5 Massive Online Analysis (MOA)

Massive **O**nline **A**nalysis (MOA) [4, 5] is a software environment for implementing algorithms and running experiments for online learning from evolving data streams. MOA is related to WEKA, the Waikato Environment for Knowledge Analysis, an open-source workbench containing implementations of a lot of batch machine learning methods [41]. MOA, like WEKA, is written in Java, taking advantage of its portability and the strong and well developed support libraries.

MOA is concerned with the problem of *classification*. Its goal is to produce a model that can predict the class of unlabeled examples, by training on examples whose label (class) is provided. Such functionality is obtained by treating data as a potentially infinite *stream*, instead of the classical *batch* setting, which operates assuming that the training data is available as a whole set.

A classification algorithm must meet several requirements to work and learn from data streams. In particular must fulfill the following four requirements:

1. Process one example at time, and inspect it only once (at most)
2. Use a limited amount of memory
3. Work in a limited amount of time
4. Be ready to predict at any point

These requirements also cover the main issues in data streams mining explained in Section 2.4. MOA provides data streams adaptations of the most common methods already adopted in batch analysis, such as:

- Decision trees

- Rules extraction from decision trees
- Lazy/Nearest neighbor
- Support vector machines/neural networks
- Bayesian methods
- Meta/Ensemble methods

Currently MOA does not provide any method for Frequent Itemset Mining over data streams. Our work intents to extend its functionalities over this crucial mining task. We use the easiness to extend the MOA framework via the MOA A.P.I. to provide a portable, usable and extendable solution for Frequent Closed Itemset Mining.

Chapter 3

Previous works

In this chapter we present several of the state-of-the-art algorithms on mining frequent closed itemsets. This analysis is a key point of our work, because it allows us to know what are the most important approaches that have been adopted in research until now. One of the analyzed solutions was selected and implemented into the MOA environment. We justify our choice, by explaining its advantages in comparison with others approaches. Each algorithm is exhaustively analyzed by identifying its key characteristics, pro and cons, in Appendix A.5, A.6, A.7, A.8 and A.9.

3.1 Preliminaries

Frequent itemset mining in data streams is a relatively new branch of study in data mining. Several different approaches were proposed in the last decade. Most of them can be classified accordingly to the window model they adopt:

- Landmark window
- Sliding window

Each window model can be classified as:

- Time sensitive
- Transaction sensitive

Furthermore, accordingly to the number of transactions that are updated each time, the algorithms can be divided into:

- Update per transaction

- Update per batch

Finally frequent algorithms can subdivided into:

- Exact
- Approximate

Exact mining requires to track all items in the window and their exact frequencies, because any infrequent itemset may become frequent later in the stream. However, because of the combinatorial explosion of number of itemsets, exact mining can be computationally intractable for big windows and data streams with a fast arrival rate. In the majority of cases approximate mining is a more realistic option. In fact, in data mining the goal is to find interesting patterns with reasonable accuracies and efficiency, rather than to provide exact but costly results.

With respect to approximate algorithms, we can identify *false-positive* algorithms if the returned set of itemsets includes all frequent itemsets but also some infrequent one. On the other hand *false-negative* solutions return a set of itemsets that does not include infrequent itemsets but miss some frequent one.

We use here the same notation used to formalize the frequent itemset mining problem in Chapter 2.

- Let $\mathcal{I} = \{x_1, x_2, \ldots, x_m\}$ be a set of *items*. A set $X \subseteq \mathcal{I}$ is an *itemset*. An itemset of size k is called a k-itemset.

- Let $\mathcal{T} = \{t_1, t_2, \ldots, t_m\}$ be a set of tids. A *transaction* is a tuple of the form $\langle t, X \rangle$, where $t \in \mathcal{T}$ is a unique transaction identifier, and X is an itemset. The transaction *supports* an itemset Y if $Y \supseteq X$.

A *transaction data stream* is a sequence of incoming transactions and a excerpt of this stream is called a *window*.

A sliding window W can be either *time-sensitive* or *transaction-sensitive*. A *time-sensitive* window W consists of a sequence of fixed-length time units, where a variable number of transactions arrive within each time unit. The window slides forward for every time unit.

A *transaction-sensitive* window W consists of a sequence of batches, where each batch is made of an equal number $b \geq 1$ of transactions. The window slides forward for every number of transactions equal to the batch size. For example, a time-sensitive window of length 1 hour contains the transactions arrived in the last hour, which may be one, or one million; a

transaction-sensitive window of length 1000 contains the last 1000 transactions, which may have arrived in one second or in one hour.

If the number of transactions that is used in the update procedure is unitary we have an *update per transaction* policy, otherwise we talk of an *update per batch* policy.

A window W is a *landmark window* if $W = \langle T_1, T_2, \ldots, T_\tau \rangle$, where each T_i is a time unit or a batch, T_1 and T_τ are the oldest and the current time unit or batch. A window W is a *sliding window* if $W = \langle T_{\tau-w+1}, \ldots, T_\tau \rangle$, where w the size of the sliding window, which slides forward each time unit or batch.

In Chapter 2 we define the support of an itemset X with respect to a transactions database $\mathcal{D}$. The same definition can be adopted in the analysis of a data stream, considering that the transactions database now is the set of transactions stored within the sliding window W.

Thus we define the *support* of an itemset X in a sliding window W, denoted as $sup(X, W)$ as the set of transactions in W that contains X, i.e.,

$$sup(X, \mathcal{D}) = |\{t_i | \langle t_i, \boldsymbol{i}(t_i) \rangle \in W \text{ and } X \subseteq \boldsymbol{i}(t_i)\}| \tag{3.1}$$

For sake for readiness, from now on we use the concise notation $sup(X)$ to represent $sup(X, W)$, if do not differently stated.

Note that we always assume that there is a lexicographical order among items. We use the notation $X \prec Y$ to denote that item X is lexicographically smaller than the item Y. For example $\{a, b, c\} \prec \{a, b, d, e\} \prec \{a, c, e\} \prec \{d\}$.

We use the characteristics mentioned above to analyze several of the state-of-the-art algorithms in Frequent Itemset Mining that have been proposed by the scientific community since the time of writing. In particular we focus on Frequent Closed Itemset Mining over data streams solutions.

3.2 A general Incremental Closed Pattern Mining solution

Bifet et al. propose in [14] a general methodology to identify closed patterns in a data stream. It uses the Galois Lattice Theory to perform the mining process and can be applied to different kinds of patterns, such as trees, graphs and itemsets.

It defines two procedures to add or remove a set of frequent closed patterns that have been computed previously over a set of transactions that have been added or removed from a sliding window. To do this efficiently,

it exploits several properties pattern datasets. It can be seen as a general representation of every incremental algorithm for frequent closed itemset mining that we will analyze.

A more detailed explanation of this work is presented in A.5.

This is a general framework for pattern mining, and no specialization over itemset mining is provided. It defines a high-level solution for mining any kind of pattern, without considering the peculiarities of any of them. This may results in inefficient processing, since no low-level detail over data structures or efficiency of the procedures is provided.

Since we are interested in really efficient solution for mining frequent closed itemsets, we will not implement this solution here. But it has been useful to consider the capabilities of updating via batches of transactions.

3.3 MOMENT

MOMENT was proposed by Chi et al. in [13]. It is the first work that have been proposed for *incremental* mining of closed frequent itemset over a data streams sliding window. It perform an *exact* mining of the set of frequent closed itemsets, using an *update per transaction* policy to keep this set up-to-date.

To monitor a *dynamically selected* set of itemsets over the sliding window, MOMENT adopts an in-memory prefix-tree-based data structure, called *closed enumeration tree* (*CET*). This tree stores information about infrequent nodes, nodes that are likely to become frequent and closed nodes. MOMENT also uses a variant of the FP-tree, proposed by Han et al. in [3], to store the information of *all* the transactions in the sliding window, with no pruning of infrequent itemsets.

MOMENT algorithm essentially performs a depth-first search along the CET, and updates the type of each node whether it is necessary. Concept drifts cause variations in the type of *boundary* nodes, such as infrequent nodes that become frequent and viceversa.

Since it is the first proposed work that considers mining of frequent closed itemsets over a data stream with limited memory consumption, MOMENT has become a reference for all the solutions have been proposed successively.

A detailed explanation of MOMENT data structures and procedures is presented in Appendix A.6.

3.4 CLOSTREAM

CLOSTREAM is an algorithm for maintaining *frequent closed itemsets* in data stream. It was proposed by Yen et al. in [7]. CLOSTREAM maintains the *complete set* of closed itemset over a emphtransaction-sensitive sliding window *without any support information*.

It uses two in-memory data structures, the *Closed Table* and the *Cid List*, to maintain information about closed itemset into the sliding window. CLOSTREAM uses an *update per transaction* policy. Update is performed by two procedures *CloStream+* and *CloStream-*, respectively used when a transaction arrives and when a transaction leaves the sliding window. Both procedures use two temp hash tables to perform an efficient update.

CLOSTREAM does not directly handle concept drift, since every closed itemset into the current sliding window is mined. Like MOMENT, it offers an *exact* solution to frequent closed itemset mining problem.

A detailed explanation of CLOSTREAM data structures and procedures is presented in Appendix A.7.

3.5 NEWMOMENT

NEWMOMENT is method to maintain *frequent closed itemsets* in data streams with a *transaction-sensitive* sliding window proposed by Li et al. in [11]. It uses an effective *bit-sequence* representation to reduce time and memory consumption of the MOMENT algorithm. It also define a new closed enumeration tree (*NewCET*), to store *only* the set of frequent closed itemsets into the sliding window.

NEWMOMENT inherits most of the characteristics of MOMENT algorithm, such as the *update per transaction* policy and the *exactness* of the mining procedures.

Although, the usage of bit-sequences results into simplified update procedures respect to MOMENT's ones. For example, window sliding can be performed efficiently via a *left-shift* of one bit of each bit-sequence stored. The several intersection operations can be converted into efficient bitwise AND operations.

A detailed explanation of NEWMOMENT data structures and procedures is presented in Appendix A.8.

3.6 INCMINE

INCMINE is an algorithm for incremental update of *frequent closed itemsets*(*FCIs*) over a high-speed data stream proposed by Cheng et al. in [6].

They propose an *approximate* solution to the problem, using a *relaxed minimal support threshold* to keep an extra set of infrequent itemsets that likely can become frequent later, and using an *inverted index* to facilitate the update process. They also propose the novel notion of *semi-FCIs*, which associate a progressively increasing minimal support threshold for an itemset that is retained longer in the window.

It uses an *update per batch* policy to maintain the updated the set of approximated frequent closed itemsets over the current sliding window. The original proposal considers *time-sensitive* sliding windows, but it can be easily adapted to *transaction-sensitive* contexts with fixed-length batches.

The *incremental update* algorithm exploits the properties of Semi-FCIs to perform an efficient update in terms of memory and timing consumption. Semi frequent closed itemsets are stored into several *FCI-arrays*, which are efficiently addressed by an *Inverted FCI Index*.

Since INCMINE is the algorithm we are going to implement, we analyze it in detail in Chapter 4.

3.7 CLAIM

CLAIM is an algorithm for Closed Approximate frequent Itemset Mining proposed by Song et al. in [10]. It perform an *approximate* mining of the set of frequent closed itemsets into a *transaction-sensitive* sliding window.

To do this, the authors defines the concept of *relaxed interval* and *relaxed closed itemset*, with the intention to reduce the maintenance cost of drifted closed itemsets in a data stream. CLAIM uses a *double bound* representation to manage the itemsets in each relaxed interval, which is efficiently addressed by several *bipartite graphs*. Such bipartite graph is arranged using a *HR-tree* (Hash based Relaxed Closed Itemset tree), which combines the characteristics of a *hash table* and a *prefix tree*.

A detailed explanation of CLAIM data structures and procedures is presented in Appendix A.9.

3.8 Comparison of algorithms and conclusions

In the previous sections we present several of the state-of-the-art algorithms in Frequent Closed Itemset Mining over data stream. We do not consider

here any solution over Frequent Itemset Mining, e.g, presented in [12, 15, 20, 19, 21, 22], and over Frequent Maximal Itemsets over data streams, e.g, presented in [23]. This is because Frequent Closed Itemsets allows a *complete* and *non-redundant* representation of the set of Frequent Itemsets. The former representation is often orders of magnitude smaller that the latter. What we obtain is a lighter representation and faster processing times, which are crucial aspects in data streams processing.

In order to better cope with *concept-drift*, all the analyzed solutions use a sliding window approach. Landmark window approaches were proposed in [21, 24], but cannot handle concept changes into a data stream as a sliding window approach can do, because the landmark window considers all the data since the start of an epoch, regardless of whether drift has occurred.

In Table 3.1 are illustrated the salient characteristics of each analyzed solution.

MOMENT was the first solution that considers Frequent Closed Itemsets as a possible compact representation of Frequent Itemsets over data streams. It uses a prefix tree based data structure (the Closed Enumeration Tree) to keep updated the set of closed itemsets and the so-called *border* itemsets. Beside the CET a modified FP-tree is used to maintain all transactions in the sliding window.

The main problem of MOMENT is hidden in these data structures. The information of *all* the transactions is stored in the modified FP-tree, with a considerable overhead in memory. The nodes in the CET do not refer only to closed itemsets, but also to non-closed and unfrequent itemsets, and also this contributes to an higher usage of memory. MOMENT algorithm is also written taking in account that nodes change type rarely, i.e., the concept does not change 'too much' along the stream. This assumption translates into higher execution time when the number of nodes that changes type is high, for example in case of sudden drifts.

NEWMOMENT tries to solve some of problems of MOMENT introducing a more efficient representation of transactions and itemsets, via bit-sequences. The window sliding can be done efficiently via left-shift, and itemsets support counting is reduced to a bitwise AND operation. The NewCET stores only the closed itemsets in the closed window, and no transactions tracking is performed, with a consistent memory saving. The experiments in [11] show improvements in performance.

CLOSTREAM uses a totally different approach from the ones above. It develops the idea, firstly proposed in [18] for CFISTREAM algorithm, that a frequent itemset mining algorithm does not need a minimum support threshold. Actually MOMENT and NEWMOMENT become as more ineffi-

cient as the minimum support threshold decreases, because of the explosion of the number of nodes in their prefix-trees. CLOSTREAM maintains the set of *all* frequent closed itemsets within the sliding window, performing an efficient update per transaction.

All previous algorithms provide an *exact* solution, but the combinatorial explosion of itemsets can affect significatively performances in terms of memory consumption and processing efficiency. For example, CLOSTREAM maintains the whole set of closed itemsets, that can be really huge and unmanageable in real cases. MOMENT and NEWMOMENT are based on a prefix-tree structure, which is not efficient for some update operations such as the search of the smallest proper closed superset of an itemset.

It is now commonly accepted that *approximate* solutions can almost completely overcome these problems. Such algorithm can provide an approximation of the set of frequent (closed) itemsets, within an error bound, and achieve significantly better performances than exact algorithms. The concept below is that most applications will not need precise support information of frequent patterns, a good approximation on support count could be more than accurate [25]. And in itemset mining over data streams this becomes even more true, because fast, real-time processing is required in order to keep up with the high data arrival rate and responses are expect to be available within very short response time.

INCMINE proposes to mine a new kind of closed itemset, named semi-Frequent Closed Itemsets, where the minimum support threshold increases for an itemset as it is retained longer in the window. A relaxed minimum support threshold mines extra infrequent itemsets that have a high potential to become frequent later. An inverted index, an efficient data structure also used in different applications such as information retrieval [26], is used to facilitate the update process. The resulting solution is a false-negative approximation of the real set of frequent closed itemsets in the sliding window.

The update process is done for each batch of transactions, which may significantly increase the processing speed of the algorithm. According to Li et al. in [27], update transaction-by-transaction leads to a huge amount of processing because update of transactions is excessively frequent. And usually the transit of transactions in a data stream is at high speed, and the contribution of one single transaction to the entire set of transactions is negligible. Thus, an update per batch policy can overcome to this problem.

Finally CLAIM proposes to group frequent closed itemsets into relaxed support intervals. The intention is to reduce the effect of small concept drift that always appear along a data stream. In exact algorithms the support of closed itemsets exactly equals to the absorbed itemsets, and slightly support

differences can lead to high costs of maintenance. They intent to reduce these side-effects by dividing the support space into several subsets and redefining the closure concept. The update process is done for each incoming transaction, and drifts are managed by a bipartite graph representation along a prefix-tree structure.

We decide to concentrate our efforts on INCMINE algorithm. It offers a distinct perspective from other solutions, and is one of the few algorithms that considers the update for batches of transactions. This point, together with the possibility of personalizing the minimum support function and the inverted index structure, which promises better performances than tree structures, convinces us to study it in depth and implement it into the MOA environment.

Algorithm	**Window model**	**Update policy**	**Solution type**	**Data structures**	**Notes**
Moment	Transaction-sens.	Per transaction	Exact	CET FP-tree+Tids Hash table for closure check	
CloStream	Transaction-sens.	Per transaction	Exact	Closed Table CidList Temp hash tables	No minimum support threshold
NewMoment	Transaction-sens.	Per transaction	Exact	NewCET Bit-sequences Hash table for closure check	
IncMine	Time-sens.	Per batch	Approximate (*false-negative*)	Inverted Index FCI-Array	Semi-FCI Increasing relaxed minimum support threshold
Claim	Transaction-sens.	Per transaction	Approximate	Bipartite graph HR-tree	Relaxed interval Relaxed FCI Bloom filter based hash

Table 3.1: Main characteristics of the analyzed algorithms.

