An Efficient Closed Frequent Itemset Miner for the MOA Stream Mining System

Massimo Quadrana
U. Politècnica de Catalunya (BarcelonaTech)
and Politecnico di Milano
massimo.quadrana@polimi.it

Albert Bifet
Yahoo! Research
abifet@yahoo.com

Ricard Gavaldà
U. Politècnica de Catalunya (BarcelonaTech)
gavalda@lsi.upc.edu

November 15th, 2013

Abstract

We describe and evaluate an implementation of the IncMine algorithm due to Cheng, Ke, and Ng (2008) for mining frequent closed itemsets from data streams, working on the MOA platform. The goal was to produce a robust, efficient, and usable tool for that task that can both be used by practitioners and used for evaluation of research in the area. We experimentally confirm the excellent performance of the algorithm and its ability to handle concept drift. We also provide a PAC-style rigorous analysis of the quality of the output of IncMine as a function of its parameters.

Keywords: Data mining, data streams, stream mining, itemset mining, MOA

1 Introduction

Computing frequent itemsets is a central data mining task, both in the static and the streaming scenarios. Important research effort has produced a substantial number of methods for the streaming case, and the problem is relatively well understood now. We noticed, however, that there are almost no public, easy-to-use implementations of the streaming methods described in the literature, a situation that effectively prevents their application in practice and conditions further research.

The aim of this paper is to describe a robust, efficient, usable, and extensible implementation for mining frequent closed itemsets over data streams, working over the MOA framework. We believe that this constitutes a double contribution: On the one hand, it will allow many practitioners to actually use itemset mining techniques in many streaming scenarios with mild learning curves given MOA's user-friendly and public character. On the other hand, for the research community, it provides a state-of-the-art implementation which may be used as reference for evaluation of new methods. It may constitute a first element in a future repository of stream itemset mining method, analogous to the one in [20] for batch itemset mining.

After thorough examination of several algorithms in the literature, we decided to implement the IncMine algorithm of Cheng *et al.* [3], for reasons to be explained. Two main gaps had to be filled in the implementation with respect to the original description: One was the batch method to mine itemsets in the successive stream segments in which IncMine processes the stream; we used the efficient implementation in [7] of the CHARM algorithm [16, 17]. The other was how to perform a certain merging operation of inverted index lists, and we selected a particular method reported to be often best in the multiple set intersection literature. Additionally, our implementation is able to deal with ease with evolving data streams, in the form of both abrupt and gradual changes.

MOA (Massive Online Analysis) [1, 19] is a data stream mining framework developed by the U. Waikato. Implementing on top of MOA ensures portability and maintainability, as well as not having to implement from scratch the basic stream processing primitives. Let us note that there was already a MOA extension, due to M. Jarka, implementing the MOMENT method [5] for frequent closed itemset mining. However, it is reported in [3] and we confirm here that IncMine is typically much faster than MOMENT with only minor loss in output quality.

We evaluate our implementation on both synthetic and real data, all containing concept drift. For the synthetic ones, we study the influence of the parameters of the algorithm on accuracy, throughput, and memory usage, as well as how it reacts and adapts to (known, measurable) concept drift. We also test our solution over a data stream generated from real data from the MovieLens database, obtaining intuitively appealing results.

In addition, we provide a theoretical analysis of the quality of the output of IncMine as a function of the several parameters of the algorithm, which matches well with the results obtained. The analysis is in the formal PAC learning model [9] and requires moderate probabilistic assumptions on the source generating the transaction stream.

The software described here is available from the MOA project site [19] as a MOA extension since September 2012.

2 Background

In this section we recall the definitions of *Frequent (Closed) Itemset Mining* and related concepts. We survey the main batch methods and those for data streams, highlighting the differences that determined our choice of one to be implemented. Finally we present the essentials of the MOA framework on which our implementation runs.

2.1 The Frequent Itemset Mining Problem

The discovery of frequent itemsets is one of the major families of techniques for characterizing data. Its goal is to find correlations among data attributes, and it is closely linked to association rule discovery.

Let $\mathscr{I} = \{x_1, x_2, \ldots, x_m\}$ be a set of binary-valued attributes called *items*. A set $X \subseteq \mathscr{I}$ is called an *itemset*. A *transaction* is a tuple of the form $\langle t, X \rangle$, where $t \in \mathscr{T}$ is a unique transaction identifier (tid) and X is an itemset. A binary database $\mathscr{D}$ is a set of transactions, all with distinct tids. We say that transaction (t, X) contains item x if $x \in X$. For an itemset X and an implicit $\mathscr{D}$, $\boldsymbol{t}$(X) is the set of transactions that contain *all* the items in X. In particular, $\boldsymbol{t}$(x) is the set of tids that contain the single item $x \in \mathscr{I}$.

The *support* of an itemset X in a dataset $\mathscr{D}$, denoted $supp(X, \mathscr{D})$, is the number of transactions in $\mathscr{D}$ that contain X, or $supp(X, \mathscr{D}) = |\boldsymbol{t}(X)|$. The *relative* support of X in $\mathscr{D}$ is $rsupp(X, \mathscr{D}) = |\boldsymbol{t}(X)|/|\mathscr{D}|$. Relative support also makes sense when $\mathscr{D}$ is a distribution over all itemsets: it is the probability that an itemset Y drawn according to $\mathscr{D}$ is such that $X \subseteq Y$.

Fix some user-defined minimum support threshold *minsupp*. Then X is said to be *frequent* in $\mathscr{D}$ if $supp(X, \mathscr{D}) \geq minsupp$. When there is no confusion about $\mathscr{D}$ and *minsupp* we will drop them and simply say "X is frequent" and write its support as $supp(X)$. We use σ to denote the relative support equivalent to *minsupp*, i.e. $\sigma = minsupp/|\mathscr{D}|$.

The frequent itemset mining problem is that of computing all frequent itemsets in the database, w.r.t. a user-specified *minsupp* value. The seminal Apriori algorithm [2], ECLAT [15], and FP-GROWTH [8] are three of the best known proposals for the task.

The search space for frequent itemsets often grows exponentially with the number of items, and furthermore the frequent itemsets themselves are often many and highly redundant, which makes the *a posteriori* analysis tedious and difficult. Several approaches for focusing on the interesting itemsets have been proposed. Here we consider frequent *closed* itemsets.