Chapter 4

Architecture and implementation of the proposed solution

Until now we have discussed the principles for frequent itemset mining on static datasets and what are the new requirements in analyzing streams of data. Then we analyzed several algorithms dealing with frequent closed itemset mining over data streams. We have identified the salient characteristics of each one, by studying the efficiency and the impact of the data structures and procedures used by each one.

It is commonly known that frequent itemset mining is a 'hard' task (in computational perspective) and performing it over high speed data streams is even harder, due to the really strict constrains in memory consumption and execution time that presents such online processing.

In addition, the requirements and features of a data stream can vary a lot from context to context, for example the data arrival rate, the amount of memory available, the type of data that one have to manage (e.g., data from sensors in a sensor network, click stream over a web page, films titles from a film rental, etc.) which can affect, for example, the density of the transactions passed to the algorithm. Therefore, it is really difficult to find a solution that adapts to every possible environment without knowing it 'a priori'. A good solution may offer to an expert user the possibility to tune correctly the algorithm, in order to obtain the best response in function of the characteristics of the data stream.

Our intention is to provide an portable, usable and extendable solution for frequent closed itemset mining. So we decide to use the functionalities of the MOA framework and extend its A.P.I. by implementing a brand new

package for itemset mining over data streams. It is implemented in Java, as the whole framework, to assure the portability of our solution. Its usability is guaranteed by the A.P.I. itself and by the MOA GUI. No particular environment of work is defined, so any kind of stream of itemset that can by passed via the MOA A.P.I. functions will work correctly.

We now show the main characteristics of our INCMINE implementation into the MOA Framework. For sake of readability only few code snippets will be presented, and they will focus only over the peculiar characteristics of our solution.

4.1 Overview

We implement a Java version of the algorithm INCMINE, presented in Section 4.2, within the MOA environment. We decide to use a *transaction-sensitive* approach instead of the *time-sensitive* approach that is proposed originally in [6], mainly to ease our testing, although it is not an essential choice. In fact is easier to evaluate performances when the number of instances that are elaborate each window sliding is a fixed number. This is because currently is not possible to define in MOA a data arrival rate of transactions in the stream, that is simply simulated passing transactions at once into a `while` cycle.

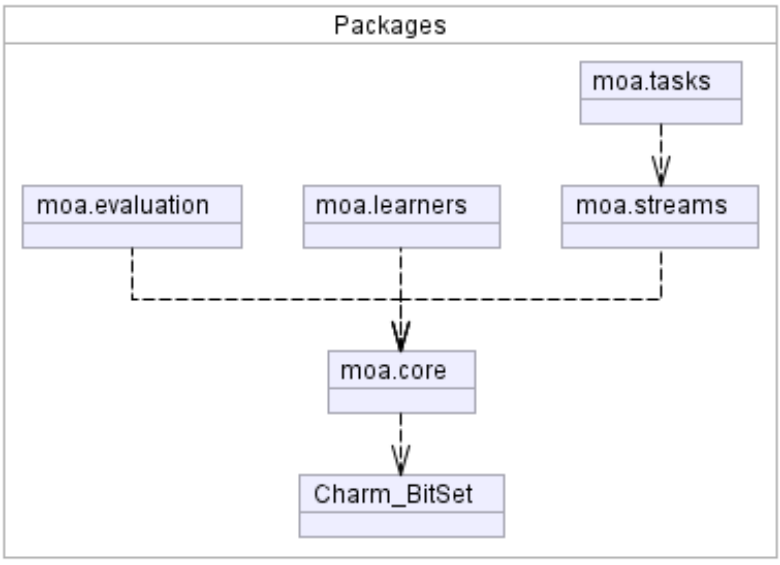


Figure 4.1: Packages of our solution.

We organize our solution into several packages, that extends some of

ones that are already defined into the MOA A.P.I. According to Figure 4.1, 6 different packages are defined

- Package `moa.task` contains classes `LearnModel` and `LearnEvaluateModel`, inherited from class `MainTask` of the MOA A.P.I.. These classes define methods to compute a generic model over a data stream and evaluate its performances.
- Package `moa.streams` contains class `ZakiFileStream`. It is a stream reader specifically written for files generated via the Zaki's IBM Generator software.
- Package `moa.learners` contains class `IncMine`, which inherits from class `AbstractLearner`, an abstract class provided by the MOA A.P.I. to define new learners within the framework. It is the core class of our INCMINE version, contains the real implementation of the algorithm.
- Package `moa.evaluation` contains class `LearningEvaluation`, which defines the functionalities needed by class `LearnEvaluateModel` to evaluate performances.
- Package `moa.core` contains several key classes for the INCMINE algorithm. Classes `FCITable` and `InvertedFCIIndex` are the implementation of the respective data structures used by the algorithm, that we already described in Appendix 4.4.

  Classes `SemiFCI` and `SemiFCIId` defines Semi-Frequent Closed Itemsets and respective IDs.

  Abstract class `SlidingWindowManager` defines a common interface for managing sliding windows, either time-sensitive or transaction-sensitive. As commented before, we decided to use a transaction-sensitive approach. We define class `FixedLengthWindowManager`, which inherits from this abstract class, that manages fixed-length batches of transactions.

  Class `Segment` defines a segment, i.e, a batch of transaction. It collects all the transactions seen so far. When its size reaches its maximum value (i.e., the fixed length of each segment), it computes the current set of Frequent Closed Itemsets, passes it to the update procedure and, when the update process is finished, it becomes ready to accept new transactions.

  Class `FrequentItemset` is an auxiliary class, not strictly needed by the algorithm, that we use to compute the set of Frequent Itemsets from the set of approximated Frequent Closed Itemsets that are computed.

Class `Utilities` defines several `static` methods that are used in different points of our solution, such as, compute the intersection of 2 ordered lists, to compute the cumulative sum of a vector until a specified position, to compute the minimum support vector, to compute the memory usage in a certain point of the execution, and many others.

- Package `Charm_BitSet` defines all classes needed to compute the set of Frequent Closed Itemsets over a transactions database using the CHARM algorithm [31, 32]. This is a modified version that uses bitsets to represent Tidsets to achieve better performances. We slightly modified the version of this algorithm that is kindly provided under GPL3 Licence by Philippe Fournier-Viger in [35].

A full `Javadoc` documentation comes along with the MOA A.P.I. extension that we have developed.

4.2 The INCMINE algorithm

INCMINE is an algorithm for incremental update of *frequent closed itemsets*(*FCIs*) over a high-speed data stream proposed by Cheng et al. in [6]. They propose an *approximate* solution to the problem, using a *relaxed minimal support threshold* to keep an extra set of infrequent itemsets that likely can become frequent later, and using an *inverted index* to facilitate the update process. They also propose the novel notion of *semi-FCIs*, which associate a progressively increasing minimal support threshold for an itemset that is retained longer in the window.

The original proposal of Cheng et al. uses a *time based sliding window*. A *window* or a *time interval* in the stream is a set of successive time units, denoted as $T = \langle t_i, \ldots, t_j \rangle$, where $i \leq j$, and $T = t_i$ if $i = j$. The window slides forward for every time unit. Thus, the *current window* is $W = \langle t_{\tau-w+1}, \ldots, t_\tau \rangle$, where w is the number of time units in W (the *size* of W).

Define $trans(T)$ as the set of transactions that arrive on the stream in a time interval T and $|trans(T)|$ as the number of transactions in T. The *support* of an itemset X over T is the number of transactions in T that support X, and its denoted as $sup(X, T)$.

Given a *Minimum Support Threshold*(*MST*), $\sigma(0 < \sigma \leq 1)$, the itemset X is a Frequent Itemset (FI) over T if $sup(X, T) \geq \sigma|trans(T)|$. X is a Frequent Closed Itemset (FCI) over T if X is a FI over T and there exist no Y such that $Y \supset X$ and $sup(Y, T) = sup(X, T)$. If $X \supset Z$ and $sup(X, T) = sup(Z, T)$, where X is a FCI over T, the relationship between

X and Z is denoted as $X \sqsupset^T Z$.

t_1	t_2	t_3
abcd	abc	abcd
abcx	abcd	az
bcy		abc

Table 4.1: Example of transactions in a stream of 3 time units

Table 4.1 records the transactions in the stream in two successive windows, $W_1 = \langle t_1, t_2 \rangle$ adn $W_2 = \langle t_2, t_3 \rangle$. For a minimum support threshold of 2, the set of FCIs in W_1 and W_2 is $\{abcd, abc, bc\}$ and $\{abcd, abc, a\}$ respectively. Note that bc is not a FCI over W_2 since $bc \sqsupset^{W_2} abc$.

4.2.1 Semi-Frequent Closed Itemsets

We want to keep track of infrequent itemsets that may become frequent later in the stream. Because of the exponential explosion of subsets, its infeasible to keep all infrequent itemsets. An usual approach is to use an *error parameter* ϵ, $0 \leq \epsilon \leq 1$, to maintain itemsets in the window as long as their support is at least $\epsilon|W|$. In INCMINE is used an improved version of this principle, by considering ϵ as a *relaxed MST* and *progressively increasing* it for an itemset that is retained longer in the window.
Define the relaxed MST $\epsilon = r\sigma$, where $r(0 \leq r \leq 1)$ is the *relaxation rate*. All itemsets whose support is less than $r\sigma|trans(t)|$ are discarded.

The *approximate support* of an itemset X over a time unit t is defined as

$$\widetilde{sup}(X,t) = \begin{cases} 0 & \text{if } sup(X,t) < r\sigma|trans(t)| \\ sup(X,t) & otherwise. \end{cases} \tag{4.1}$$

The approximate support of X over a time interval $T = \langle t_j, \ldots, t_k \rangle$ is defined as

$$\widetilde{sup}(X,T) = \sum_{i=j}^{k} \widetilde{sup}(X,t_i) \tag{4.2}$$

Let $W = \langle t_{\tau-w+1}, \ldots, t_\tau \rangle$ be a sliding window of size w and $T^k = \langle t_{\tau-k+1}, \ldots, t_\tau$ be the most recent k time units in W, where $1 \leq k \leq w$.
Lets define a *progressively increasing MST function*, $minsup(k)$, as follows:

$$minsup(k) = \lceil m_k \times r_k \rceil \tag{4.3}$$

where $m_k = \sigma|trans(T^k)|$ and $r_k = (\frac{1-r}{w})(k-1) + r$. The term m_k is the minimum support required of a FI over T^k, while the term r_k progressively increases the relaxed MST at the rate of $(\frac{1-r}{w})$ for each older time unit in the window. An itemset is kept in the window only if its support over T^k is no less than $minsup(k)$ for some $k \in \{1, \ldots, w\}$. Note that $minsup(k)$ is a *non-decreasing* function that always assumes values in $0 \leq minsup(k) \leq \sigma|trans(T^k)|$.

k	10	9	8	7	6	5	4	3	2	1
$minsup(k)$	182	148	117	90	66	46	30	17	8	2

Table 4.2: Example of $minsup(k)$ function for stream with an uniform input rate of 2000 trans./time unit ($\sigma = 0.01$, $r = 0.1$, $w = 10$).

Time Unit	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}
$sup(ab, t_i)$	3	1	2	3	2	1	4	7	11	19	21
$sup(cd, t_i)$	3	11	20	29	11	8	17	28	37	41	39

Table 4.3: Example of SemiFCIs along with their supports for 11 time units.

Finally, an itemset X is a *semi-frequent itemset* (*semi-FI*) over W if $\exists k$ such that $\widetilde{sup}(X, T^k) \geq minsup(k)$. An semi-FI X is a *semi-frequent closed itemset* if $\nexists Y \supset X$ such that $\widetilde{sup}(Y, T^k) = \widetilde{sup}(X, T^k)$. X is called a k-semiFCI if X is a semi-FCI and $k = MAX\{k : \widetilde{sup}(Y, T^k) \geq minsup(k)\}$. Table 4.2 and Table 4.3 shows an example of $minsup(k)$ function and two sample semi-FCIs with different behavior. Itemset ab has an *unpromising* support over $\langle t_1, \ldots, t_6 \rangle$ and, thus, is discarded. Although the trend in the support from t_7 to t_{11} shows that ab may become frequent, thus ab is stored in the window only after t_7. On the other hand, the support of the itemset cd is always greater than the corresponding $minsup(k)$, thus cd is always maintained in the window.

From the above definitions comes that INCMINE provides an *approximate solution* to the Frequent Closed Itemset mining problem. In particular, this method is a *false-negative* approach. The set of false-negatives is defined as $\{X|(\widetilde{sup}(X, W) < \sigma|trans(W)|) \wedge (sup(X, W) \geq \sigma|trans(W)|)\}$. False-negatives are mostly itemsets with skewed support distribution over the stream.

4.2.2 The incremental update algorithm

INCMINE incrementally updates the set of semi-FCIs over a sliding window with an *update per batch* policy. Let t_τ be the current time unit, $W_L = \langle t_{\tau-w}, \ldots, t_{\tau-1} \rangle$ and $W_C = \langle t_{\tau-w+1}, \ldots, t_\tau \rangle$ be the *last* window and the *current* window. Also name as F,L and C the set of semi-FCIs over t_τ, W_L and W_C, respectively.
The task consist in *incrementally update the set of semi-FCIs over a sliding window* by updating L with F to give C. Note that F is generated with a relaxed MST $r\sigma$ by an existing *non-streaming* FCI mining algorithm.

The pseudo-code of the incremental algorithm is shown in Algorithm 4.1. It exploits different properties of semi-FCIs in order to obtain a solution efficiently. In detail those properties are (no proof is provided here)

Lemma 1 For any semi-FI X that is not a semi-FCI over T, there exists an *unique* semi-FCI Y such that $X \subset^T Y$ and the size of Y is the smallest among all semi-FCIs that are supersets of X. Itemset Y is called the *Smallest semi-FCI Superset*(*SFS*) of X.

Theorem 1 Given a semi-FCI X over t_τ

1. If $X \in F$, then $X \in C$.
2. If $X \notin F$ and $X \in L$, then $X \in C$ if and only if $\exists Y \in F$ such that $X \sqsubset^{t_\tau} Y$; $\exists k \in \{2, \ldots, w\}$ such that $\widetilde{sup}(X, T^k) \geq minsup(k)$; and $\forall X'$, where $X \subset X' \subseteq Y$, $\widetilde{sup}(X, T^k) \geq \widetilde{sup}(X', T^k)$.
3. If $X \notin F$ and $X \notin L$, then $X \in C$ if and only if $\exists Y \in F \backslash L$ such that $X \sqsubset^{t_\tau} Y$; $\exists Z \in L$ such that $X \sqsubset^{W_L} Z$ and $Y \cap Z = X$; and $\exists k \in \{2, \ldots, w\}$ such that $\widetilde{sup}(X, T^k) \geq minsup(k)$.

Theorem 2 Given X and Y, where $X \sqsubset^{t_\tau} Y$, $X \notin L$ and $Y \notin L$. If $\exists Z \in L$ such that $X \subset Z \subset Y$, then $X \notin C$. It follows that, after processing a subset Z of $Y \in F \backslash L$, if $Z \in L$, we can skip processing all $X \subset Z$ where $X \notin L$.

From Theorem 1(1), all semi-FCIs in F are in C and, hence, they are added to L (Line 11). L is returned as C at the end of update (Line 29). Any subset X of $Y \in F$, where $X \notin L$, is in C if and only if $Y \notin L$, by Theorem 1(3). So, when $Y \in L$, only its subsets that are in L must be processed (Lines 2-9), while its subsets that are not in L are processed only when $Y \notin L$ (Lines 11-23).
When a subset of Y has been updated previously, all its subsets do not have to be processed (Lines 6-7, 17-18); this because the procedure processes

Algorithm 4.1 IncMine(F,L)

```
1: foreach Y ∈ F in size-ascending order do
2:     if Y ∈ L then
3:         ComputeK(Y, 1)
4:         foreach X ⊂ Y in size-descending order do
5:             if X ∈ L then
6:                 if X is updated then
7:                     skip processing X and all its subsets
8:                 else
9:                     UpdateSubsetInL(X, Y)
10:    else/* Y ∉ L */
11:        L ← L ∪ {Y}
12:        if ∃Z ∈ L such that Z ⊐^{W_L} Y then
13:            sup~(Y, t_i) ← sup~(Z, t_i), for τ − w + 1 ≤ i ≤ τ − 1
14:        ComputeK(Y, 1)
15:        foreach X ⊂ Y in size-descending order do
16:            if X ∈ L then
17:                if X is updated then
18:                    skip processing X and all its subsets
19:                else
20:                    UpdateSubsetInL(X, Y)
21:                    skip processing all subsets of X that are not in L
22:            else/* X ∉ L */
23:                UpdateSubsetNotInL(X, Y)
24: foreach X ∈ L in size-descending order dos
25:    if X is not updated then
26:        k = ComputeK(X, 1)
27:        if k > 0 and ∃Z ∈ L such that Z ⊐^{W_C} X then
28:            L ← L − {X}
29: return C ← L
```

semi-FCIs in *size-ascending order*, and any itemset is *first* updated by its SFS or itself (**Lemma 2**). By Theorem 2, after processing a subset $X \in L$, all its subsets that are not in L must be skipped (Line 21), and only X's subsets in L are processed (Line 2-9).

4.3 CHARM for Frequent Closed Itemset mining

A crucial point in the INCMINE processing is the mining of frequent closed itemsets from the current segment of transactions. This could be a very computationally intensive phase, depending on the characteristics of the data stream and on the minimum support threshold and relaxation rate that have been chosen. Thus, the efficiency of the adopted procedure for frequent closed itemset mining could have a significant influence over the global efficiency of the algorithm.

The literature offers several algorithms for this task, from the CHARM algorithm that we present in Section 2.3, to the DCI-Closed algorithm, proposed by Lucchese et al. in [39]. A survey over different algorithms with an interesting comparison over their performances can be found in [40].

Since neither the MOA framework nor WEKA offer any solution for this task, it became necessary to introduce a Java implementation of such frequent closed itemset mining to guarantee the correct working of the algorithm. Thus, we decide to integrate some of the algorithms that are available into the Sequential Frequent Pattern Mining framework [35]. This framework is essentially a collection of implementations of several algorithms over sequential pattern mining, frequent itemset mining and association rule mining, and some others. Among them there is available an implementations of the CHARM algorithm and an improved version of the same algorithm which uses bit sets to represent transactions.

We adapt both algorithms to work within our environment. After several testing phases, that we do not detail here, we decide to adopt the bit set version of Charm, which guarantees significantly better performances. This version of the Charm algorithm is included in the package `Charm_Bitset` that we mentioned in Section 4.1.

4.4 Data structures in depth

INCMINE uses an *Inverted Index Structure* to manage efficiently all the semi-FCIs stored in the sliding window. The set L is partitioned accordingly to the size of the semi-FCIs in the last window W_L. Each partition is stored in an array, called *FCI-array*, and each semi-FCI in the FCI-array is assigned an *ID*, which corresponds to its position in the array, and its approximate support. An array containing semi-FCIs of size n is named a *size-n FCI-array*.

To each size-n FCI-array is associated a *garbage queue*. When a semiFCI is deleted from an FCI-array, its ID is pushed into the garbage queue. When

a new semiFCI have to be inserted into a FCI-array, its ID (position) is popped out from the garbage queue. If the garbage queue is empty, then the new semiFCI is appended to the array. Table 4.4 shows an example of FCI-arrays.

Along with the set of FCI-array, an inverted index, called *Inverted FCI Index*(*IFI*), is used. Its components are

- An *Item Array*(*IA*), which stores all items in $\mathcal{I}$ in lexicographical order.
- Each item in the IA is associated with a list of variable-length arrays called *ID-arrays*. Each ID-array stores the IDs of size-n semiFCIs in ascending order of their integral values (a *size-n ID-array*).

For each semiFCI $X = \{x_1, \ldots, x_n\}$ in L, its ID is stored in the size-n ID-array of each item x_i in the Item Array, $0 \leq i \leq n$. An example of Inverted FCI Index is shown in Figure 4.2.

ID	**Size-1**	**Size-2**	**Size-3**	**Size-4**
0	x	xy	xyz	bxyz
1	y	xz	bxy	abcd
2	b	bx	bxz	
3	g	by	abd	
4		bc		
5		bd		

Table 4.4: Table composed by 4 FCI-arrays

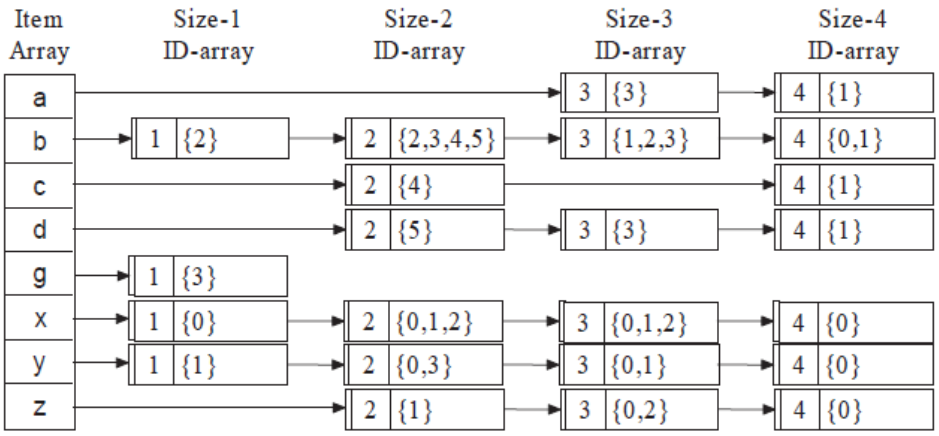


Figure 4.2: Example of Inverted FCI index.

We want to discuss here one point that we found to be crucial for the efficiency of the final algorithm. In particular with the Inverted FCI Index we can, given an itemset X, efficiently get its position into the corresponding FCI-array, we can select its Smallest Semi-FCI Superset (SFS), and obviously we can insert and delete it. Algorithms 4.2, 4.3, 4.4, 4.5 show in detail these operations.

Algorithm 4.2 Select($X = \{x_1, \ldots, x_n\}$)

1: locate x_i, for $1 \leq i \leq n$, in the Item Array
2: perform *join* on the size-n ID-arrays of all x_i
3: return the join result, if any, as the ID of X

Algorithm 4.3 SelectSFS($X = \{x_1, \ldots, x_n\}$)

1: $j \leftarrow 1$
2: *join* the size-$(n + j)$ ID-arrays of all x_i, for $1 \leq i \leq n$
3: increment j until the join obtains a result for some size-$(n+j)$ ID-arrays or terminate the join when the end of the list of ID-arrays of some x_i is reached
4: return the join result, if any, as the ID of X's SFS

Algorithm 4.4 Insert($X = \{x_1, \ldots, x_n\}$)

1: **if** the size-n garbage-queue is empty **then**
2: store X at the end of the size-n FCI-array
3: **else**
4: $ID \leftarrow POP$(size-n garbage-queue)
5: store X in Position ID of the size-n FCI-array
6: insert ID in the size-n ID-arrays of all x_i, for $1 \leq i \leq n$

Algorithm 4.5 Delete($X = \{x_1, \ldots, x_n\}$)

1: push X's ID, i.e., its position in the size-n FCI-array, into the size-n garbage-queue
2: delete X from the size-n FCI-array
3: delete X's ID from the size-n ID-arrays of all x_i, for $1 \leq i \leq n$

We can see how the ID of each size-n FCI-array is duplicated n times, but its memory consumption is not too large since the size of that are stored usually is not big. The efficiency of the inverted indexing comes by the

joining of sorted arrays, which is simple and fast. But in case of several selections, its overhead becomes significative, so it needs to be implemented in such an efficient way.

4.5 Efficient inverted indexing

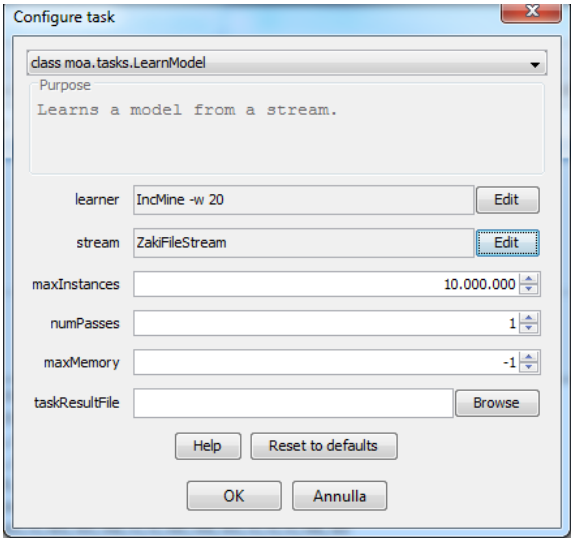
Another point that is not well specified in the original algorithm is how to perform the join operation among the sorted arrays into the inverted index. This operation has a significant influence over the processing performance, since it have to be performed several times for each Semi-FCI that have to be added, deleted or updated. Thus, it is necessary to implement this procedure as much efficiently as possible.

Culpepper et al. in [26] provides a survey over algorithms for efficient multiple sets intersection for inverted indexing. Actually there is a huge variety of approaches that can be adopted to intersect two or more ordered sets. One of those, which is proved to be an efficient solution in several cases, is named *Small Versus Small* (*svs*), and it is the one we adopt in our solution. Essentially, the intersection is computed by proceeding from smallest to largest set. This tends to produce the smallest intermediate results, therefore to be the most efficient processing order.

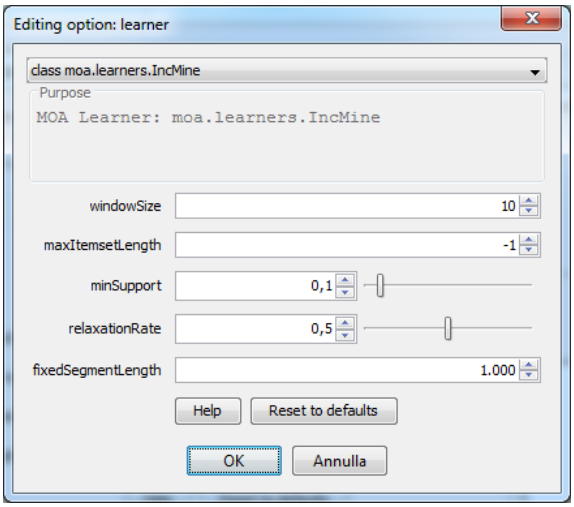
Algorithm 4.6 Small Versus Small intersection

INPUT: A set of n sets $S_1, \ldots, S_n$
OUTPUT: An ordered set of answers A
1: Reorder sets $S_i, 1 \leq i \leq n$ such that $|S_1| \leq \ldots \leq |S_n|$
2: $A \leftarrow S_1$
3: **for** $i = 2$ to n **do**
4: $\quad A \leftarrow Intersect(A, S_i)$
5: **return** A

The function *Intersect* is a performed as a common sequential merge, picking out common elements to the two sets via a loop in which a single comparison is made at each iteration, and depending on the result of the comparison, the successive element of one of the two lists is considered. This implementation results in an important increase in performances respect to the trivial solutions adopted in the early development stages.



(a) LearnModel configuration.



(b) IncMine configuration.

Figure 4.3: Edit settings within the MOA GUI.

4.6 IncMine within the MOA environment

Let us now see how IncMine can be used within the MOA environment. MOA allows two run task either in command line or via its GUI. In order to run MOA with our frequent itemset mining extension, the library `IncMine.jar` must be included into the library path of MOA. For example, if we want to lunch the MOA GUI with our library simply have to edit the file `moa.bat` in the following way:

```
java -cp IncMine.jar;moa.jar -Javaagent:sizeofag.jar moa.gui.GUI
```

This command will start the graphic interface of the MOA environment. To run IncMine we have to use the Classification tab, and select task `moa.task.LearnModel`, or `moa.task.LearnEvaluateModel` if we also want performance evaluation.

Figure 4.3a shows an screen sample of `moa.task.LearnModel` configuration. We can set here the type of learner, the stream in input, the maximum number of transactions to be processed, the number of times the stream have to be passed (usually set to 1) and the maximum amount of memory desired. Figure 4.3b shows how the properties of IncMine algorithm can be set via user interface.

We can configure the learner and launch MOA also via command line. Consider the following example:

```
java -cp ncMine.jar;moa.jar -Javaagent:sizeofag.jar moa.DoTask ‘‘LearnModel
-m 100000 -l (IncMine -w 20 -m 5 -s 0.05 -r 0.4 -l 5000) -s (ZakiFileStream
-f T40I10D100K.ascii)’’
```

It runs the `LearnModel` procedure via a call to `moa.DoTask`. Each option defined into a MOA class has a letter associated. The syntax to assign a value to an option is *ClassName [-opt value]*. Table 4.5 shows the options that are defined in the classes of our project.

4.7 Example of client class

Let us now analyze a simple Java code that shows how to use the salient functionalities of our work (Listing 4.1). We do not use classes `LearnModel` or `LearnEvaluateModel` in this sample code, but the way we analyze the stream is essentially the same that these classes do.

Firstly a new `ZakiStreamReader` is created and instantiated. It reads transactions from file `T40I10D100K.ascii` and provides an *iterator-like* interface to get them. While the stream has more instances (condition that can be checked via the method `hasMoreInstances()`), one can get the next transaction by calling method `nextInstance()` of the stream object.

In line 9 we create and instantiate the IncMine learner object. Then we configure the settings of the mining procedure (Lines 12-17). In MOA this is done by providing a value to some public options that are defined within the learner class, accordingly to the type of value that they are supposed to accept. In our specific case, IncMine allows to specify a value for

- `minSupportOption`, i.e., the Minimum Support Threshold σ. It allows values in $(0, 1]$, with default value 0.1.

opt	Option
s	`minSupportOption`
r	`relaxationRateOption`
l	`fixedSegmentLengthOption`
w	`windowSizeOption`
m	`maxItemsetLengthOption`

(a) Class `IncMine`.

opt	Option
f	`zakiFileOption`

(b) Class `ZakiFileStream`.

opt	Option
l	`learnerOption`
s	`streamOption`
m	`maxInstancesOption`
p	`numPassesOption`
b	`maxMemoryOption`

(c) Class `LearnModel`.

opt	Option
l	`learnerOption`
s	`streamOption`
e	`evaluatorOption`
i	`instanceLimit`
t	`timeLimitOption`
f	`sampleFrequencyOption`
b	`maxMemoryOption`
q	`memCheckFrequencyOption`
d	`dumpFileOption`

(d) Class `LearnEvaluateModel`.

Table 4.5: Options of the classes defined in IncMine library.

- `relaxationRateOption`, i.e., the Relaxation Rate r. It allows values in $[0, 1]$, with default value 0.5.
- `fixedSegmentLengthOption`, i.e., the fixed-length of each segment. Its default value is equal to 1000.
- `windowSizeOption`, i.e, the number of segments that will be kept into the sliding window. It also corresponds to the size of the support vector associated to each SemiFCI and to the maximum k value allowed by the progressively increasing MST function $minsup()$. (Look at Table 4.2 for an example). It has a default value of 10.
- `maxItemsetLengthOption`, i.e., the maximum length of itemsets to be considered. It can be useful set it to speed up the mining process when one knows the maximum size of the itemset of interest, because all itemsets in the current segment with length greater that this limit are discarded and do not influence the update process. It can be disabled by setting its value to -1. In this case the algorithm will perform a full mining of frequent closed itemset.

Listing 4.1: Compute FCIs over a data stream using IncMine.

```
1  import moa.learners.IncMine;
2  import moa.streams.ZakiFileStream;
3
4  public class Main {
5
6      public static void main(String args[]){
7          //read the stream T40I10D100K.dat
8          ZakiFileStream stream = new ZakiFileStream(``T40I10D100K.ascii'');
9          IncMine learner = new IncMine(); //create the learner
10
11         //configure the learner
12         learner.minSupportOption.setValue(0.01d);
13         learner.relaxationRateOption.setValue(0.5d);
14         learner.fixedSegmentLengthOption.setValue(1000);
15         learner.windowSizeOption.setValue(20);
16         learner.maxItemsetLengthOption.setValue(-1);
17         learner.resetLearning();
18
19         //prepare the stream for reading
20         stream.prepareForUse();
21
22         while(stream.hasMoreInstances() ){
23             //pass the next instance to the learner
24             learner.trainOnInstance(stream.nextInstance());
25         }
26         //output the final set of SemiFCIs
27         System.out.println(learner);
28     }
```

Chapter 5

Experimental results

During the last stages of the implementation phase, it becomes necessary to perform a testing session to identify problems in the implementation as well as strong and weak points. We were able to improve several aspects of the algorithm using the results of the early testing stages. Once we got a *stable* version of the software, we were able to perform an intensive testing session to evaluate how INCMINE algorithm works with both synthetic and real data streams.

In this Chapter first we explain how we generate these data streams. We show how INCMINE performs under different types of input, e.g., with drifting data streams, compared with MOMENT algorithm, which is still the standard for (exact) frequent itemset mining in data streams. Since INCMINE is an approximate algorithm, we detail its accuracy of the algorithm in terms of two well-known accuracy measures, the *precision* and *recall*. We provide an evaluation of the achievable throughput and of the memory consumption of the algorithm under the MOA environment.

At the time of testing we were not able to find an efficient Java implementation of MOMENT algorithm, so we decided to use use the original C++ provided by the author. C++ is commonly accepted to be more efficient than Java, so this difference must be taken into account when evaluating the results.

5.1 Generating synthetic data streams

To provide a fair evaluation of a stream data mining algorithm it is necessary to test it with different kind of inputs. Actually there is no application available for data streams generation for frequent itemset mining. Usually researchers decide to create *static* transactions databases and provide them

to the algorithm in a *stream fashion*. If the dataset is sufficiently large, although it is not infinite as real data stream should be, it can be considered a good test bench for evaluate the capabilities of the proposed algorithm.

One the most used synthetic data generator for itemset patterns is Zaki's *IBM Datagen* software [33]. It generates transactions databases for literal, sequential and taxonomic itemset mining. In our case we will use the first typology to generate different data streams. The easiness of generating such synthetic dataset convinced us to introduce into our solution a specific stream reader for the type of files that are outputted by this utility (i.e., the `ZakiFileStream` that we presented in Chapter 4.

The IBM Datagen program allows to specify several characteristics of the output dataset, such as

1. The *number of transactions* (default value: 10^6).
2. The *average number of items* per transaction (default value: 10).
3. The *number of different items*, which equals to the dimension $|\mathcal{I}|$ of the set of items $\mathcal{I}$ (default value: 10^8).
4. The *number of patterns* (default value: 10^4).
5. The *average length* of the maximal pattern (default value: 4).
6. The *correlation* between patterns (default value: 0.25).
7. The *average confidence* in a rule (default value: 0.75).

In the following sections we refer to different synthetic datasets. Their name follows the syntax:
$T\{avg\_t\_length\}I\{num\_diff\_items\}D\{num\_t\}[P\{avg\_patt\_length\}][C\{correlation\}]$

The number of different items is expressed in thousands. Any option that has not an assigned value assumes its default value. For example the datasets:

T40I10D100K refers to a dataset of 10^5 transactions, with an average of 40 items per transaction over a dictionary of 10^4 different items.