A frequent itemset $X \in \mathscr{F}$ is *closed* if it has no frequent superset *with the same support*. A most important property is that although in practice there are far fewer frequent closed itemsets than frequent itemsets, the latter set (and the supports) can be computed from the first (and the supports). To be precise, an itemset is frequent if and only if it is a subset of some frequent closed itemset. Consequently, algorithms that obtain the frequent closed sets directly without internally generating all frequent itemsets provide essentially the same information with potentially large savings in computational resources and less redundant output. Among the several methods in the literature for batch mining frequent closed itemsets we mention CLOSET [11], CHARM [16], and CLOSET+ [13].

Frequent itemset mining in data streams is a relatively new branch of study in data mining. Roughly speaking, the goal is the same as in the batch case, except that the set of desired frequent closed patterns is defined not with respect to a fixed database of transactions but with respect to an imaginary window W over the stream that shifts (and perhaps grows or shrinks) with time. We thus adopt notations such as the support on a database, $supp(X, \mathscr{D})$, to this scenario in the natural way, e.g. $supp(X, W)$.

Several different approaches were proposed in the last decade. Most of them can be classified according to the window model they adopt or other features. The window may be *landmark* (contains all elements since time 0) or *sliding* (contains only some number of most recent elements); it may be *time sensitive* (contains stream elements arrived in the last T time units) or *transaction sensitive* (contains the last N items, no matter how spaced in time they have arrived), may do *updates per transaction* or *updates in batches*. Most importantly, may be *exact* or *approximate* depending on whether it will produce the exact set of desired patterns or whether it may have *false positives* and/or *false negatives.* Exact mining requires tracking all items in the window and their exact frequencies, because any infrequent itemset may become frequent later in the stream. However, that quickly becomes infeasible for large windows and fast data streams, and approximate mining is sufficient for most scenarios.

A somewhat outdated survey on frequent itemset mining on data streams (both closed and general) is [4].

2.2 Choosing a method for Frequent Closed Itemset Mining on Streams

We next mention some of the most important methods discussed in the literature, highlighting their features relevant for our choice of a method to implement. We discuss only sliding-window approaches, since landmark-window ones cannot be expected to deal with concept drift.

MOMENT, proposed by Chi *et al.* in [5], was the first for *incremental* mining of closed frequent itemsets over a data stream, and perhaps for that reason has become a reference for all solutions proposed later. It is an *exact* mining algorithm, using a sliding window and an *update per transaction* policy. 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 [8], to store the information of *all* transactions in the sliding window, with no pruning of infrequent itemsets.

CLOSTREAM, proposed by Yen *et al.* in [14], maintains the *complete set* of closed itemsets over a *transaction-sensitive* sliding window *without any support information*. It 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 temporary hash tables to perform an efficient update. CLOSTREAM does not easily handle concept drift, since all closed itemsets in the (possibly long) sliding window are equally considered, even if a change has occurred within the window.

NEWMOMENT, proposed by Li *et al.* [10], maintains a *transaction-sensitive* sliding window and uses bit-sequence representations to reduce time and memory consumption w.r.t. MOMENT. Also, it uses a new type of closed enumeration tree (*NewCET*) to store *only* the set of frequent closed itemsets into the sliding window. Otherwise, it inherits most characteristics of MOMENT, such as *update per transaction* policy and *exactness*.

IncMine, proposed by Cheng *et al.* in [3], offers an *approximate* solution to the problem, using a *relaxed minimal support threshold* to keep an extra set of infrequent itemsets that are likely to 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 approximate set of frequent closed itemsets over the current sliding window, which results in a much better average time-per-transaction, at the risk of temporarily loosing accuracy of the maintained set while each batch is being collected. 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*. A more detailed description of IncMine is given in the next section.

CLAIM was proposed by Song *et al.* in [12] for *approximate* mining using a *transaction-sensitive* sliding window. The authors define the concepts of *relaxed interval* and *relaxed closed itemset*, in order to 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*.

We can now compare the algorithms in order to choose one for our implementation. MOMENT's main drawback is that it internally stores *all* transactions in a modified FP-tree, with considerable memory overhead, and the data structure is optimized for the case in which change is very rare. NEWMOMENT partially improves this problem. But in any case exact methods (MOMENT, NEWMOMENT, CLOSTREAM) pay a large computational price for exactness. Among the two approximate ones we considered, IncMine and CLAIM, CLAIM is described in the paper as performing update-per-transaction, and we did not see evidence that the relatively complex update rules would translate to better performance than IncMine's approach, even if we changed CLAIM to batch updates. We thus chose

IncMine for implementation on MOA.

2.3 MOA

MOA (Massive Online Analysis) [1, 19] is a data stream mining framework developed by the U. Waikato. It is closely related in spirit and structure to the popular WEKA framework for batch data mining.

We decided to implement on top of the MOA framework for a number of reasons, including:

- It is the most complete public framework for stream mining, with a fast-growing user base.
- It is implemented in Java, which ensures portability, with both API and GUI interfaces intended to hide much of the process complexity to the user.
- It provides substantial help for developers, as most of the stream-managing functionalities are already there, and researchers, as it provides also functionality for synthetic data generation and evaluation.
- No particular running environment or data source is assumed: any kind of itemset stream that can be passed to MOA via its API can be processed.

As a downside, currently MOA runs in a single machine (no support for parallel or distributed processing), with the consequent limitation in processing speed and memory.

3 IncMine and our Implementation

IncMine [3] is an algorithm for incremental update of *frequent closed itemsets* (*FCIs*) over a high-speed data stream. We first provide a high-level description of the algorithm, at the level required to understand the performance analysis in Section 4, and then indicate three points where we departed from or completed the original description. More detail can be found in the original paper.

3.1 A high-level description of IncMine

The main features of IncMine are:

- It is 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. Itemsets near (below) the support threshold may or may not be reported as frequent, i.e., be false negatives.

- It uses a *time-sensitive* approach: the sliding window contains the elements arrived in the last T steps, be it none or many.

- It also introduces the notion of *semi-FCIs*, which associate a progressively increasing minimal support threshold for an itemset that is retained longer in the window. This way, FCI's that were once frequent have to keep proving their high frequency to be retained. This, together with the use of the window in itself, contributes to IncMine handling of concept drift.

- It builds and maintains an *inverted index* of the semi-FCIs to efficiently perform updates.