T50I10kD1MP6C05 refers to a dataset of 10^6 transactions, with an average of 50 items per transaction over a dictionary of 10^4 different items, maximal patterns of average length 6 and correlation between patterns of 0.5.

As we will see, options such as the average number of items per transaction and their average length, the average length of the maximal pattern and the correlation between patterns, have great influence over the number of itemsets that can be extracted and, in general, over the density of the resulting data stream.

5.1.1 Experiments

The initial testing phase was intended to measure the performances of our solution when the input is a standard data stream, with no drift or other particularities. We used the `T40I10D100K` dataset, a sparse dataset also provided by Zaki in [33], which is used as test set in several of the papers that we have analyzed so far. We study the accuracy of the algorithm in terms of precision and recall. We analyze its throughput and memory usage. We provide a comparison between INCMINE and MOMENT algorithm.

Notice that the MOMENT implementation we used was coded in C++. C++ code is compiled while Java code is interpreted by an intermediate software, the Java Virtual Machine (JVM). This introduces a significant overhead in executing Java applications with respect to C++ ones, since the latter can be run without external applications. According to [43], interpreted Java runs in the range of 20 times slower than C++. The factor may vary a lot depending on the type of application, the particular Java Virtual Machine used, the compiler used, but usually C++ code outperforms Java one in the majority of applications. Consider the consequences of this fact in the analysis of the following experiments.

Unless otherwise stated, in the following experiments the *segment length* of INCMINE is fixed to 500 transactions, while its *window size* is fixed to 10 segments. This equals to consider a *window length* of 5000 transactions for MOMENT algorithm.

With this experiments we want to evaluate the following characteristics of our solution:

- The *accuracy*, in terms of *precision* and *recall*.
- The *throughput*, to measure the processing time of the algorithm.
- The *memory consumption*, to measure the space occupied in memory by its data structures.

We will also study the dependance of the above measures on the minimum support threshold σ and the relaxation rate r. Essentially, INCMINE mines, for every segment of transactions S, the set of frequent closed itemsets with

support grater than $r\sigma|S|$, and then uses those itemsets to update the state of the entire sliding window. We consider as frequent only closed itemsets with support greater than $\sigma|S|$. The factor r allows to extract also infrequent closed itemsets that potentially can become frequent in the future. Thus, the lowest is the value of r, the bigger is the set of itemsets that is extracted from each segment at every window slide. This influences both the processing time and the accuracy of the results.

For example, consider segments of length $|S_i| = 500$ transactions. If we run INCMINE with a minimum support threshold $\sigma = 0.1$ and a relaxation rate $r = 0.5$, the algorithm will extract from each segment all the frequent closed itemsets with support greater $minsup_{Semi-FCI} = 0.5 \cdot 0.1 \cdot 500 = 25$. INCMINE will use those itemsets to keep updated the set of Semi-FCIs within the sliding window, while the set of approximate frequent closed itemset can be extracted considering only itemsets with support greater than $minsup_{FCI} = 0.1 \cdot 500 = 50$. A detailed explanation of the role of these variable is available in Section 4.2.

Note: We used MOA Release 2012.03 [4]. The solution was implemented with NetBeans 7.1.1 IDE (Build 201203012225) [42], using the Sun Java 1.7.0_03 JVM. We have developed and tested the software on a system with an Intel Core i5 M450 2.40 GHz Dual Core CPU and 4GB RAM. The operative system used was Windows 7. We set the Maximum heap size (`-Xms`) of the JVM to 1 GB for every INCMINE execution.

r	Prec.	Rec.
0.1	1.000	1.000
0.2	1.000	1.000
0.3	1.000	1.000
0.4	1.000	1.000
0.5	1.000	1.000
0.6	1.000	1.000
0.7	1.000	0.993
0.8	1.000	0.949
0.9	1.000	0.821
1.0	1.000	0.696

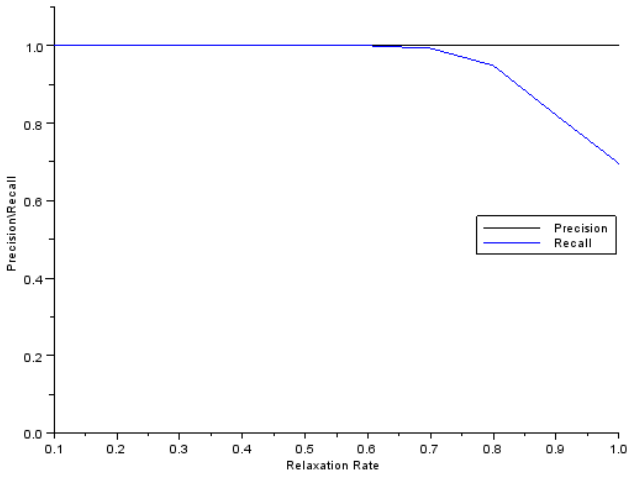


Figure 5.1: Precision and Recall of INCMINE algorithm varying r ($\sigma = 0.1$).

Accuracy

INCMINE provides an approximate solution to the frequent closed itemset mining problem. As we describe in Section 4.2, INCMINE method uses a *false-negative* approach. The *relaxation rate* r has a direct influence over the approximation, and it is important to estimate how this approximation affects the quality of the response. We fixed the MST $\sigma = 0.1$ and we vary r in $[0.1, 1]$. Thus we evaluate the effects of the variation of the relaxation rate on the *precision* and recall of the algorithm.

Precision and recall are respectively defined as $(|A \cap B|/|B|)$ and $(|A \cap B|/|A|)$, where A and B are the actual set and the approximate set of frequent itemsets over each window, respectively. We recover the set of FIs from the set of FCIs that are obtained by INCMINE at every entire window slide (computed with the procedure described in Algorithm 2.2), and we compare them to the real set of FIs computed with an implementation of the Eclat algorithm available in [35]. Finally we average the results we obtain for every window slide and report them.

We report the precision and recall of INCMINE in Figure 5.1. Precision and recall of MOMENT are omitted since MOMENT is an exact algorithm. We can see that INCMINE achieve really good accuracies for almost all values of r, demonstrating the good impact that the progressively increasing *minsup* function has. We always get a precision of 1, as any false-negative algorithm should have, and recall drops down only for values grater than 0.9.

In order to discover any possible skewed behavior, we decided to measure the accuracy also for different values of the minimum support threshold σ, fixed $r = 0.5$. We report in Figure 5.2 the values of precision and recall that we obtain. INCMINE always attains high quality results, even for a relaxation rate of 0.5. We can notice that in some cases (precisely for σ equal to 0.2 and 0.7) precision is not exactly 1 as expected. A deeper analysis shows that these little differences are due to few itemsets that stay in the 'border' between frequent and not frequent itemsets (i.e., itemsets whose support is exactly equal to $\sigma|S|$), but this does not influence the overall quality of the processing.

Throughput

It is also important to measure the effects of such variation in the parameters over the processing speed of the algorithm. We measure the average *throughput*, expressed in transactions per second (trans/sec), of processing for the entire data stream for different ranges of relaxation rate and min-

σ	Prec.	Rec.
0.02	0.995	0.972
0.03	1.000	0.991
0.04	1.000	0.999
0.05	1.000	0.999
0.06	1.000	0.999
0.07	0.994	1.000
0.08	1.000	1.000
0.09	1.000	1.000
0.10	1.000	1.000

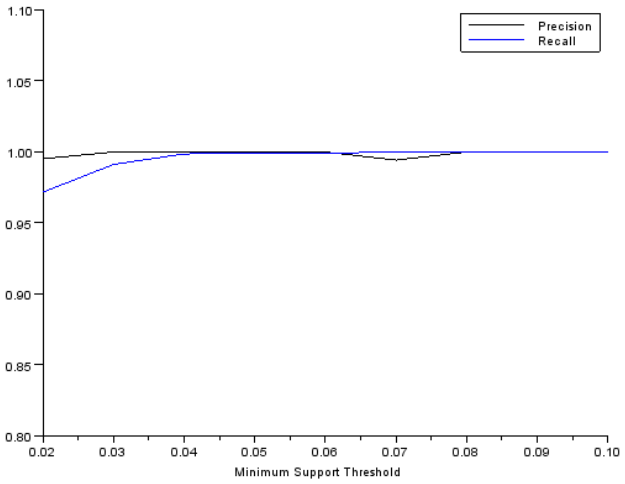


Figure 5.2: Precision and Recall of INCMINE algorithm varying σ ($r = 0.5$).

imum support threshold. We report in Figure 5.3 the average throughput values for $r \in [0.1, 1]$. The processing speed grows as the relaxation factor increases, this because higher values of r implies a lower number of frequent closed itemsets mined in every segment. This number has a great influence in the overall processing time.

We test MOMENT on the same data stream, with minimum support threshold $minsup = \sigma|S| = 0.1 \cdot 500 = 500$. Since MOMENT is independent from such relaxation rate, its throughput is constant for all these tests.

INCMINE clearly outperforms MOMENT for every value of $r \geq 0.2$, and only for $r = 0.1$ the performances of the two algorithms are comparable. For example, for $r = 0.5$ the throughput of INCMINE is more than two orders of magnitude bigger that MOMENT's one. At the same time, INCMINE achieves very good accuracies with this value of r, so we decide to adopt $r = 0.5$ for every future experiment.

Similarly to what we have done before, we also study the behavior of the throughput with respect to the minimum support threshold σ. We fixed $r = 0.5$ and we average the throughput obtained for $\sigma \in [0.02, 0.10]$. Figure 5.4 clearly shows that INCMINE outperforms MOMENT in every case, and the difference between them grows as the minimum support threshold increases. In every case, a part from $\sigma = 0.02$, INCMINE's throughput is at least one order of magnitude higher than MOMENT's one.

The authors performs similar tests in [6], comparing their C++ imple-

r	MOMENT	INCMINE
0.1	265.1	100.2
0.2	265.1	3372.5
0.3	265.1	12364.8
0.4	265.1	19686.3
0.5	265.1	27232.2
0.6	265.1	34232.3
0.7	265.1	40985.9
0.8	265.1	47408.9
0.9	265.1	52426.4
1.0	265.1	53723.7

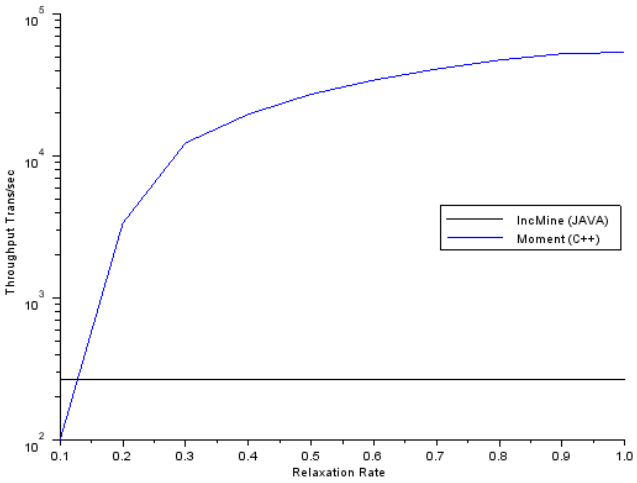


Figure 5.3: Throughput in trans/sec for different values of r ($\sigma = 0.1$). The minimum support used for MOMENT is equal to 500. Observe the logarithmic scale in the y axis.

mentation of INCMINE using the CHARM code provided by the author [31] with the same MOMENT implementation we used here. Although the differences between the two implementations, the results we obtained are comparable to the ones provided by the authors. At the moment we cannot provide a quantitative comparison between them, due to the different architecture used. However, since we obtain qualitatively a similar behavior, we can state that our proposal is a good and correct implementation of the original one.

Memory usage

Every data stream algorithm may run over systems with limited amount of available memory, such as nodes in a sensor network, which cannot handle memory loads as big as may a mainstream system can do. For this reason, in the different proposals we analyzed in Chapter 3, researchers tries to introduce data structures that can reduce as more as possible the memory consumption, meanwhile they want to achieve better processing performances (e.g., in terms of throughput).

So it becomes essential, also for our proposal, to have an idea of what would be its average memory consumption and how the parameter of the algorithm influences this measure. We decide to not compare INCMINE

σ	MOMENT	INCMINE
0.02	29.9	50.2
0.03	52.1	821.1
0.04	57.6	2461.6
0.05	79.5	6286.5
0.06	95.4	9582.4
0.07	130.7	13859.6
0.08	151.3	16163.0
0.09	245.0	19388.6
0.10	265.1	21181.0

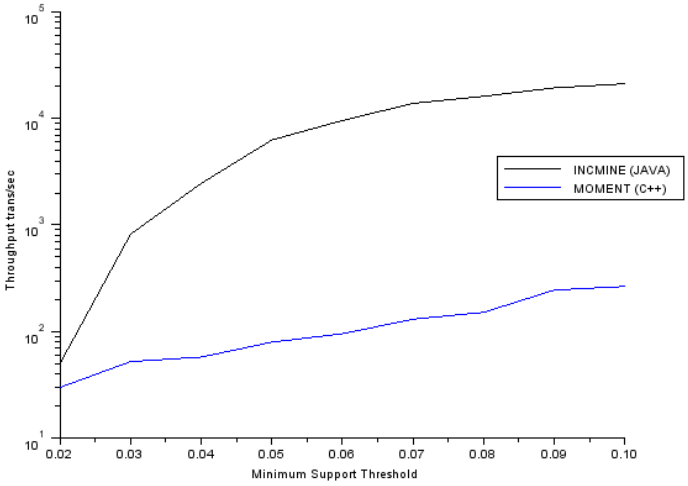


Figure 5.4: Throughput in trans/sec for different values of σ ($r = 0.5$). The minimum support used for MOMENT is equal to $\sigma \cdot 5000$. Observe the logarithmic scale in the y axis.

with MOMENT in this case, because of the great differences between the two architectures they are based on. In particular the Java Virtual Machine (and garbage collection) directly influences the memory measurement we obtain for INCMINE; such systems are not available for a C++ written program.

Note: We run the algorithm 10 times for each combination of the parameters, then we report the average results.

First we analyze the memory consumption of INCMINE in function of the minimum support threshold σ, fixed $r = 0.5$. In Table 5.1 we compare the overall memory consumption with the effective size in memory of the main data structures of INCMINE. The average memory consumption correctly decreases as the minimum support threshold increases, because a lower number of frequent closed itemsets have to mined and stored.

In the same table we also report the average size in memory of the main data structures used by INCMINE (the FCI-arrays and the Inverted FCI Index). Their size is several times lower than the whole memory consumption of the algorithm. This because the mining of frequent closed itemsets in the current segment is a very memory intensive phase, while the data structures used provides a *really compact* notation of the set of Semi-FCIs stored within the sliding window. For example, for a value of $r = 0.05$, we have an average memory usage of 221 MB, and only 1.4 MB of these are reserved

σ	Total Memory Usage	Data Structures Size
0.02	225.2	23.1
0.03	266.5	6.3
0.04	226.6	3.1
0.05	221.1	1.4
0.06	217.8	0.9
0.07	202.6	0.6
0.08	198.3	0.5
0.09	192.3	0.4
0.10	187.2	0.3

Table 5.1: Average memory consumption for σ ($r = 0.5$) in MB. We report the overall (Total) memory usage and the real size in memory of INCMINE's data structures (FCI-arrays + Inverted FCI Index).

to store the data structures of the algorithm. This suggests that a possible point of optimization for the algorithm could be found in the memory usage of the frequent closed itemset mining of each segment.

We also analyze the effects of changing *window size*. We fixed the minimum support threshold $\sigma = 0.05$ and study how the overall memory consumption and the size of the data structures varies. In Figure 5.5 we can see the behavior of the total memory consumption. For vales lower than 60 the memory consumption is almost constant. For larger windows the average memory consumption increases. These unexpected results can be justified by the intervention of the Garbage Collector, a mechanism implemented into the JVM to delete unreferenced objects to free memory, whose intervention skews the measurements of the memory usage.

Instead, if we look at the size in memory of the FCI-arrays and Inverted FCI Index shown in Figure 5.6, we can see that exists a linear dependance between the number of segments retained in the window and the size in memory of such data structures. A bigger sliding window corresponds to a higher number of frequent closed itemsets that have to be stored, thus a bigger space in memory is occupied. But it remains several times lower than the global memory consumption, proving the memory efficiency of the data structures that have been used.

5.2 Introducing concept drift

It is really important to verify how a data stream algorithm reacts to concept drift. A data stream is a continuously evolving source of data, and a good

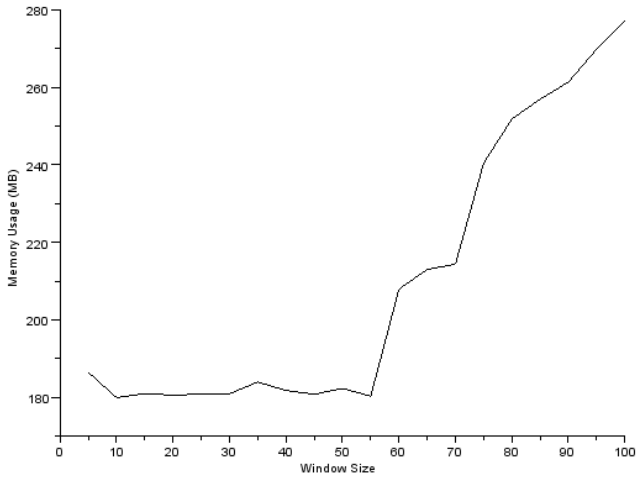


Figure 5.5: Average overall memory consumption for different window size values ($\sigma = 0.05$, $r = 0.5$).

mining algorithm has to react as quick as possible to any change, whether it is a sudden or a soft change.

There are infinite possible changes, but we are interested only in drifts that affect the result of our algorithm, that is, the set of frequent itemsets. This can be detected by monitoring the number of added and deleted frequent itemsets in the data structures, like we use in real data processing, presented in Section 5.3.

In many other cases detect a drift can be even easier, by simply monitoring the total number of frequent itemsets. If it varies significantly, we detect a change. If not, we may not detect a change, and we get a false negative. We think that these cases are rare in practice, since almost every drift produces a change in the number of frequent itemsets. In this way we also get a computationally simpler test.

While real data streams are 'naturally' affected by concept drift, it is necessary to find a way to introduce concept drift into synthetic data streams. By testing our algorithm over different synthetic concept drifts, we can get an idea on how it will work in real cases, and study what are the configurations that better adapts to each possible situation.

Bifet et al. introduce in [36] a new experimental framework for concept drift. They explain a straightforward way to introduce artificial drift to data streams generators. Starting from two different data streams corresponding to two different concepts, simply we need to define the probability that every

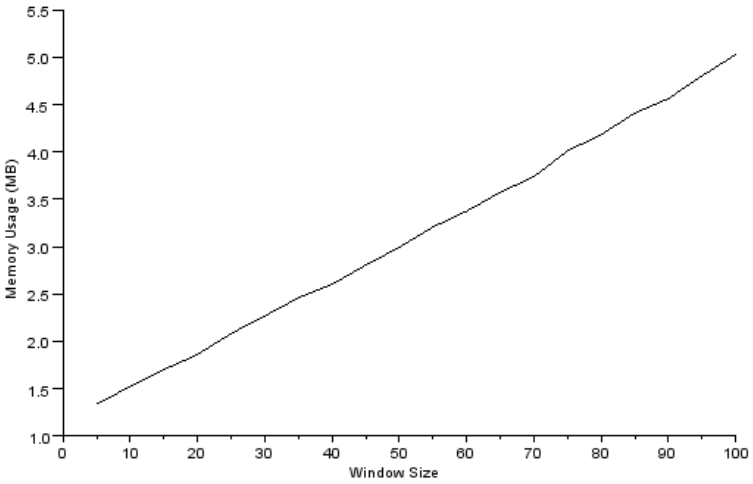


Figure 5.6: Average memory consumption of INCMINE's data structures for different window size values ($\sigma = 0.05$, $r = 0.5$).

new instance of the stream belongs to the new concept after the drift (i.e., the probability that an instance of the second stream appears in the final stream). They use the sigmoid function

$$f(t) = 1/(1 + e^{-s(t-t_0)}) \tag{5.1}$$

to express this probability. As shown in Figure 5.7, it has a derivative at the point t_0 of $f'(t_0) = \tan\alpha = s/4$. It can easily seen that, as $s = 4\tan\alpha$ and $\tan\alpha = 1/L$, then $s = 4/L$. In the sigmoid model the parameter t_0 specifies the point of change and L the length of change. This sigmoid-based approach can be applied to create a data stream with multiple changes by joining different concept changes.

We implement two scripts in `python`, named `merge-sigmoid.py` and `merge-step.py`, which outputs a new stream with, respectively, a sigmoid or a sudden drift (a.k.a. a *step* drift, since we pass from a stream to another in a binary way) between the following two data streams created with the IBM Datagen.

1. T40I10kD1MP6
2. T50I10kD1MP6C05

Since the correlation between transactions in the `T50I10kD1MP6C05` stream is higher than `T40I10kD1MP6` one, the former has a higher density and more

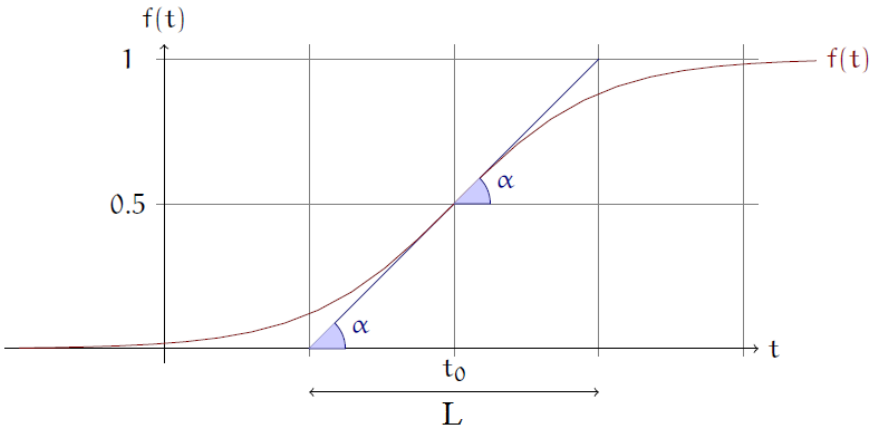


Figure 5.7: The sigmoid function.

frequent itemsets can be extracted. This difference between the two streams is sufficient to evaluate correctly the quality of the reaction to every kind of concept drift.

In every of the following experiments INCMINE uses a minimum support threshold $\sigma = 0.01$, a relaxation rate $r = 0.5$ and segments of 1000 transactions.

5.2.1 Reaction to sudden drift

It is important to have an idea of how much time needs a data stream algorithm to adapt to a concept change. The more time is needed, the less is the *adaptive power* of the algorithm. In data streams mining, it means that we have to wait longer to have a response that reflects the real state of data that are mined. And, obviously, we want to reduce this time as more as possible, in order to stay in line with the online processing paradigm.

We now analyze the reaction to *sudden drift*. A sudden drift consists in an abrupt change from a concept to another. If we consider the two concepts as two binary values, in a sudden drift we pass from one value to another with no intermediate changes, like a step function. Sudden drifts allows to compute easily the *reaction time* of a data stream algorithm.

The starting time t_{start} of the concept drift can be defined exactly (i.e., looking at the transaction where we pass from one concept to the other in the synthetic drift generation); we can consider that a frequent itemset data stream algorithm 'reaches' a concept when the its number of Frequent Itemsets (or Frequent Closed Itemsets) is *sufficiently close* to the number of Frequent Itemsets of this concept.

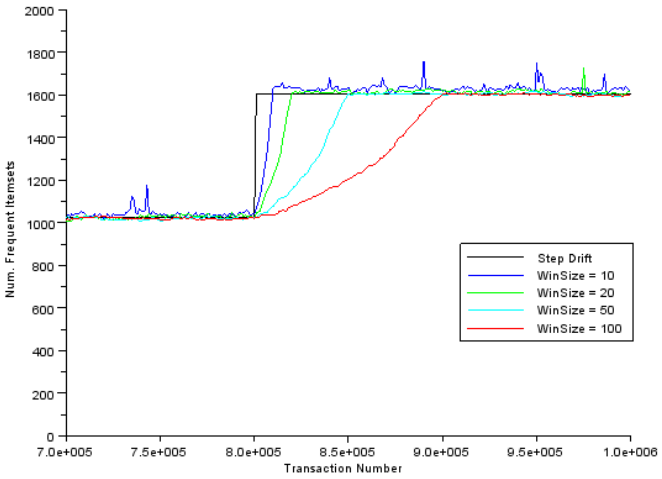


Figure 5.8: Step drift and number of extracted FIs for $window\_size \in \{10, 20, 50, 100\}$.

Since the number of FIs varies every transaction, we decide that a concept is reached when the absolute difference between the number of FIs mined and the real number of FIs is lower that the 5% of the latter value.

$$t_{reach} = t \text{ such that } \frac{||FI_{mined}(t)| - |FI_{real}(t)||}{|FI_{real}(t)|} \leq 0.05. \tag{5.2}$$

Notice that the real number of FIs is also a random variable, so we consider as $FI_{real}(t)$ as the average number of Frequent Itemset mined along the whole stream (concept). We compute it by mining each stream 10 times and getting the average number of FIs mined.

We define the *reaction time* of a frequent itemset stream mining algorithm as difference between the time of reach and the starting time of the drift, expressed in number of transactions.

$$reaction\_time = t_{reach} - t_{start} \tag{5.3}$$

We generate a sudden drift between the two test streams, passing from one to the other at transaction $8 \cdot 10^5$, which also corresponds to our t_{start}. We represent in Figure 5.8 the evolution of the number of Frequent Itemsets mined for different dimensions of the sliding window. This picture shows clearly that increasing the window size also increases the reaction time. Have a bigger window means reduces the influence of the last segments mined on the overall frequent closed itemsets that are stored, and this implies longer time to react to a sudden drift. In particular, the drift ends when almost

win_size	*react_time*
10	9
20	18
30	27
40	36
50	46
60	55
70	64
80	73
90	82
100	91

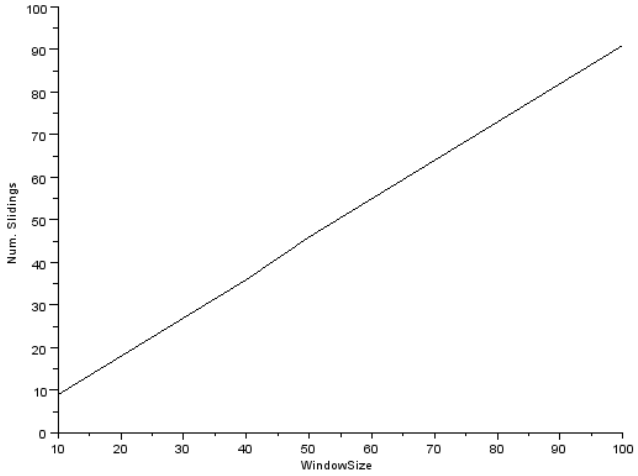


Figure 5.9: Reaction time for $window\_size \in [10, 100]$.

all the transactions in the window belongs to the new concept, as we can notice from Figure 5.9.

5.2.2 React to sigmoidal drift

Once analyzed the dependance between the window size and the capability of INCMINE to react to a sudden drift, we test it over gruadual drifts. Since, at the best of our knowledge, there is no easy way to provide a measure to express the reaction time of a data stream mining algorithm to soft sigmoidal drifts, we provide only graphical results here.

We generate and test three different sigmoidal drifts, with the following characteristics, ordered from the harder to the softer

1. point of change $t_0 = 8 \cdot 10^5$, drift length $L = 5 \cdot 10^3$ transactions.
2. point of change $t_0 = 8 \cdot 10^5$, drift length $L = 5 \cdot 10^4$ transactions.
3. point of change $t_0 = 8 \cdot 10^5$, drift length $L = 2 \cdot 10^5$ transactions.

Figures 5.10, 5.11, 5.12 show the evolution of the number of Frequent Itemsets extracted from the Frequent Closed Itemsets returned by INCMINE. In every case, the behavior is almost the same we have noticed for abrupt changes. That is, bigger windows corresponds to longer times of response. But now, depending on the nature of the stream, the drift can last from

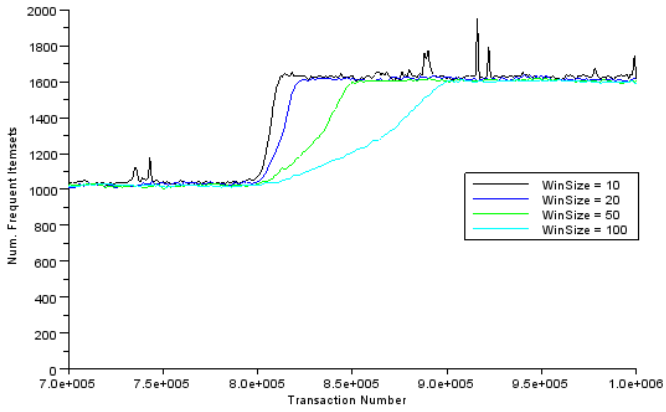


Figure 5.10: Sigmoid drift ($t_0 = 8 \cdot 10^5$, $L = 5 \cdot 10^3$).

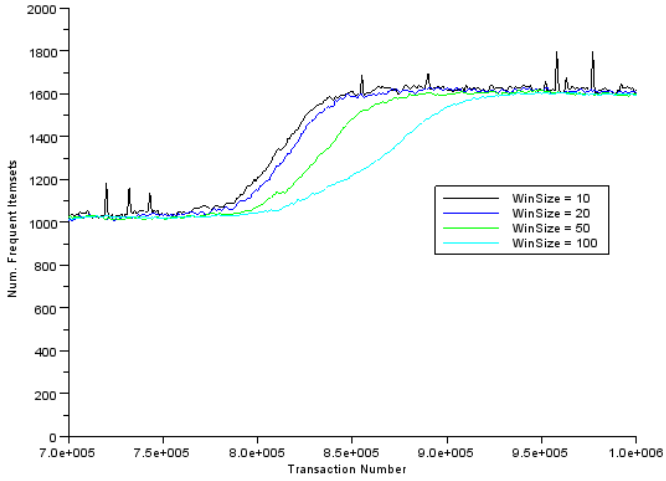


Figure 5.11: Sigmoid drift ($t_0 = 8 \cdot 10^5$, $L = 5 \cdot 10^4$).

some hundreds of transactions to several thousands, and this amplifies the differences between small and bigger windows.

Another interesting information can be extracted from the graphs above. Some skewed behaviors, such as peaks in the number of FIs, are well filtered by big windows. This fact is clearly visible in Figure 5.12, which shows the entire stream processing. Peaks that appears for windows of size 10 almost disappears for bigger windows, which shows a more stable processing.

Consequently, it is important to select accurately the window size to re-

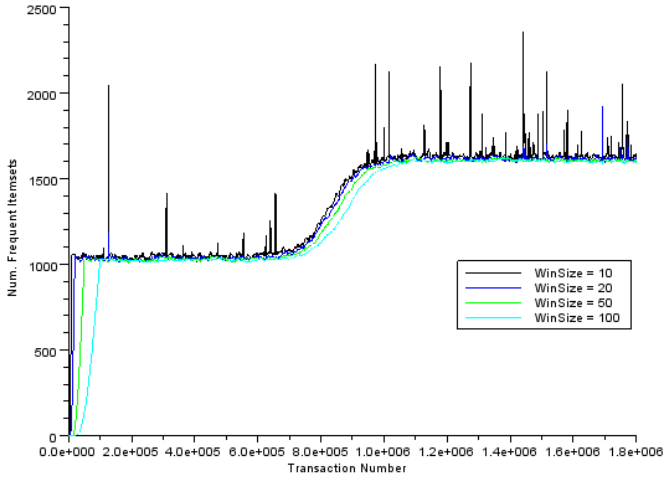


Figure 5.12: Sigmoid drift ($t_0 = 8 \cdot 10^5$, $L = 2 \cdot 10^5$).

act quickly in case of concept drift, but also to have a more stable processing where no change happens.

We will discuss more in the conclusions the problem of the balance between keeping a stable, noise-tolerant computation and having a fast reaction time. In particular, it is a challenge to do this automatically without user intervention.

5.3 Experiments with real data

In parallel with testing synthetic data streams, we advised the necessity to run our solution with some *real data stream*. Currently there are several data streams available online (e.g., Yahoo news, Twitter, financial data streams), but it is cumbersome to create a specific interface for them, and this is out from the scope of the project. Furthermore, it is not clear if such data streams can be used for itemset mining.

We elaborate a huge but batch dataset, and we use its timing information to create a stream where transactions are passed in order of ascending time.

We use the *MOVIELENS* dataset, a free dataset provided by Group Lens Research [37]. MOVIELENS dataset records user ratings for movies. A rating is a value between $(1, \ldots, 5)$ with half ratings, that a user provides after seeing it. MOVIELENS is originally intended to be used in recommendation systems, neither for online processing nor for itemset mining purposes. The

former point effectively was not a problem, since we have already seen how to treat static datasets as data streams. But the latter was real issue, since we have to convert data coming from a film recommendation system into a transactions database ready to be passed to our algorithm.

The MOVIELENS dataset contains about 10 millions ratings applied to 10681 movies by 71567 users of the online web service MovieLens. It collects ratings starting from *29 Jan 1996 00:00:00 GMT* to *Wed, 15 Aug 2007 07:20:00 GMT*. It is composed of several files, but we used only files `ratings.dat`, containing the information over the ratings of each user, and file `movies.dat`, the dictionary of all the movies. Data in each file are organized as follows

`ratings.dat` *UserID::MovieID::Rating::Timestamp*, where Timestamp represents the seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970.

`movies.dat` *MovieID::Title::Genres*.

According to this representation, we create a transactions database using *MovieID* as items and grouping ratings by *Timestamp*, then sort them for *Timestamp* in ascendent order.

The principal problem in this phase was to identify a good way to group ratings together. In fact, grouping by the simple timestamp results in a huge dataset with small transactions, since two movies appears in the same transaction only if they were rated in the same second.

We need to group movies with thicker granularity, and after several tries, we decide to group into the same transaction movies that were rated into the same 5 minutes.

We also impose a maximum of 50 items for each transaction, and we subdivide longer transactions into several different ones of the same length. This approximation becomes necessary to reduce the effect of some really skewed transactions that appear after grouping. We will see that this process does not affect the quality of the processing and the results.

We obtain a data stream of 622265 transactions with an average of 10.37 items per transaction. It represents the movies that have been rated into the MOVIELENS recommendation system in the same 5 minutes interval, independently from the user that have done the evaluation. Transactions are not uniformly distributed along this time interval.