- It performs batch rather than per-transaction updates. As discussed, this is crucial to have reasonable efficiency, and in many cases one can assume stationarity in moderately long segments of the stream, or live with slight inaccuracies for short transitory periods. Within each segment, itemsets in the segment are mined using some batch method and then the result is used to update a global data structure.

Let us now give a high-level description of IncMine, aiming at the analysis in Section 4.

IncMine uses a *time-sensitive* sliding window of transactions, that is, it aims at reporting the closed itemsets that are frequent in a recent time window of duration W. Let L denote the set of FCI's mined at any given time from the current time window. At each *time unit*, a new set of transactions B arrives and L must be updated to reflect the transactions in B and forget the effect of the transactions received W time units ago. Roughly speaking, IncMine performs this by first mining the set C of FCI's in B, and then updating L with the contents of C, according to a clever set of rules which also implement the forgetting of expired transactions.

However, a direct implementation of this idea is costly. The number of itemsets that have to be stored in order to perform this task exactly can grow to be very large, because even itemsets that seem very infrequent at this time have to be kept in L, just in case they start appearing more often now and become frequent in the window within the next W time units.

Exact algorithms are bound to be costly by this requirement. In contrast, IncMine adopts the following heuristic to cut down on memory and computation. Recall that an itemset X is frequent on a window containing N transactions if it has support at least σN there. Imagine, say, that in the last $W/2$ time units we have received N transactions and that an itemset has been seen in only $\sigma N/10$ of them. It is possible that in the next $W/2$ time units X appears sufficiently frequently to achieve relative support σ, but it seems unlikely. One is tempted to declare X *non-promising* at this point in time and drop it from L. This creates the possibility that that X is a *false negative* $W/2$ times units later, i.e., it is frequent but not reported.

One can then use a *relaxation parameter* $r \in [0,1]$ and declare that all itemsets not having relative support $r\sigma$ in at some point are dropped. IncMine takes this idea a bit further, by noticing that the longer an itemset has been in the window, a higher relaxation parameter should be used. That is, to keep X promising for a window W one should require a higher relative support for X after seeing $3/4$ of the elements in W than when when only $1/4$ of them have been seen, as in the latter X has more time to catch up with the required minimum support. Thus, Incmine uses r to define an increasing sequence of supports as follows:

- For $k \in [1..W]$, define $r(k) = (k-1) \cdot (1-r)/(W-1) + r$. Observe that $r(1) = r$ and $r(W) = 1$.

- for any two time units a, b, let $T_{a..b}$ be the set of time units comprised between a and b, and $N_{a..b}$ the number of transactions received during $T_{a..b}$.

- X is a *semi-FI* at any given time t if there is a $k \in \{1..W\}$ such that $supp(X, T_{t-k+1..t}) \geq r(k) \cdot \sigma N_{t-k+1..t}$. In words, if it was $r(k)\sigma$ frequent at some point during the last W time units. Furthermore, it is a *semi-FCI* if in addition it is closed w.r.t. the set of transactions in $T_{t-W+1..t-k+1}$.

- At any time t, an itemset may be dropped from L because it does not seem promising: if for every $k \in \{1..W-1\}$ its frequency is lower than $r(k)\sigma$, it seems unlikely that it will become σ-frequent in the next 1 or 2 or ... W time steps, so it is dropped.

- The set L kept by IncMine at time t is, precisely speaking, the set of semi-FCI in the window $T_{t-W+1..t}$ or, in words, the set of FCI's that have not been dropped as unpromising during the last W time units.

It is clear thus that a key part of the algorithm is the update procedure for maintaining the set L of FCI's updated in this way. We omit its somewhat lengthy description here and refer to the original paper, since it is not required for understanding neither the analysis nor the experiments that follow.

3.2 Some details of our implementation

We departed from the description in [3] (or completed it) in the following three points:

1. Window type. We decided to implement a *transaction-sensitive* window instead of the *time-sensitive* window proposed in [3], mainly to ease our testing (as it is easier to compare performance when sliding windows have fixed size). It is also the norm in MOA.

2. Batch miner. We had to choose a particular batch method for mining a given segment for frequent closed sets. Our choice was the CHARM method [16], which is available as part of the Sequential Frequent Pattern Mining framework [7], a package for sequence, itemset, and association rule mining available under GPL3 License. In fact, it provides two versions of CHARM, the original one and an improved one which uses bitsets to represent transactions. We used the improved, bitset-based one as it provided better performance in our tests. We intend to replace it with an independent, standalone version of CHARM in the future.

3. Inverted indexing. One of IncMine's most sophisticated contributions is the *Inverted Index Structure* to manage efficiently all the semi-FCIs stored in the sliding window. Each set is partitioned accordingly to the size of the semi-FCIs in the last window. 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 semi-FCI is deleted from an FCI-array, its ID is pushed into the garbage queue. When a new semi-FCI 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 semi-FCI is appended to the array. 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 $\mathscr{I}$ in lexicographical order, and, associated to 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 semi-FCIs in ascending order of their integral values (a *size-n ID-array*).

With this structure we can, given an itemset X, efficiently get its position in the corresponding FCI-array, select its Smallest Semi-FCI Superset (SFS), and insert or delete it. The efficiency of the inverted indexing comes from the efficiency and simplicity of joining two *sorted* arrays, which is simple and fast. But when several sorted arrays have to be joined into the inverted index, the order for pairwise (or k-wise) joining has a significant impact on efficiency, and the policy is not discussed in [3]. Luckily, the problem has been extensively studied, for example in the Information Retrieval field. Culpepper *et al.* in [6] provide a survey of algorithms for efficient multiple set intersection for inverted indexing. We adopted the *Small vs. Small* approach, which is considered efficient in many cases. Essentially, the intersection is computed by proceeding from smallest to largest list. This tends to produce smallest intermediate results, therefore to be the most efficient processing order.

4 Analysis of IncMine

In this section we prove a PAC-style guarantee on the quality of approximation of IncMine or, more precisely, on the transaction-sensitive variant that we have implemented. We believe the result is interesting because it explains theoretically some of the results we will observe in the experimental session, and also because PAC-analyses of frequent pattern mining algorithms are relatively rare so far.

We first state the probabilistic assumptions on the stream that generates the data stream. It is a formalization of the intuition behind the idea of progressive support central to IncMine: there is an underlying distribution that remains stable for reasonable stretches of time and generates the observed items; the sliding window of size W can then be viewed as a sample from that distribution.