We perform several processing to find the best configuration for INCMINE. Since we are treating with real data, it is not trivial to configure a stream data mining algorithm to perform perfectly in every possible condi-

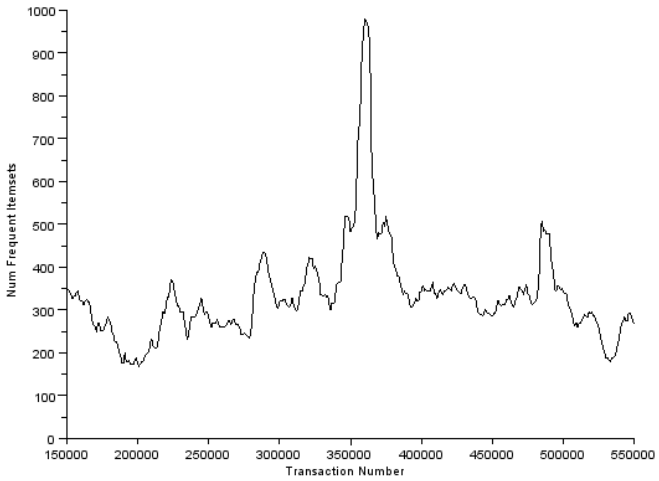


Figure 5.13: Number of FIs for a subset of transactions of the MOVIELENS stream. It corresponds to transactions collected between 28 Mar 2000 23:00:00 GMT and 02 Aug 2006 01:35:00 GMT.

tion, since performances can be affected from different factors that can vary in an unpredictable way, such as the density of data.

Considering that the number of different items is similar to what we use in previous synthetic tests (in the order of 10K elements), we can use the knowledge we acquired there to guide the first stages of the processing. We aim to extract what are the movies that are used to be rated together and to detect when a change is occurring.

To react quickly to concept drifts, the window size should be small, e.g, 10 segments per window. The dimension of each segment depends on the level of detail we want to obtain. For example, if we consider segments of 1000 transactions each, it means that we may achieve a minimum *timing resolution* of $1000trans \cdot 5min/trans = 5000min \approx 3.5days$, in the hypothesis of uniformly distributed transactions. Actually this hypothesis does not hold, so the real timing resolution can be a greater number of days.

The minimum support threshold should be decided accordingly to the previous two parameters, and to the depth of the mining process we want to perform. This because the lower is minimum support threshold the longer will be the itemsets mined. On the other hand, an excessively low threshold can return rare and, in fact, not interesting itemsets. And also it may slow down excessively the processing, considering the effect of the relaxation factor.

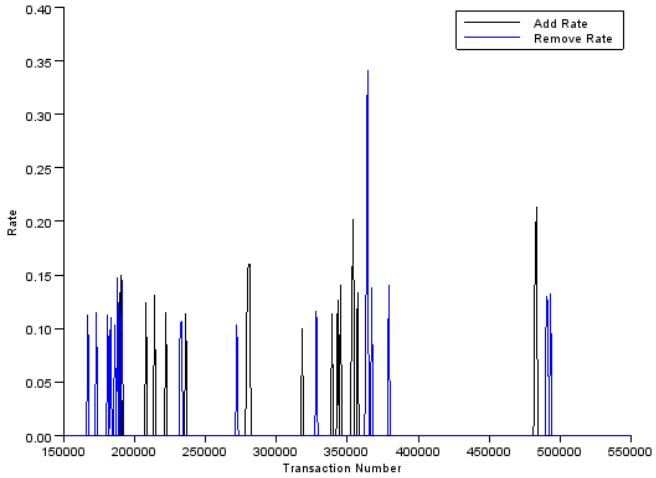


Figure 5.14: Significant FIs add and remove rates for a subset of transactions of the MOVIELENS stream. It corresponds to transactions collected between 28 Mar 2000 23:00:00 GMT and 02 Aug 2006 01:35:00 GMT.

Figure 5.13 shows the evolution on the number of Frequent Itemsets extracted from INCMINE for a significant subset of transactions of the MOVIELENS data stream. It has been previously configured to work with a minimum support threshold $\sigma = 0.02$, relaxation rate $r = 0.5$ and a window size of 10 segments, with 100 transactions per segment. We mine itemsets of maximum length 5, since we are interested in groups of 2 or 3 movies that appear frequently together.

Obviously this results are totally different on the ones we obtained from synthetic drifts, where the differences between concepts we introduced were clear. In this case no subdivision among concepts can be clearly identified. The distribution of the FIs has one important peak, which can be attributed to an important variation on the density of the data stream. There are also other minor peaks and variations, but no sufficient information can be extracted looking only to the total number of FIs (or, similarly, to FCIs).

Thus, it becomes necessary to track also the number of itemsets that have been added or removed in every segment. This information can help us to better identify changes. A sudden increase (decrease) of frequent itemsets can testify that a change is occurring, because several of the itemsets that were kept into the sliding window stop to be frequent (and viceversa).

For every window slide we compute the rates of frequent itemsets that have been added or removed with respect to their total number. So, for a

segment S, we define the following measures:

$$\begin{aligned} add\_rate &= \frac{|FI_{added}(S)|}{|FI(S)|} \\ rem\_rate &= \frac{|FI_{removed}(S)|}{|FI(S)|} \end{aligned}$$

We consider that a drift is occurring when one of the above coefficients becomes grater than 0.1, which means that at least the 10% of the frequent itemsets have been renewed in the last window slide.

Figure 5.14 shows the significant values of the previous rates for a subset of transactions. Each peak corresponds to a possible drift in the concept, thus the set of frequent itemsets is outputted to be analyzed.

Table 5.2 reports the top 3 frequent itemsets of dimension grater than 1 for some drifted transaction, along with the corresponding date. It is really interesting to look how the set of frequent itemsets varies along years, and how the most rated movies corresponds to some of the most famous ones at the time. The evolution of popular movies over time would have been unnoticed if we had used a batch algorithm that ignored the temporal sequence of the ratings.

(*transaction*, *date*)	Frequent Itemsets
(168000, 16 Jul 2000)	• Lord of the Rings: The Fellowship of the Ring, The (2001); Beautiful Mind, A (2001). • Harry Potter and the Sorcerer's Stone (2001); Lord of the Rings: The Fellowship of the Ring, The (2001). • Ocean's Eleven (2001); Lord of the Rings: The Fellowship of the Ring, The (2001).
(189000, 23 Nov 2000)	• Spider-Man (2002); Star Wars: Episode II - Attack of the Clones (2002). • Bourne Identity, The (2002); Minority Report (2002).
(223000, 29 Jun 2001)	• Lord of the Rings: The Fellowship of the Ring, The (2001); Lord of the Rings: The Two Towers, The (2002). • Minority Report (2002); Signs (2002). • Lord of the Rings: The Two Towers, The (2002); Catch Me If You Can (2002); .
(282000, 05 Jul 2002)	• Lord of the Rings: The Fellowship of the Ring, The (2001); Lord of the Rings: The Two Towers, The (2002). • Lord of the Rings: The Two Towers, The (2002); Lord of the Rings: The Return of the King, The (2003). • Lord of the Rings: The Two Towers, The (2002); Pirates of the Caribbean: The Curse of the Black Pearl (2003).

Table 5.2: Example of the evolution of Frequent Itemsets extracted from the MOVIELENS data stream when a drift is detected.

Chapter 6

Conclusions and future works

In this work we have presented a frequent closed itemset mining solution based on the INCMINE algorithm. This algorithm is perfectly integrated within the Massive Online Analysis framework, and ready to be used from every user.

We have obtained very good accuracies in terms of precision and recall in the experiments. We have also obtained good performances in processing time. Experimental results shows clearly that our solution is faster than MOMENT, taking in account the our solution is approximate, while MOMENT is an exact algorithm. For not so small values of the relaxation factor we are able to get a good accuracy and, meanwhile, the algorithm runs several orders of magnitude faster than MOMENT.

The memory consumption of the algorithm is reasonable, although we were not able to compare it with MOMENT due to the excessive differences between the implementations of these algorithms that we used.

The characteristics of the algorithm guarantee adaptability to concept drift. We have verified it simulating different types of change in synthetic data streams. We have also tested the algorithm over a data stream generated from real data, obtaining interesting results. The algorithm well adapts also to real concept changes, providing a response that reflects the state of stream in its temporal evolution.

The final numbers of hours per stage, schedule and costs is as in Table 6.1. We can see that the main deviations with respect to the initial plan are due to more time required for acquiring background knowledge and paper analysis.

It remains an open problem how to make the algorithm to be auto-adaptive. This is a well known problem, where the algorithm should choose its parameters by itself, such as the minimum support threshold, the mini-

Phase	Deadline	Hours	Cost	Total
Required knowledge acquisition	15/02/2012	100	15 €/h	1500 €
Paper analysis	15/03/2012	200	15 €/h	3000 €
Design/Implementation	30/04/2012	250	20 €/h	5000 €
Testing I	15/05/2012	75	15 €/h	1125 €
Testing II	31/05/2012	75	15 €/h	1125 €
Reporting	20/06/2012	100	15 €/h	1500 €
TOTAL	-	**800**	-	**13250 €**

Table 6.1: Final time schedule and economic cost of the project.

mum support function and the window size. The ADWIN method proposed in [34] has been used successfully in other contexts to achieve this adaptation. Its main idea is to keep a window whose length varies according to the change detected in the stream, so that when change is occurring it shrinks to achieve faster reaction time, and when there is no change it enlarges to achieve more stability.

Other improvements can be obtained by trying different algorithms for frequent closed itemset mining, which may reduce the memory consumption of the algorithm.

Bibliography

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. VLDB Conference, 1994.

[2] J. Han, H. Cheng, D. Xin, X. Yan. Frequent pattern mining: current status and future directions. Data Min. Knowl. Discov. 15(1): 55-86 (2007).

[3] J. Han, J. Pei, Y. Yin. Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. Data Min. Knowl. Discov. 8(1): 53-87 (2004).

[4] MOA Massive Online Analysis. http://moa.cs.waikato.ac.nz/

[5] A. Bifet, G. Holmes, R. Kirkby, B. Pfahringer. MOA: Massive Online Analysis. Journal of Machine Learning Research 11: 1601-1604 (2010).

[6] J.Cheng, Y. Ke, W. Ng. Maintaining frequent closed itemsets over a sliding window. J. Intell. Inf. Syst. 31(3): 191-215 (2008).

[7] S. Yen, C. Wu, Y. Lee, V. Tseng, C. Hsieh. A Fast Algorithm for Mining Frequent Closed Itemsets over Stream Sliding Window. FUZZ-IEEE Conference, 2011.

[8] M. Memar, M. Deypir, M. Sadreddini, S. Fakhrahmad. An Efficient Frequent Itemset Mining Method over High-speed Data Streams

[9] L. Guon, H. Su, Y. Qu. Approximate mining of global closed frequent itemsets over data streams. Journal of the Franklin institute, 2011.

[10] G. Song, D. Yang, B. Cui, B Zheng, Y. Liu, K. Xie. CLAIM: An Efficient Method for Relaxed Frequent Closed Itemset Mining over Stream Data. DASFAA Conference, 2007.

[11] H. Li, C. Hob, S. Lee. Incremental updates of closed frequent itemsets over continuous data streams. Expert Syst. Appl. (2009).

[12] H. Li, S. Lee. Mining frequent itemsets over data streams using efficient window sliding techniques. Expert Syst. Appl. (2009).

[13] Y. Chi, H. Wang, P.S. Yu, R.R. Muntz. Catch the moment: maintaining closed frequent itemsets over a data stream sliding window. Knowl. Inf. Syst. (2006).

[14] A. Bifet, R. Gavaldà. Mining Frequent Closed Trees in Evolving Data Streams. Intell. Data Anal. (2011).

[15] T. Calders, N. Dexters, B. Goethals. Mining Frequent Itemsets in a Stream. ICDM Conference, 2007.

[16] J. Cheng, Y. Ke, W. Ng. A Survey on Algorithms for Mining Frequent Itemsets over Data Streams. Knowl. Inf. Syst. (2008).

[17] D. Knuth. The Art of Computer Programming, volume 3: Searching and Sorting. Addison-Wesley, 1998.

[18] N. Jiang, L. Gruenwald. CFI-Stream: Mining Closed Frequent Itemsets in Data Streams. KDD Conference, 2006.

[19] M. Memar, M. Deypir, M. H. Sadreddini, S. M. Fakhrahmad. An Efficient Frequent Itemset Mining Method over High-speed Data Streams

[20] M. Deypir, M. H. Sadreddini. A Dynamic Layout of Sliding Window for Frequent Itemset Mininig over Data Streams. Journal of Systems and Software 85(3): 746-759 (2012)

[21] G. S. Manku, R. Motwani. Approximate Frequency Counts over Data Streams. VLDB Conference, 2002.

[22] J. H. Chang, W. S. Lee. estWin : Adaptively Monitoring the Recent Change of Frequent Itemsets over Online Data Streams. J. Information Science (2005).

[23] D. Lee, W. Lee. Finding Maximal Frequent Itemsets over Online Data Streams. ICDM Confernce, 2005.

[24] H. Li, S. Lee, M. Shan. An Efficient Algorithm for Mining Frequent Itemsets over the Entire History of Data Streams

[25] J. Pei, G. Dong, W. Zou, J. Han. On Computing Condensed Frequent Pattern Bases. ICDM Confernce, 2002.

[26] J. S. Culpepper, A. Moffat. Efficient Set Intersection for Inverted Indexing. ACM Trans. Inf. Syst. 29(1): 1 (2010).

[27] C. Li, K. Jea. An Adaptive Approximation Method to Discover Frequent Itemsets over Sliding-window-based Data Streams. Expert Syst. Appl. (2011).

[28] A. Rajaraman, J. D. Ullman. Mining of Massive Datasets. Cambridge University Press, 2011.

[29] M. J. Zaki. Lecture notes from the Data Mining course at Rensselaer Polytechnic Institute. `www.cs.rpi.edu/~zaki/`

[30] M. J. Zaki. Scalable algorithms for association mining. IEEE Trans. Knowl. Data Eng. 12(3): 372-390 (2000).

[31] M. J. Zaki, C. J. Hsiao. CHARM: An efficient algorithm for closed association rule mining. SDM Conference, 2002.

[32] M. J. Zaki, K. Goudaz Fast Vertical Mining Using Diffsets. KDD Conference, 2003.

[33] M. J. Zaki. `http://www.cs.rpi.edu/~zaki/www-new/pmwiki.php/Software/Software`

[34] A. Bifet, R. Gavaldà. Learning from time-changing data with adaptive windowing. SDM Conference, 2007.

[35] Philippe Fournier-Viger. A Sequential Pattern Mining Framework `http://www.philippe-fournier-viger.com/spmf/index.php`

[36] A. Bifet. Adaptive Learning and Mining for Data Streams and Frequent Patterns. Ph.D. thesis, Dept. LSI, Universitat Politècnica de Catalunya, 2009.

[37] GroupLens Research, University of Minnesota `http://www.grouplens.org/`

[38] P. Tan, M. Steinbach, V. Kumar. Introduction to Data Mining. Addison-Wesley, 2005.

[39] C. Lucchese, S. Orlando, and R. Perego. Fast and memory efficient mining of frequent closed itemsets. IEEE Trans. Knowl. Data Eng. 18(1): 21-36 (2006).

[40] S. Ben Yahia, T. Hamrouni, E. Mephu Nguifo. Frequent closed itemset based algorithms: A thorough structural and analytical survey. SIGKDD Explorations 8(1): 93-104 (2006).

[41] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I. H. Witten. The WEKA Data Mining Software: An Update. SIGKDD Explorations 11(1): 10-18 (2009).

[42] `http://netbeans.org/`

[43] B. Eckel. Thinking in Java. Prentice-Hall, 2006.

[44] P. Palmerini, S. Orlando, R. Perego. Statistical Properties of Transactional Databases. SAC Conference, 2004.

[45] A. Tsymbal. The problem of concept drift: definitions and related work. Technical Report TCD-CS-2004-15, 2004.

[46] K.O. Stanley. Learning concept drift with a committee of decision trees. Tech. Report UTAI-TR-03-302, 2003.

Appendix A

A.1 ECLAT Algorithm

Algorithm A.1 is a pseudo-code for the ECLAT algorithm, presented by Zaki et al. in [30].

Algorithm A.1 Eclat(P,$minsup$)

```
1: foreach ⟨X, t(X)⟩ ∈ P do
2:     P_X ← ∅
3:     foreach ⟨Y, t(X)⟩ ∈ P, with Y > X do
4:         N_XY ← X ∪ Y
5:         t(N_XY) ← t(X) ∩ t(Y)
6:         if sup(N_XY) ≥ minsup then
7:             P_X ← P_X ∪ {⟨N_XY, t(N_XY)⟩}
8:             print N_XY, sup(X_XY)
9:     Eclat(P_X,minsup)
```

A.2 FP-GROWTH Algorithm

Algorithm A.2 is a pseudo-code for the FP-GROWTH algorithm, presented by Han et al. in [3].