Time $t = 1, 2, \ldots$ is discrete. At each time t, exactly one transaction is received from a distribution D_t on the set of all possible transactions. Samples at different times are independently drawn. Distributions D_t may evolve ("drift") over time. Obviously, if their evolution is arbitrarily complex there is no way to perform any mining, as we only sample one point from any one distribution. Informally speaking, one hopes that drastic changes from t to $t+1$ do not occur too often, or else distributions may change at every step, but only slightly.

The algorithm maintains a sliding of size W and the overall goal at time t is to provide an approximation of the set of FCIs in the window $t-W+1...t$. The algorithm partitions the stream in batches of size B, with $B \ll W$.

IncMine(σ,r) denotes the result of executing IncMine with minimum support parameter σ and parameter relaxation r on a given data stream of itemsets.

Theorem 1 *Fix a time t and assume that the distributions D_t have remained stable during the last T time steps (i.e. $D_{t-T} = \ldots = D_{t-1} = D_t$). Let O_t be the set of FCI output by IncMine(σ,r) at time t. Then, for every itemset X and every δ,*

1. *if* $\mathrm{rsupp}(X, D_t) \leq (1-\varepsilon)\sigma$ *then, with probability at least* $1-\delta$*, X is not in O_t.*
2. *if* $\mathrm{rsupp}(X, D_t) \geq (1+\varepsilon)\sigma$ *then, with probability at least* $1-\delta$*, X is in O_t.*

provided $\varepsilon \geq \sqrt{\frac{3}{\sigma W} \ln \frac{W}{\delta B}}$ *and* $r \leq 1 - \sqrt{\frac{2}{\sigma B} \ln \frac{W}{\delta B}}$.

Note that because IncMine may not have false positives, only false negatives, part (2) of the algorithm seems unnecessary. However, here the notion of false positive / negative is with respect to the generating distribution, not with respect to the actual stream of transactions observed by IncMine.

A qualitative interpretation of the bounds may be as follows: Fix window size W and batch size B. Clearly, itemsets whose probability is very close to σ may go either way (i.e., appear or not appear in the output). The bound on ε tells what "very close" means in this context. Observe that, as W grows, ε tends to 0, i.e., the uncertainty margin narrows for larger windows. The bound on r says what is the maximum r that one should use, mainly as a function of σB. Observe that if σB is large, r can be taken closer to 1; in any case σB should be somewhat larger than 1, as otherwise even σ-frequent itemsets will not reliably show up in batches of size B.

The bounds can be used in the reverse direction: given a desired σ and a desired value of the tolerance margin ε, determine what values of W, B, and r are appropriate.

The proof of the theorem is given in the Appendix. It uses crucially the Chernoff bounds on large deviations of sums of random variables, a standard tool in analysis of probabilistic algorithms. Interestingly, the proof indicates that the simple definition of the increasing sequence of supports $r(k)$ proposed in [3] is very generous in its definition of non-promising; this explains the very small false negative rate we will report in our experiments. In fact, the analysis suggests a better, perhaps optimal, sequence that will drop more non-promising itemsets earlier while at the same time maintaining the performance guarantees given by this theorem. Additionally, this new sequence does not require the user to enter or guess a parameter r, because it is deduced from the values of the other parameters.

5 Experiments

We evaluated our implementation with both synthetic and reality-based data streams. In this section we first explain how we generated these data streams. We then report the performance of IncMine under different types of input, i.e. streams with and without drift, compared with the MOMENT algorithm, which is still the standard for (exact) frequent itemset mining in data streams. We finally describe our experiments with a real-world dataset drawn from the MovieLens database.

Since IncMine is an approximate algorithm, we detail its accuracy in terms of two well-known accuracy measures, recall and precision. In our setting, *recall* is the fraction of true FCI's that do appear in the system's output, and *precision* is the fraction of itemsets in the output that truly are FCI. Thus, 1 minus the recall is also the *false negative rate*. We also provide an evaluation of the throughput (transactions processed per second) and of the amount of memory used.

At the time of testing, the only Java implementation of MOMENT we could find was M. Jarka's MOA-MOMENT, available as well as a MOA extension. But in our initial experiments with synthetic datasets, we found that this implementation often could not finish execution correctly because it quickly ran out of memory, and furthermore was orders of magnitude slower than our IncMine implementation. We decided to use the original C++ implementation provided by the authors of MOMENT [18]. C++ code is commonly accepted to be more efficient than the equivalent Java code (by variable and somewhat unpredictable factors), so this difference must be taken into account when discussing throughput.

5.1 Generating synthetic data streams

Since there is no standard synthetic stream generator adapted for frequent itemset patterns, usually researchers create (large) static transactions databases and provide them to the algorithm in a stream fashion. The most used synthetic data generator for itemset patterns is M. Zaki's *IBM Datagen* software [18]. Using standard notation, Datagen's synthetic datasets are named with the syntax `TxIyDz[Pu][Cv]`, where `x` is the average transaction length, `y` is the size of the set of items (in thousands), `z` is the number of generated transactions (in thousands), `u` is the average length of the maximal pattern, `v` is the correlation among patterns, and `[ ]` denotes optional parameters. For example, `T40I10D100K` names a dataset of 10^5 transactions, with an average of 40 items per transaction over a dictionary of 10^4 different items.

The initial testing phase was intended to measure the performances of our solution when the input contains no drift. We used the `T40I10D100K` dataset, a sparse dataset also provided by Zaki in [18] and used as test set in several previous papers. We analyzed our IncMine implementation in terms of precision and recall, and its throughput and memory usage, comparing it to MOMENT.

We used MOA Release 2012.03 [19] and worked with NetBeans 7.1.1 IDE (Build 201203012225), 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 running Windows 7. We set the Maximum heap size (`-Xms`) of the JVM to 1Gb for every IncMine execution. Unless otherwise stated, in the following experiments the *segment length B* of IncMine is fixed to 500 transactions, while its *window size W* is fixed to 10 segments. This corresponds to using a *window length* of 5000 transactions for MOMENT.

Accuracy. In a first experiment, we fixed the minimum support threshold to $\sigma = 0.1$ and varied the relaxation rate r in $[0.1, 1]$, thus evaluating the effect of the variation on the precision and recall (equivalently, false negative rate) of the algorithm. We recovered the set of FIs from the set of FCIs that are obtained by IncMine at every entire window slide and compared it to the real set of FIs computed with an implementation of the ECLAT algorithm available in [7]. The results, in terms of recall and precision averaged over all windows, are as follows for IncMine: Precision is always 1, as it should be in any false-negative algorithm. Recall is 1 up to $r = 0.6$, then decreases to 0.993, 0.949, 0.821, and 0.696 respectively for $r = 0.7$, 0.8, 0.9, 1, i.e. a reasonable degradation. Results for MOMENT are always 1 since it is an exact algorithm.

In a second experiment we measured accuracy as we varied the minimum support threshold σ, with r fixed to 0.5. Recall and prediction are essentially 1 for all values, even for a relaxation rate of 0.5. In a few cases precision is not exactly 1 as expected, but never goes below 0.994. Upon examination, these small discrepancies are due to a few itemsets placed exactly at the border between frequent and non-frequent itemsets (i.e., itemsets whose expected support is almost exactly $\sigma|S|$, hence empirically go "in the wrong side" because of random fluctuation when generating the dataset with given parameters). Hence, precision is 1 on the actual dataset, though not exactly 1 w.r.t. the underlying generating model.

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 and for different ranges of relaxation rate and minimum support threshold.

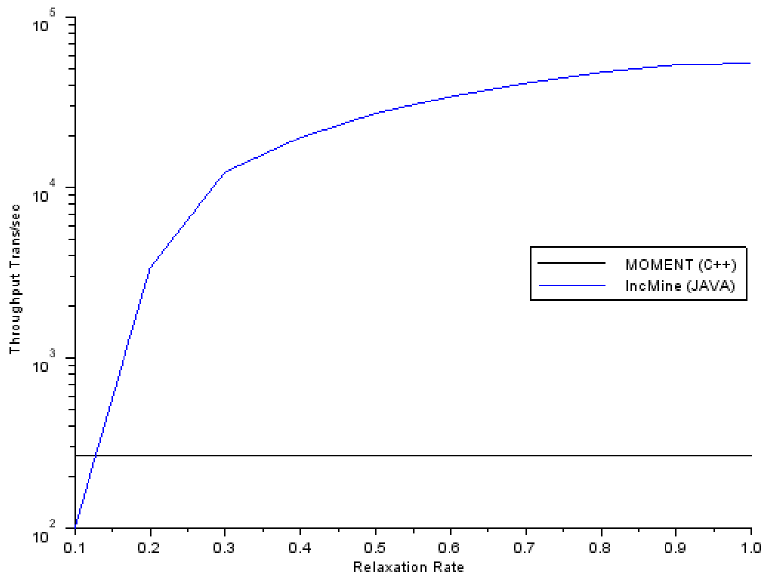

Figure 1: Throughput in trans/sec for different values of r ($\sigma = 0.1$). The minimum support used for MOMENT is equal to 500. Note the logarithmic scale in the y axis.

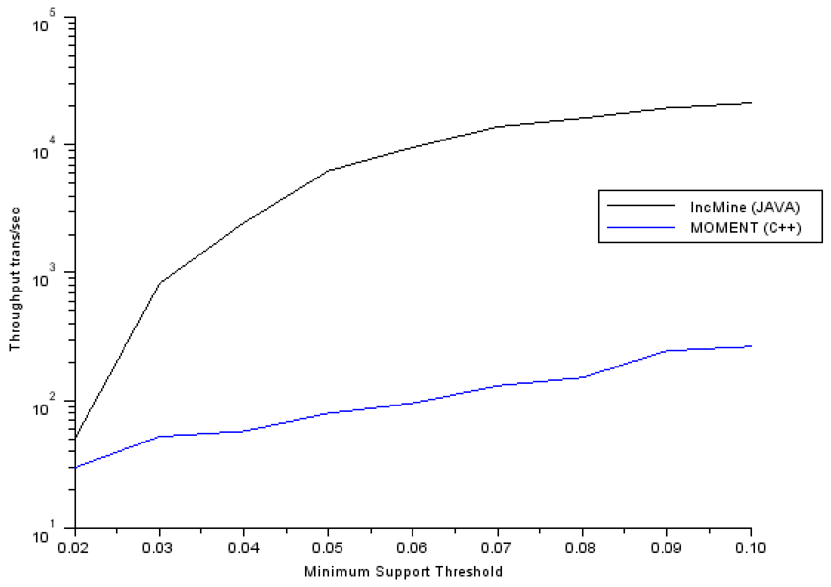

Figure 2: Throughput in trans/sec for different values of σ ($r = 0.5$). The minimum support used for MOMENT is equal to $\sigma \cdot 5000$. Note the logarithmic scale in the y axis.

Figure 1 reports the average throughput values for $r \in [0.1, 1]$, with a logarithmic scale in the y axis. Processing speed grows as the relaxation factor increases, since higher values of r imply a lower number of frequent closed itemsets mined in every segment. The figure also includes the result of executing MOMENT on the same data stream, with minimum support threshold $minsupp = \sigma \cdot |S| = 0.1 \cdot 5000 = 500$. Since MOMENT does not use a relaxation rate, its throughput is constant in this test. IncMine clearly outperforms MOMENT for every value of $r \geq 0.2$, and only for $r \simeq 0.1$ the performances of the two algorithms are comparable, that is, when forcing IncMine to be an almost exact algorithm. For example, for $r = 0.5$ the throughput of IncMine is more than two orders of magnitude larger that MOMENT's. At the same time, IncMine achieves almost perfect accuracies with this value of r , so we decided to adopt $r = 0.5$ for every future experiment.

Like before, we also study the behavior of the throughput with respect to the minimum support threshold σ. We fixed $r = 0.5$ and averaged the throughput obtained for $\sigma \in [0.02, 0.10]$. Figure 2 clearly shows that IncMine outperforms MOMENT in every case, and the difference between them grows as the minimum support threshold increases. Except below $\sigma = 0.02$, IncMine's throughput is at least one order of magnitude higher than MOMENT's.

IncMine's authors [3] performed similar tests comparing their C++ implementation of IncMine using CHARM author's code [16] and the same implementation of MOMENT we used here. Our results are qualitatively comparable to theirs (a quantitative comparison is impossible due to the differences in coding language and experimental platform), which we take as evidence for the correctness of our implementation.