A.3 Properties of the Closure Operator

Define by $\boldsymbol{c} : 2^{\mathcal{I}} \to 2^{\mathcal{I}}$ the *closure operator* as

$$\boldsymbol{c}(X) = \boldsymbol{i} \circ \boldsymbol{t}(X) = \boldsymbol{i}(\boldsymbol{t}(X)) \tag{A.1}$$

The closure operator $\boldsymbol{c}$ maps itemsets to itemsets, an it is:

Algorithm A.2 FP-Growth(R,P,$minsup$)

```
1: if IsPath(R) then
2:     foreach X_i ⊆ R do
3:         X ← P ∪ X_i
4:         sup(X) ← min_i ∈ X{sup(i)}
5:         print X, sup(X)
6: else
7:     I ← {i | i ∈ R, sup(i) ≥ minsup}
8:     foreach i ∈ I in increasing order of sup(i) do
9:         X ← P ∪ {i}
10:        sup(X) ← sup(i)
11:        print X, sup(X)
12:        R_X ← ∅ /* Projected FP-tree for X */
13:        foreach path ∈ PathToRoot(i) do
14:            pathsup(i) ← support of i in path path
15:            Insert p into FP-tree R_X with support pathsup(i)
16:        if R_X ≠ ∅ then Eclat(R_X, X, minsup)
```

1. Extensive: $X \subseteq \boldsymbol{c}(X)$

2. Monotonic: If $X_i \subseteq X_j$ then $\boldsymbol{c}(X_i) \subseteq \boldsymbol{c}(X_j)$

3. Idempotent: $\boldsymbol{c}(\boldsymbol{c}(\mathrm{X})) = \boldsymbol{c}(\mathrm{X})$

An itemset X is called *closed* if $\boldsymbol{c}(X) = X$, i.e., if X is a fixed-point of the closure operator $\boldsymbol{c}$. If $X \neq \boldsymbol{c}(X)$, then X is not closed, but the set $\boldsymbol{c}(X)$ is called its *closure*. From the properties of the closure operator, both X and $\boldsymbol{c}(X)$ have the same tidset.

A.4 CHARM Properties

Given a collection of IT-pairs $\{\langle X_i, \boldsymbol{t}(X_i)\rangle\}$, the following properties hold:

1. If $\boldsymbol{t}(X_i) = \boldsymbol{t}(X_j)$, then $\boldsymbol{c}(X_i) = \boldsymbol{c}(X_j) = \boldsymbol{t}(X_i \cup X_j)$

2. If $\boldsymbol{t}(X_i) \subset \boldsymbol{t}(X_j)$, then $\boldsymbol{c}(X_i) \neq \boldsymbol{c}(X_j)$ but $\boldsymbol{c}(X_i) = \boldsymbol{c}(X_i \cup X_j)$

3. If $\boldsymbol{t}(X_i) \neq \boldsymbol{t}(X_j)$, then $\boldsymbol{c}(X_i) \neq \boldsymbol{c}(X_j) \neq \boldsymbol{t}(X_i \cup X_j)$

A.5 A general Incremental Closed Pattern Mining solution

We present now a general methodology to identify closed patterns in a data stream proposed by Bifet et al. in [14]. It uses intersection of patterns and the Galois Lattice Theory to perform the mining process. It can be applied to different kinds of patterns, such as trees, graphs and, obviously, itemsets. It can be seen as general representation of all incremental algorithms that we will analyze later in this Chapter.

It uses several propositions that can be extracted from Formal Concept Analysis to provide two efficient procedures to update the set of closed patterns of a first dataset $\mathcal{D}_\infty$ with the set of closed patterns $\mathcal{D}_\in$.In particular the propositions that are used by these two procedures are (no proof is provided here; note that we represent the closure of a pattern t into a dataset of transaction D as $\boldsymbol{c}_D(t)$)

Proposition 1 Adding a pattern transaction t to a dataset of patterns D does not decrease the number of closed patterns for D.

Proposition 2 A pattern transaction t added to a dataset of patterns D will be a closed pattern for D.

Proposition 3 Adding a transaction with pattern t ti a dataset of patterns D where t is closed does not modify the number of closed patterns for D.

Proposition 4 Deleting a pattern transaction from a dataset of patterns D does not increase the number of closed patterns for D.

Proposition 5 Deleting a pattern transaction that is repeated in a dataset of patterns D does not modify the number of closed patterns for D.

Proposition 6 Let D_1 and D_2 be two datasets of patterns. A pattern t is closed for $D_1 \cup D_2$ if and only if it is in the intersection of its closures $\boldsymbol{c}_{D_1}(t)$ and $\boldsymbol{c}_{D_2}(t)$.

From these properties the following corollary can be deduced

Corollary 1 Let D_1 and D_2 be two datasets of patterns. A pattern t is closed for $D_1 \cup D_2$ if and only if

- t is a closed pattern for D_1, or
- t is a closed pattern for D_2, or

Algorithm A.3 Closed_Subpattern_Mining_Add$(T_1, T_2, minsup, T)$

```
1: T ← T_1
2: foreach t in T_2 in size-ascending order do
3:     if t is closed in T_1 then
4:         sup_T(t) ← sup_T(t) + sup_{T_2}(t)
5:         foreach t' that is a subpattern of t do
6:             if t' ∈ T_1 then
7:                 if sup(t') is not updated then
8:                     insert t' into T
9:                     sup_T(t') ← sup_T(t') + sup_{T_2}(t')
10:                else
11:                    skip processing t' and all its subpatterns
12:    else
13:        foreach t' that is a subpattern of t do
14:            if sup(t') is not updated then
15:                if t' ∈ T_1 then
16:                    sup_T(t') ← sup_T(t') + sup_{T_2}(t')
17:                if t' is closed then
18:                    insert t' into T
19:                    sup_T(t') ← sup_T(t') + sup_{T_2}(t')
20:            else
21:                skip processing t' and all its subpatterns
22: delete from T patterns with support below minsup
```

- t is a subpattern of closed pattern in D_1 and a closed pattern in D_2 and it is in $\boldsymbol{c}_{D_1 \cup D_2}(t)$.

Note that these propositions, considering patterns as itemsets, are also used by a lot of algorithms that we present later.

Let's now consider D_1 as the set of transactions seen so far, whose set of closed pattern is T_1, and we want to update it with a new batch of transactions D_2, whose set of closed patterns is T_2, in order to get the set of closed patterns in $D_1 \cup D_2$. This is the principle of an *update per batch* policy, and the adding procedure is described in Algorithm A.3.
In a similar way we can update D_1 when a set of transactions D_2 is removed. The deletion procedure in pseudo-code is described in Algorithm A.4.

Algorithm A.4 Closed_Subpattern_Mining_Delete$(T_1, T_2, minsup, T)$

```
1: T ← T_1
2: foreach t ∈ T_2 in size-ascending order do
3:     foreach t' that can be obtained deleting nodes from t do
4:         if sup(t') is not updated then
5:             if t' ∈ T_1 then
6:                 if t' is not closed then
7:                     delete t' from T
8:                 else
9:                     sup_T(t') ← sup_T(t') − sup_{T_2}(t')
10: delete form T patterns with support below minsup
```

A.6 MOMENT

MOMENT was proposed by Chi et al. in [13]. It is the first paper that considers the problem of mining frequent closed itemsets over a data stream sliding window using limited memory space. It has become a reference for all the works that have been proposed after its publication.

A.6.1 Data structure

It adopts an *incremental* approach based on a in-memory prefix-tree-based data structure, called *closed enumeration tree (CET)*, to monitor a *dynamically selected* set of itemsets over the sliding window. They includes frequent closed itemsets, and itemsets that form a *boundary* between frequent closed itemsets and the rest of itemsets. Concept drifts in a data stream are reflected by status changes of itemsets that are in the boundary (e.g. itemsets that pass from non-frequent to frequent).

Similar to a prefix tree, each node in the *CET* represents an itemset I. But unlike a prefix tree, which maintains all itemsets, only the subset of closed and boundary itemsets is maintained. As long as the window size is *reasonably large* and concept drifts *not too dramatic*, most itemsets do not change their status, and only an update of itemsets support is needed. Four types of nodes, and consequently itemsets, are identified

Infrequent gateway nodes A node n_I is an *infrequent gateway node* if (1) I is an infrequent itemset, (2) its parent n_J is frequent and (3) I is obtained by joining its parent J with one of J's siblings.

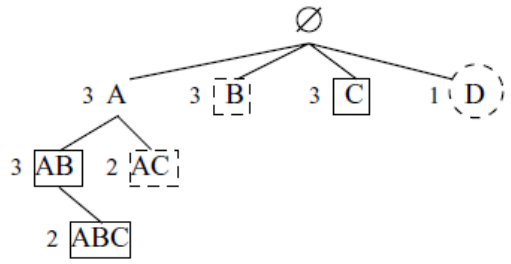
Unpromising gateway nodes A node n_I is an *unpromising gateway node* if (1) I is a frequent itemset, and (2) there exists a frequent closed

itemset J such that $J \prec I$, $J \supset I$ and $sup(I) = sup(J)$.

Intermediate nodes A node n_I is an *intermediate node* if (1) I is a frequent itemset, (2) n_I has a child node n_J such that $sup(I) = sup(J)$ and (3) n_I is not an unpromising gateway node.

Closed nodes A node n_I is a *closed node* if I is a frequent closed itemset. A closed node chan be in internal node or a leaf node.

In Figure A.1 is shown an example of CET. Infrequent gateway nodes are represented by dashed circles, unpromising gateway nodes by dashed rectangles and closed nodes by solid rectangles.



Tids t	**i**(t)
1	cd
2	ab
3	abc
4	abc

Figure A.1: Example of Closed Enumeration Tree

Each node n_i of the CET stores the information over its type, the itemsets I itself and its support, and the sum of tids of the transactions in which I occurs (tid_sum).
The latter is used to maintain an hast table which stores all the frequent closed itemsets. The pair $(support, tid\_sum)$ is used to perform the closure check and, meanwhile, to avoid a huge number of collisions that would be generated if only support were used as key.
To check if n_I is an unpromising gateway node, we simply have to hash on the $(support, tid\_sum)$ of n_I, fetch the list of frequent closed itemsets stored at the corresponding entry of the hash table, and check if there is an itemset J in the list such that $J \prec I$, $J \supset I$ and $support(J) = support(I)$.

Transactions in the sliding window are stored in and FP-tree, which is slightly different from the one proposed by Han et al. in [3] for mining frequent patterns without candidate generation. In the original FP-tree each transaction is stored along a root-path; when transactions has a common prefix, the common part is stored only once, using a counter to record the

number of times the common part is repeated. An head table is used to record the starting points of each item. An example of FP-tree is shown in Figure A.2.

In MOMENT the FP-tree is used to store *all* the transactions in the sliding window, so no pruning of infrequent items is performed. Furthermore, in addition to the head table in traditional FP-trees, another table, the tid table, is used to point each tid in the sliding window to the tail node of the corresponding itemset in the FP-tree. By using this table along the FP-tree, no further storage of transaction is needed.

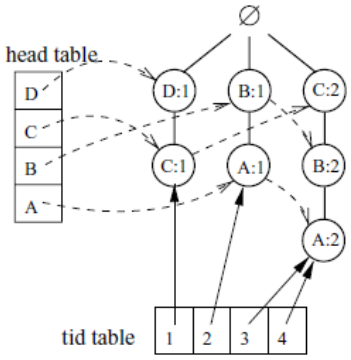


Figure A.2: Example of FP-tree

To build the CET, a depth-fist procedure *Explore* consults the FP-tree to determine the support and tid_sum of each stored itemset. The outline of this procedure is reported in Algorithm A.5. The $leftcheck(n_I)$ procedure looks up the hash table to see if there exists a previously discovered closed itemset that has the same support as n_I and which also subsumes I, and if so, returns true, otherwise false.

A.6.2 The incremental update algorithm

MOMENT uses an *update per transaction* policy. New transactions are inserted in the window by calling an *Addition* procedure, and old transactions are deleted by calling the *Deletion* procedure. Both procedures traverses the parts of the CET that are related with the incoming/outcoming transaction T, updates its *support*, *tid_sum* and its node type if necessary.

For each *incoming* transaction T, MOMENT can change the type of the related node n_I if

- n_I *was an infrequent gateway node.* If n_I now becomes frequent, then (1) for each left sibling of n_I MOMENT checks if a new children should

Algorithm A.5 Explore (n_I,$minsup$)

```
1: if sup(n_I) < minsup then
2:     mark n_I as an infrequent gateway node
3: else if leftcheck(n_I) = true then
4:     mark n_I as an unpromising gateway node
5: else
6:     foreach frequent right sibling n_K of n_I do
7:         create a new child n_{I∪K} for n_I
8:         compute support and tid_sum for n_{I∪K}
9:     foreach child n_{I'} of n_I do
10:        Explore(n_{I'},minsup)
11:    if ∃ a child n_{I'} such that sup(n_{I'}) = sup(n_I) then
12:        mark n_I as an intermediate node
13:    else
14:        mark n_I as a closed node
15:        insert n_I into the hash table
```

be created, and (2) the original pruned branch (under n_I) must be re-explored to se if new descendants should be created.

- *n_I was an unpromising gateway node.* Node n_I may become promising, so the original pruned branch must be explored.
- *n_I was an intermediate node.* Node n_I may become a closed node, if T contains I but none of n_I's children who have the same support of n_I before the addition. However, n_I cannot change to an infrequent or unpromising gateway node.
- *n_I was a closed node.* Node n_I remains closed.

In any case the Addition procedure will not decrease the number of closed itemsets in the CET. For each *outgoing* transaction T, MOMENT can change the type of the related node n_I if

- *n_I was an infrequent gateway node.* Node n_I does not change type.
- *n_I was an unpromising gateway node.* Node n_I may become infrequent, but cannot change to promising.
- *n_I was an promising node.* Node n_I may become unpromising, then *leftcheck* must be run over n_I and its descendants and left-sided siblings have to be updated consequently.

- n_I *was a closed node.* Node n_I can become non-closed and have to be removed from the hash-table. We can check it by looking at the supports of the children of n_I after visiting them. If n_I remains closed, still need to update its entry in the hash-table.

A.7 CLOSTREAM

CLOSTREAM is an algorithm for maintaining *frequent closed itemsets* in data stream. It was proposed by Yen et al. in [7]. Like CFI-STREAM, an algorithm proposed by Jing et al. in [18], CLOSTREAM is able to maintain the *complete set* of closed itemsets over a *transaction-sensitive* sliding window *without any support information.* But unlike CFI-STREAM, CLOSTREAM does not need a costly subset generation for each processed transaction.

A.7.1 Data structure

CLOSTREAM uses two in-memory data structures called *Closed Table* and *Cid List* respectively, plus an additional hash table.
The Closed Table maintains the information about closed itemsets. Each closed itemsets CI is associated to an unique identifier Cid and its support count SC.
Cid List is used to maintain items and their *cidsets*. Each item $x \in \mathcal{I}$ is associated to a set named *cidset*. The $cidset(x)$ contains all $Cids$ of x's super closed itemsets.
As a new closed itemset Y is found, CLOSTREAM puts it into the Closed Table and assigns it a cid c. Then c is added into the cidsets of all the items $y \in Y$ in the Cid List. An example of these two data structures is shown in Table A.1.

A hash table, called $Temp_A$, is used to put those itemsets that need to be updated as a transaction arrives. It uses a $(TI, Closure\_Id)$ as a pair $(key, value)$. Once an itemset $X \subseteq \mathcal{I}$ needs to be updated, it is put in the TI field. The cid of its closure is stored in the as its Closure_id value.

A.7.2 The incremental update algorithm

CLOSTREAM uses an *update per transaction* policy to maintain updated the set of closed itemsets into the current sliding window. Two procedures called *CloStream+* and *CloStream-* are used when a transaction arrives and when a transaction leaves, respectively.

Tid t	**i**(t)
1	cd
2	ab
3	abc
4	abc
5	acd
6	bc

(a) Sample data stream.

Cid	*CI*	*SC*
0	0	0
1	cd	2
2	ab	3
3	abc	2
4	c	4
5	acd	1
6	a	4
7	ac	3

(b) Original Closed Table.

Item	*Cidset*
a	{2,3,5,6,7}
b	{2,3}
c	{1,3,4,5,7}
d	{1,5}

(c) Original Cid List.

Table A.1: Example of CLOSTREAM data structures for $W = \{t_1, \ldots, t_5\}$.

To perform these procedures, some properties of the closure operator, that we previously described in Equation A.1, are exploited. In particular, given two itemsets $X, Y \subseteq \mathcal{I}$ (no detailed proof is provided)

Property 1 If $\boldsymbol{c}(X) = Y$, then $sup(X) = sup(Y)$.

Property 2 X is closed if and only if $\forall Z \supset X, sup(X) > sup(Z)$.

Property 3 If $X \subseteq Y$, then $\boldsymbol{t}(Y) \subseteq \boldsymbol{t}(X)$ and $\boldsymbol{c}(X) \subseteq \boldsymbol{c}(Y)$.

Property 4 $\boldsymbol{i}(\boldsymbol{t}(X)) \cap \boldsymbol{i}(\boldsymbol{t}(Y)) = \boldsymbol{i}(\boldsymbol{t}(X) \cup \boldsymbol{t}(Y))$.

From these properties and using the idempotency of the closure operator, the following theorem can be derived (no proof is provided)

Theorem 1 For any two closed itemset X and Y, if $(X \cap Y) = S$ and $S = \emptyset$, then S is a closed itemset.

When a transaction $T_A = \langle t_A, X \rangle$ arrives, the procedure *CloStream+* is called. First of all $SC(X)$ is increased and, from Property 2, is surely closed. Only its subsets have to be updated to find new closed itemsets. According to Theorem 1, they can be found by intersecting X with all the closed itemsets in the Closed Table.
To perform this operation efficiently, CloStream+ uses the Cid List and a *SET function* defined as $SET(X) = cidset(x_1) \cup \ldots \cup cidset(x_k)$, for $X = \{x_1, \ldots, x_k\}, i_i \in \mathcal{I}$. Each Cid in $SET(X)$ represents a closed itemset which has at least one common item with X. This allows to reduce the number of itemset X can be intersected with.