5.1.1 Memory usage

Memory consumption is one of the key parameters in data stream algorithms, as it is often the limiting resource when the volume or complexity of the incoming data is large. For example, as mentioned already, we could not finish the experiments with the existing MOA-MOMENT package as it ran out of memory early in the execution. In fact, we decided to not compare IncMine with MOMENT in this case, because of the large 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; this factor does not exist for a C++ written program. We focused instead on analyzing the effect of the different parameters of IncMine on its memory consumption.

Note: For all experiments in this section, the results reported are the average of 10 independent executions, to smooth transient effects caused by dynamic memory allocation and collection.

First we analyze the memory consumption of IncMine as a function of the minimum support threshold σ, with fixed $r = 0.5$. In Table 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 increases as the minimum support threshold decreases, because a higher number of frequent closed itemsets have to mined and stored.

In Table 1 we also report the average size in memory of the main data structures specific to IncMine (the FCI-arrays and the Inverted FCI Index). Their size is one or two orders of magnitude smaller than the whole memory consumption of the algorithm or, in other words, the bulk of the memory is not really used by IncMine but by the batch miner it uses as a subroutine. This suggests that an important point of optimization for the algorithm could be reducing the memory used for the frequent closed itemset mining of each segment, possibly by a specialized algorithm. Observe, though, that it is the size of IncMine's structures the one that grows dangerously fast as as the support decreases; it seems to follow a law of the form $O(1/\sigma^{\alpha})$, for $\alpha \cong 2.5$.

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 3 we can see the behavior of the total memory consumption. For values lower than 60, memory consumption is almost constant, and for larger windows the average memory consumption increases linearly. This is due to the fact that the JVM reserves a certain amount of memory at the start of the execution. Instead, if we look at the size in memory of the FCI-arrays and Inverted FCI Index shown in Figure 4, we can see that there is a linear dependence between the number of segments retained in the window and the size in memory of such data structures. As before, it remains several times lower than the memory used by the batch segment miner, proving the memory efficiency of the data structures that have been used.

5.2 Introducing drift

We tested our implementation on datasets containing both sudden and gradual drift.

For sudden drift, *reaction time* can be crisply defined (less so in gradual drift) and is the measure on which we focused. The starting time of the concept drift

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

Table 1: Average memory consumption for varying 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).

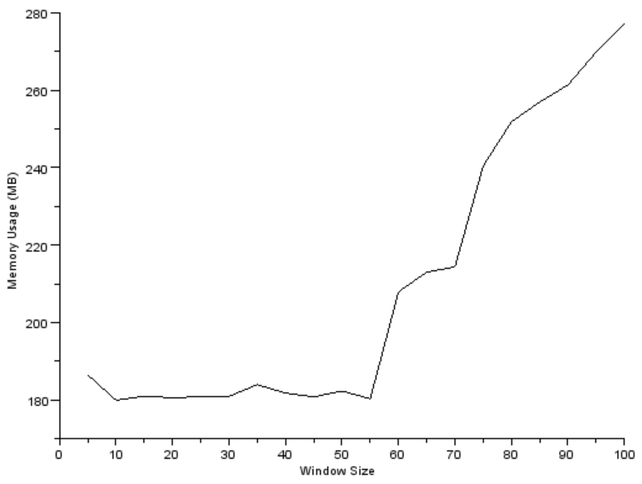

Figure 3: Average overall memory consumption for different window size values ($\sigma = 0.05$, $r = 0.5$).

can be defined exactly (i.e., looking at the transaction where we pass from one concept to the other in the synthetic dataset with drift); we can consider that a frequent itemset data stream algorithm 'reaches' a concept when its set of FCI is 'close' to the true set of FCI of this concept. To be precise, we decided by convention that a concept is reached when the size of the difference set is less than 5% of the number of true FCI for the new concept. We define the *reaction time* of the algorithm as time elapsed from the time the change occurs until the new concept is reached.

We created a new dataset by joining the two datasets `T40I10kD1MP6` and `T50I10kD1MP6C05`, passing from one to the other at transaction $8 \cdot 10^5$. Since the former has lower correlation between transactions than the latter, it has a higher density and more 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.

Reaction times are presented in Figure 5, as a function of window size. We find them remarkably small compared to typical results in evaluating reaction time in stream mining. Also, Figure 5 presents the evolution of the number of mined FCI's over time for different window sizes. Again one can see that longer windows imply larger reaction time, but an additional phenomenon can be observed: the plots for shorter windows are spiky, and become smoother as window size increases. This is due to the effect of random fluctuations which are of course more visible in shorter windows. In effect, window size controls a trade-off between stability and reaction time.

We also used datasets with gradual drift by smoothly merging the two datasets. Using MOA's "sigmoidal drift" capability for merging data streams, we could specify the duration and slope of the transition. In every case, the behavior was almost the same we noticed for abrupt changes, that is, longer windows correspond to longer reaction times, but provide more stable results.

5.3 Experiments with real data

Given the scarcity of accepted real benchmark streams with drift, and particularly for frequent pattern tasks, we transformed a real, but batch dataset as a basis. The *MovieLens* dataset, a free dataset provided by Group Lens Research [21], records, records user movie ratings. A rating is a value between $(1,\ldots,5)$ with half-point ratings, that a user provides after seeing it. The database contains about 10 million ratings applied to 10,681 movies by 71,567 users, from January 1996 to August 2007. MovieLens is intended for recommending systems research and evaluation, so 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 transactional, binary database to be used by our method. Importantly, the transactions generated were ordered by the timestamp of the corresponding rating, so in increasing chronological order. This introduced naturally some drift in the transaction database, as discussed presently.

We created a transaction database using each movie ID as an item, grouping ratings by timestamp with 5-minute granularity, then sorting by timestamp order. As a consequence, a "transaction" with timestamp T records the set of all movies that were rated together between times T and $T+5$ (in minutes), independently of the users that emitted the ratings. We imposed a maximum of 50 items for each transaction, and subdivided longer transactions into several different ones of the same length. This approximation becomes necessary to reduce the effect of a few very skewed transactions that appeared after grouping. This way, we obtained a data stream with 622,265 transactions and an average of 10.37 items per transaction. Transactions are certainly not uniformly distributed along this time interval. Considering that the number of different items is similar to what we used in the synthetic tests (about 10K), we used the knowledge we acquired there to guide the choice of execution parameters; we omit details.