By performing intersection operations on X and every closed itemset whose Cids are in $SET(X)$, closed subsets of X and respective closures

can be identified. CloStream+ uses a temp table, denoted as $Temp_A$, to store the closed subsets of X and their closures. $Temp_A$ is a hash structure and consists of two fields $(UItemset, Closure\_Id)$, where $UItemset$ is the itemset which needs to be updated and $Closure\_Id$ is the identifier of its closure. After that all closures have been identified, support counts are increased and new closed itemsets are added.

$UItemset$	$Closure\_Id$
bc	3
c	4
b	2

(a) Table $Temp_A$ after adding transaction t_6.

$DItemset$	$Closure\_Id$	HS
cd	1	5
c	4	7

(b) Table $Temp_D$ after adding transaction t_1.

Table A.2: Examples of temp table for updating the sliding window.

The deletion of a transaction is treated in as similar fashion by procedure *CloStream-*. When a transaction $T_D = \langle t_A, X \rangle$ is deleted, only the support counts of its subsets will be decreased by one, that is, only its subsets may be changed from closed to non-closed.
As before, closed subsets may be found via intersection, and to perform efficiently this operation SET function is used. CloStream- uses a temp table, named $Temp_D$, to store information about closed subsets of X. It consists of the the triple $(DItemset, Closure\_Id, HS)$, where $DItemset$ is a closed subset of X, $Closure\_Id$ is the identifier of its closure and HS is the identifier of a closed superset of $DItemset$ (it is used to check if $Ditemset$ is closed).

As T_D is deleted from the window, X is inserted into $Temp_D$ and its closure are found with the aid of $SET(X)$ function. At the same time CloStream- verifies whether exists a closed superset of the itemsets int temp table. Then, consulting $Temp_D$, the support of the itemsets that are still closed is decreased and the others are deleted from the Closed Table. It Table A.2 are shown two examples of temp table for a sliding window of size 5.

A.8 NEWMOMENT

NEWMOMENT is method to maintain *frequent closed itemsets* in data streams with a *transaction-sensitive* sliding window proposed by Li et al. in [11]. As suggested by the name, this is an improved version of the MO-

MENT algorithm that we describe in Section 3.3. It uses an effective *bit-sequence* representation of items to reduce time and memory consumption. Also a new prefix tree structure called *NewCET*.

For each item $x \in \mathcal{I}$ in the current window W, a *bit-sequence* $Bit(x)$ of $w = |W|$ bits is constructed. The ith bit of $Bit(x)$, $1 \leq i \leq w$, is set to 1 if and only if x belongs to the ith transaction of the W, otherwise it is set to 0.

tid t	**i**(t)
1	abc
2	bcd
3	abc
4	bc
5	bd
6	cd

(a) Data stream example.

item	W_1	W_2	W_3
a	1010	0100	1000
b	1111	1111	1110
c	1111	1110	1101
d	0100	1001	0011

(b) Bit-sequences associated to each item.

Table A.3: Data stream and bit-sequence examples for a window size $w = 4$.

In Table A.3 is shown an example data stream and the consequent bit-sequences that can be computed for 3 sliding windows of size $w = 4$. Bit-sequence representation allows an efficient window sliding process. Transaction *deletion* can be performed simply with a *left shift* of one bit of each stored sequence. After deletion, to *add* a new transaction $T = \langle t, X \rangle$ one have to set to 1 the most right bit of a sequence associated to each item x if $x \in X$, otherwise set it to 0.

A.8.1 Data structure

NEWMOMENT uses an extended prefix tree data structure, named *NewCET* (New Closed Enumeration Tree), which consists of three parts

1. The bit-sequences of all 1-itemsets in the current transaction-sensitive window W.
2. The set of frequent closed itemsets in W.
3. A hash table to perform closure check, using support as key.

Essentially the structure is really similar to the CET of MOMENT, a part that *only* frequent closed itemsets are stored into the prefix tree. Each node n_I in the NewCET has the corresponding bit-sequence $Bit(I)$ to store the support information in W. Figure A.3 shows an example of NewCET.

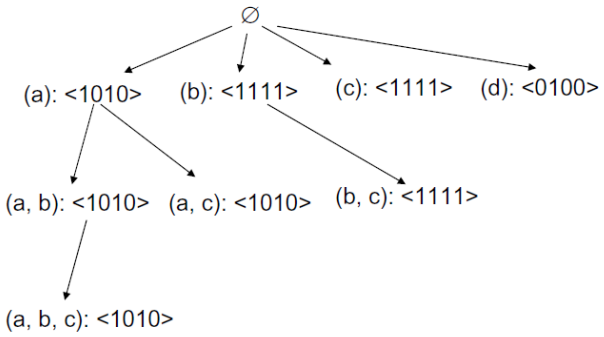


Figure A.3: Example of NewCET.

A.8.2 The incremental update algorithm

The *Build* procedure to initialize the NewCET, shown in Algorithm A.6, performs a depth-first search along the current NewCET and is really similar to the *Explore* procedure of Moment, detailed in Algorithm A.5. One difference is in child generation, which is now performed efficiently via a bitwise AND over bit-sequences. Another important difference is that only frequent closed itemsets are retained into the tree, and at the end all the bit-sequences of k-itemsets, $k > 1$, are deleted, maintaining only their supports. As common function $leftcheck$ performs the closure check by consulting the hash table.

Algorithm A.6 Build(n_I,$minsup$)

```
1: if sup(n_I) ≥ minsup then
2:     if leftcheck(n_I) = false then
3:         foreach frequent sibling n_K of n_I do
4:             generate a new child n_{I∪K} for n_I
5:             Bit(I ∪ K) ← BitwiseAND(I, K)
6:         foreach child n'_I of n_I do Build(n'_I,minsup)
7:         if ∃ a child n'_I such that sup(n'_I) = sup(n_I) then
8:             retain n_I as frequent closed itemset
9:             insert n_I into the hast table
```

Also deletion and addition, shown in Algorithm A.7 and A.8 respectively, procedures results simplified respect to MOMENT ones. As done in MOMENT, NEWMOMENT adopts an *update per transaction* policy. In deletion, first all bit-sequences of 1-itemsets are left-shifted of 1 bit, and then *Delete* procedure is called to maintain the NewCET updated, includ-

Algorithm A.7 Delete(n_I,$minsup$)

```
1: if n_I is not relevant to the deleted transaction then
2:     return
3: else if sup(n_I) ≥ (minsup − 1) then
4:     foreach sibling n_K of n_I such that sup(n_K) ≥ (minsup − 1) do
5:         generate a new child n_{I∪K} for n_I
6:         Bit(I ∪ K) ← BitwiseAND(I, K)
7:     foreach child n'_I of n_I do
8:         Delete(n'_I,minsup)
9:     if sup(n_I) ≥ minsup then
10:        if leftcheck(n_I) = false then
11:            if n_I was previously closed then
12:                update sup(n_I)
13:                update n_I in the hash table
14:        else
15:            retain n_I as a frequent closed itemset
16:            insert n_I into the hash table
17:    else
18:        if n_I was previously closed then
19:            mark n_I as non-closed
20:            remove n_I from the hash table
21: else
22:    if n_I was previously closed then
23:        mark n_I as non-closed
24:        remove n_I from the hash table
```

ing closed itemsets whose support is greater than $minsup - 1$.

In addition, every bit-sequence of 1-itemset sets its rightmost bit accordingly to the incoming transaction. Then the procedure *Append* is called, which is almost the same as *Build*, a part that the former checks whether a closed itemset is already in the NewCET.

A.9 CLAIM

CLAIM is an algorithm for Closed Approximate frequent Itemset Mining proposed by Song et al. in [10]. They propose an *approximate* frequent closed itemset model, called *relaxed frequent closed itemsets* (RC). To accelerate the drifted itemsets update, all frequent relaxed closed itemsets with

Algorithm A.8 Append(n_I,$minsup$)

```
1: if sup(n_I) ≥ minsup then
2:     if leftcheck(n_I) = false then
3:         foreach frequent sibling n_K of n_I do
4:             generate a new child n_{I∪K} for n_I
5:             Bit(I ∪ K) ← BitwiseAND(I, K)
6:         foreach child n'_I of n_I do
7:             Append(n'_I,minsup)
8:         if ∃ a child n'_I such that sup(n'_I) = sup(n_I) then
9:             if n_I was previously closed then
10:                update sup(n_I)
11:                update n_I in the hash table
12:        else
13:            retain n_I as a frequent closed itemset
14:            insert n_I into the hash table
```

the same support are arranged by one *bipartite graph model*. A *Bloom filter* based hash method is introduced to match drifted itemsets in bipartite graph. Both previous mechanisms are combined in a compact tree structure, named *HR-tree*.

The *relaxed closed itemsets* model (RC) is introduced to reduce the maintenance cost of closed itemsets in a data stream. Because of the nature of closed itemsets, the support of closed itemsets *exact equals* to the support of absorbed itemsets, but any little concept drift can lead to changing of closed itemsets, and thus to high cost of maintenance. Let call σ, $0 \leq \sigma \leq 1$, a minimum support threshold, and define two concepts

Relaxed Interval The support space of all itemsets can be divided into $n(= \lceil 1/\epsilon \rceil)$, where ϵ is a user-specified relaxed factor, and each interval can be denoted by $I_i = [l_i, u_i)$, where $l_i = (n-i) * \epsilon \geq 0$, $u_i = (n-i+1) * \epsilon \leq 1$ and $i \leq n$.

Relaxed Closed Itemset An itemset X is called a *relaxed closed itemset* if and only if there exists no *proper superset* X' of X such that they belong to the same interval I_i.

For a relaxed closed itemset X with $sup(X) \in I_i(= [l_i, u_i))$, if $l_i \geq \sigma$, then X is *frequent* with *frequent interval* I_i, otherwise X is *infrequent* with *infrequent interval* I_i. An interval I_i with $l_i < \sigma \leq u_i$ is called a *critical interval*, and it is divided into two intervals, an infrequent interval $[l_i, s)$ and frequent

interval $[s, u_i)$. The set of RC in a relaxed interval is also called the *upper bound* of the interval. In such case, only maintaining this upper bound cannot track the bound drift efficiently. Thus, for each interval a *lower bound* is defined. Symmetrically to the upper bound definition, an itemset X belongs to the lower bound of a relaxed interval if and only if there exists no *proper subset* X' of X such that they belong to the same interval I_i. This leads to a *double bound* representation of frequent closed itemsets.

tid t	**i**(t)
1	abcde
2	cde
3	abce
4	acde
5	abcde
6	bcd
7	bce

Table A.4: Stream dataset example

I_i	Itemsets	Lower bound	Upper bound
$I_1 = (0.8, 1]$	c, d, e, cd, ce	c, d, e	cd, ce
$I_2 = (0.6, 0.8]$	a, b, ac, ae, bc, de, ace, cde	a, b, de	bc, ace, cde
$I_c = [0.45, 0.6]$	ab, ad, be, abc, abe, acd, ade, bce, abce, acde	ab, ad, be	abce, acde
$I_n = [0, 0.45)$	abd, bde, abcd, abde, bcde	abd, bde	abcde

Table A.5: Double bound of relaxed closed itemsets for $\epsilon = 0.2$ and $\sigma = 0.45$.

Table A.5 shows an example of double bound of relaxed closed itemsets extracted from transactions 1 to 6 of Table A.4. The traditional closed itemset mining can be obtained by setting $\epsilon = 1 \backslash |W|$, while maximal frequent itemset is achieved by setting $\epsilon = 1$.

To efficiently update this double bound representation, a data structure, the *bipartite graph* is introduced. A bipartite graph $BG = (U, L, E)$ has to distinct vertex sets U and L with $U \cap L = \emptyset$, and edge set $E = \{(u, l) | u \in U \wedge l \in L\}$. For an interval I_i, $U_i(L_i)$ is a subset of the upper bound (lower bound). The edge $e(u, l) \in E(u \in U_i, l \in L_i)$ means that there exists an *including* relationship between itemset u and l, $u \supset l$.

The usage of a bipartite graph helps the update process of RC. For example, when an itemset l of the lower bound is deleted, it has to be substituted by its super-pattern l', we it should be generated by scanning each itemset u in the upper bound satisfying $l' \subseteq u$. BG edges represent this relationship, avoiding an $O(n)$ control over all the n elements of the upper bound for each deleted itemset.

A.9.1 Data Structure

In CLAIM a *compact prefix tree* based structure, named *HR-tree* (Hash based Relaxed Closed Itemset tree), is proposed. It allows to arrange bipartite graphs using a combination of two common data structures: a *hash table* and a *prefix tree*.
In the hash table, each interval corresponds to a bipartite graph and each graph has its two hash function entries: $H_i(BF_u, X)$ for the upper bound and $H_i(BF_l, X)$ for the lower bound. Every entry corresponds to a pointer which points to the itemset list in the upper or lower bound of BG in the prefixed tree. As every prefix tree, each node corresponds to one itemset with its support.
For any bipartite graph $BG = (U, L, E)$, an itemset X, $X \notin BG$, belongs to set BF_u if and only if X is contained by at least one element $Y \in U$, but there exists no itemset $X' \notin BG$ such that $X \subset X' \subset Y$. It can be shown that an upper bound drifted itemset X is contained by BG if and only if $X \in BF_u$.
In the same way in itemset X, $X \notin BG$, belongs to set BF_l if and only if X is contained by at least one itemset $Y \in L$, but there exists no itemset $X' \notin BG$ such that $X \supset X' \supset Y$. It can be show that a lower bound drifted itemset is contained by BG if and only if $X \in BF_l$. An example of bipartite graph structure is shown in Figure A.4.

Hash functions directly answers whether a drifted itemset X belongs to the BG by querying sets BF_u and BF_l, making unnecessary to access to all related itemsets in the prefix tree. To ease this *belonging* queries, a *Bloom filters* based hash method is used.
A Bloom Filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set [17]. In CLAIM the following hash function is used

$$H(X) = \{h_i(X) | h_i(X) = mod(F(X, \mathcal{I})^i, m)\}, 0 < i \leq K \tag{A.2}$$

where $F(X, \mathcal{I})$ is a function that associate an integer value to each binary-

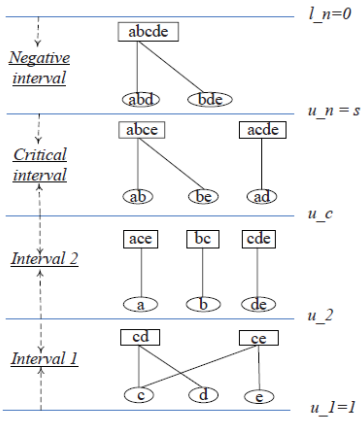


Figure A.4: Example of bipartite graph model.

valued itemset $X \subset \mathcal{I}$ in this way

$$F(X, \mathcal{I}) = \sum x \in X 2^{(pos(x)-1)} \tag{A.3}$$

where $pos(x)$ refers to the position of an item x in $\mathcal{I}$ in right first order.

A.9.2 The incremental update algorithm

Algorithm A.9 shows CLAIM algorithm in pseudo-code. It uses an *update per transaction* policy, where only the *upper bound drift* of a bipartite graph is cared at the *insertion* of a transaction, and only the *lower bound drift* is cared at the *deletion* of a transaction.

Initially the HR-tree is *null* and the relaxed intervals are generated accordingly to their definition. The set of relaxed closed itemsets RC is initialized by the insertion of stream elements until the sliding window is full. For each incoming transaction, or stream element, support counting is performed scanning the HR-tree starting from the *root* in prefix order, and drifted itemsets can be collected directly. But there are drifted itemsets that do not appear directly in the double bound, named *internal drifted itemsets*, since no support information is recorded. Thus, all the subsets of a drifted itemset X is analyzed to check whether it is a drifted itemset or not.

After that, all internal drifted itemset is deleted from the original bipartite graph, which can be decomposed into serval independent bipartite graphs possibly, and then recombined using with the aid of drifted itemset location via hash function.

Algorithm A.9 Claim(W,ϵ,σ)

1: $HR\text{-}tree \leftarrow null$, $I \leftarrow \bigcup_0^{\lceil 1/\epsilon \rceil - 1} I_i$, RC =initialize(W)
2: **foreach** incoming stream element $e \in W$ **do**
3: $drifted - itemset \leftarrow$scan($HR - tree$,$e$)
4: **foreach** $X \in drifted - itemset$ belonging to interval i **do**
5: /* Find all internal drifted itemsets contained by X in BG */
6: $BoundSet \leftarrow X$
7: **foreach** itemset $x \subset X$ in BG **do**
8: $sup(x) \leftarrow$explore(x,W)
9: **if** $(sup(x) = sup(X) \wedge \exists p \in BoundSet | x \in p)$ **then**
10: $BoundSet \leftarrow (BoundSet - p)$,$BoundSet \leftarrow BoundSet \cup x$
11: /* Bipartite graph decomposition */
12: $BG \leftarrow$Removed drifted itemsets in BG
13: **if** $BG_1^i \cap \ldots \cap BG_m^i = null$ **then**
14: BG is decomposed into $BG_1^i, \ldots, BG_m^i$ with HR-tree adjustment
15: /* Bipartite graph combination */
16: **foreach** $x' \in BoundSet$ **do**
17: /* Drifted itemset location */
18: $BG_1^{i-1}, \ldots, BG_n^{i-1} \leftarrow H_{i-1}(BF_l, x')$
19: **if** $n > 1$ **then**
20: $BG_{new} \leftarrow$combination$(BG_1^{i-1}, \ldots, BG_n^{i-1}, x')$ with HR-tree adjustment
21: **foreach** outgoing stream element $e \in W$ **do**
22: /* Similar to previous part, details are omitted here. */