One advantage of using the MovieLens database is that we can actually check whether the itemsets found make sense with regard to the external reality: Typically, a movie or group of movies will receive the highest number of ratings shortly after it is released. We verified that this seems to occur for major hits. For example, $\{$*Ocean's Eleven*, *Lord of the Rings: The Fellowship of the Ring*$\}$ is a frequent itemset in 2001, while $\{$*Spider-Man*, *Star Wars: Episode II - Attack of the Clones*$\}$ appears in 2002, and $\{$*Lord of the Rings: The Two Towers*, *Pirates of the the Caribbean: The Curse of the Black Pearl*$\}$ is frequent in 2003, coinciding with their release dates. A batch, non-streaming method will miss this fine temporal structure, even though it is often of highest interest in applications.

On the other hand, we can conjecture that drift is continuously occurring in the database, but unlike the synthetic case, we have not direct way of quantifying it. We propose, as a candidate empirical measure of drift, the number of itemsets that enter and leave the set of frequent itemsets per time unit, since we should expect no such changes when there is no drift. Indeed, we found high fluctuations of this measure in the MovieLens stream (corresponding perhaps to times with many releases) but it remained essentially zero over time for synthetic streams without drift.

6 Conclusions

We believe we have produced a solid, usable tool for frequent closed itemset mining on streaming scenarios that may help bringing this technology to actual industrial usage. At the same time, our implementation can be used as a reference or baseline for evaluation of further research in the area.

Potential extensions of our work include building self-tuning algorithms that choose their parameters (semi)automatically; the definition of both synthetic and real, truly streaming, benchmarks; and possibly trying other base (batch) miners besides CHARM, optimized for this purpose, that may reduce memory consumption. An important question, but to our knowledge not yet addressed, is the possibility of parallelizing this or another method for closed itemset mining on streams, in order to increase the throughput. As already mentioned, MOA does not currently support parallel or distributed processing.

Acknowledgments

M. Quadrana and R. Gavaldà's work was partially supported by MICINN project TIN2011-27479-C04-03 and by the SGR2009-1428 project (LARCA). A preliminary version of this work was presented at the CCIA 2013 conference. We are grateful to the CCIA 2013 reviewers for helpful comments.

References

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

[2] R. Agrawal, R.Srikant. Fast Algorithms for Mining Association Rules in Large Databases VLDB Conference, 1994.

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

[4] J. Cheng, Y. Ke, W. Ng. A survey on algorithms for mining frequent itemsets over data streams. Knowl. Inf. Syst. 16(1): 1-27 (2008).

[5] 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).

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

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

[8] 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).

[9] M.J. Kearns, U. Vazirani. An introduction to computational learning theory. The MIT Press, 1994.

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

[11] J. Pei, J. Han, R. Mao. CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets. ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, 2000.

[12] 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.

[13] J. Wang, J. Han, J. Pei. CLOSET+: Searching for the best strategies for mining frequent closed itemsets. KDD conference, 2003.

[14] 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.

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

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

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

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

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

[20] A Frequent Itemset Mining Implementations Repository. Maintained by B. Goethals. Last accessed: July 5th, 2013. `http://fimi.ua.ac.be/`

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

Appendix: Proof of Theorem 1

We will use the following well known bounds on the tails of sums of independent random variables; see e.g. [9].

Lemma 1 (Chernoff bounds) *Let* $X_1, \ldots, X_n$ *be independent 0/1-valued random variables with* $\Pr[X_i = 1] = \mu$. *Let* S *be the random variable* $(\sum_{i=1}^{n} X_i)/n$. *Then for every* $\varepsilon < 1$ *we have:*

1. $\Pr[S \geq (1-\varepsilon)\mu] \leq \exp\left(-\frac{1}{2}\varepsilon^2 \mu n\right)$, *and*

2. $\Pr[S \geq (1+\varepsilon)\mu] \leq \exp\left(-\frac{1}{3}\varepsilon^2 \mu n\right)$.

Fix itemset X, time t, a window size W and a batch size B. Let B_1, B_2, $\ldots B_{W/B}$ denote the most recent transaction batches (B_1 is oldest and $B_{W/B}$ is most recent). We use $B_{a..b}$ to denote the union of B_a through B_b. Recall that O_t is the set of FCI output at time t, and also that IncMine maintains an internal set L_t of semi-FCI with the invariant that $X \in L_t$ if and only if, for all $k \in 1..W$, $rsupp(X, B_{1..k}) \geq r(k)\sigma$. Observe that for a set to be in O_t it must 1) enter $L_{t'}$ at some $t' \leq t$, 2) not be dropped between t' and t (remain in the L set), and 3) have relative support at least σ in W. Let D denote D_t $(= D_{t-1} = \ldots D_{t-W+1})$; we omit X from the *rsupp* function as only one X is considered.

To prove (1), suppose that $rsupp(D) \leq (1-\varepsilon)\sigma$. Because IncMine has no false positives, if X is in O_t then $rsupp(B_{1..W/B}) \geq \sigma$, that is, it has empirical support at least σW in the window of the last W elements. Therefore, we have to bound the probability that X has empirical support above σW although its expected support according to D is at most $(1-\varepsilon)\sigma W$. This is

$$\begin{aligned}
\Pr[X \in O_t] &\leq \Pr[rsupp(B_{1..W/B}) \geq \sigma W] \\
&\leq \Pr[rsupp(B_{1..W/B}) \geq (1+\varepsilon)(1-\varepsilon)\sigma W \\
&= \exp\left(-\frac{1}{3}\varepsilon^2 \sigma W\right).
\end{aligned}$$

where we have used that $1 \geq (1+\varepsilon)(1-\varepsilon)$ and the Chernoff bound. This is less than δ if and only if

$$\varepsilon \geq \sqrt{\frac{3}{\sigma W}\ln\frac{1}{\delta}}.$$

which is implied by the bound on ε given in the theorem.

Proving (2) is more complex as it involves the rule for dropping non-promising itemsets. For this proof we redefine slightly the meaning of $r(k)$: rather than the cutpoint at time step $t-k$, it is the cutpoint at the border of the k-th batch B_k; its value is therefore $r(k) = (1-r)/W \cdot (B \cdot k)) + r$.

Suppose that $rsupp(D) \geq (1+\varepsilon)\sigma$. We claim that if X is not in O_t then for some $k \in \{1..W\}$ we have $rsupp(B_{1..k}) \geq r(k)\sigma$. This may be (as discussed above) because it never entered the L set in the last W time units, or because it did but it was dropped later, or because it was not dropped but reached the end of W but did not have the required support σ. All three cases are considered in the above given the range of k. Therefore, we bound:

$$\begin{aligned}
&\Pr[X \notin O_t] \\
&\leq \Pr[\exists k \in \{1..W/B\} : rsupp(B_{1..k}) \leq r(k)\sigma] \\
&\leq \sum_{k \in \{1..W/B\}} \Pr[rsupp(B_{1..k}) \leq r(k)\sigma] \\
&\leq \frac{W}{B} \cdot \max_{k \in \{1..W/B\}} \Pr[rsupp(B_{1..k}) \leq \frac{r(k)}{1+\varepsilon}(1+\varepsilon)\sigma] \\
&\leq \frac{W}{B} \cdot \max_{k \in \{1..W/B\}} \exp\left(-\frac{1}{2}\left(1-\frac{r(k)}{1+\varepsilon}\right)^2 (1+\varepsilon)\,\sigma B k\right).
\end{aligned}$$

Define $f(k) = \left(1-\frac{r(k)}{1+\varepsilon}\right)^2 (1+\varepsilon)\,\sigma B k$. To determine the max in this inequality, we need to choose the k that minimizes $f(k)$.

The derivative of f has a single zero in the range of k, which is a maximum. Therefore $f(k)$ achieves its minimum at one of the two endpoints of the range, that is, either $k = 1$ or $k = W/B$. We thus need to verify that

$$\frac{W}{B} \cdot \exp\left(-\frac{1}{2}(1 - r(1)/(1+\varepsilon))^2(1+\varepsilon)\sigma B \cdot 1\right) \leq \delta$$

and

$$\frac{W}{B} \cdot \exp\left(-\frac{1}{2}(1 - r(W/B)/(1+\varepsilon))^2(1+\varepsilon)\sigma B \cdot (W/B)\right) \leq \delta.$$

For the first inequality, use that $r(1) = r$, and the inquality is true if

$$\frac{1}{2}(1 - r/(1+\varepsilon))^2(1+\varepsilon)\sigma B \geq \ln(W/\delta B)$$

which is certainly true if

$$(1-r)^2 \geq \frac{2}{\sigma B}\ln(W/\delta B)$$

which is the bound on r given in the theorem.

For the second inequality, use that $r(W/B) = 1$, and the inequality is true if

$$\frac{1}{2}(1 - 1/(1+\varepsilon))^2(1+\varepsilon)\sigma W \geq \ln(W/\delta B). \qquad (1)$$

Given that $(1 - 1/(1+\varepsilon))^2(1+\varepsilon) \leq \varepsilon^2$ for all $\varepsilon > 0$, the inequality trivially follows from the bound given for ε by the theorem. This completes the proof.

It can be seen from the analysis that that the increasing sequence $r(k)$ chosen in the original IncMine paper is not optimal. The main loss occurs when we loosely bound

$$\Pr[\exists k : rsupp(B_{1..k}) \leq r(k)\sigma] \leq \frac{W}{B} \cdot \max_k \Pr[rsupp(B_{1..k}) \leq r(k)\sigma]$$

A better, perhaps optimal, sequence of $r(k)$ would keep the second probability constant over k so that the bound above is tight. More precisely:

- Require $r(W/B) = 1$, which forces the value of ε by Equation (1).
- For any $k < W/B$, let $r(k)$ be defined by

$$(1 - r(k)/(1+\varepsilon))^2 \cdot k = (1 - r(W/B)/(1+\varepsilon))^2 \cdot (W/B).$$

Note that the sequence does not depend on r, which therefore becomes unnecessary to the algorithm.

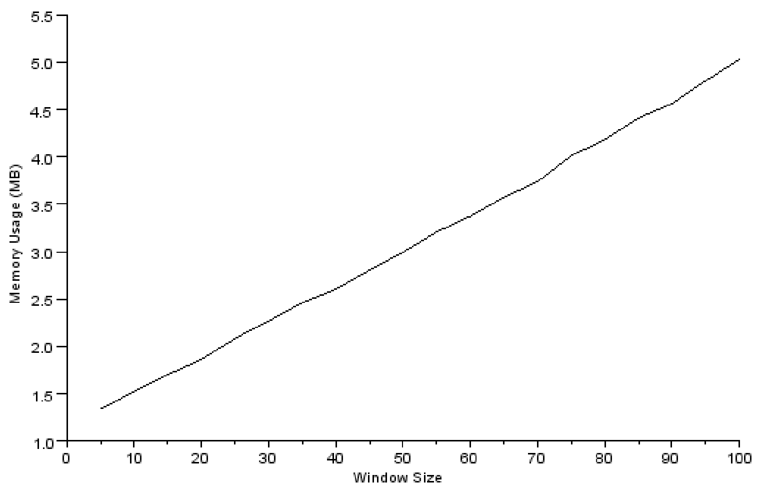

Figure 4: Average memory consumption of IncMine's data structures for different window size values ($\sigma = 0.05$, $r = 0.5$).

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

Figure 5: Reaction time for *window_size* $\in [10, 100]$ (in number of segments of size 500).

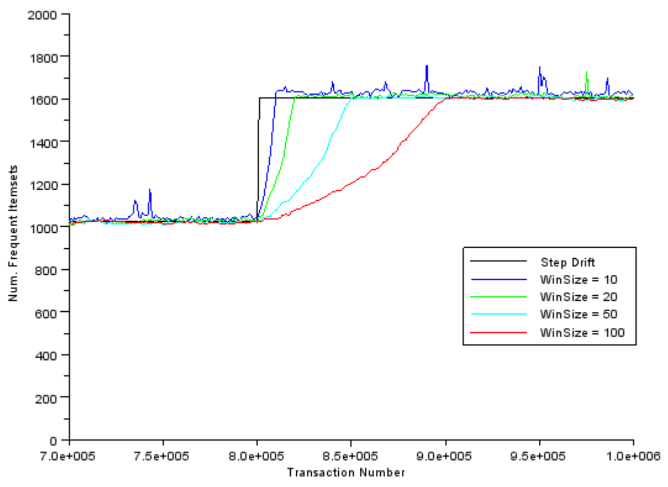

Figure 6: Number of extracted FCIs over time for window size $\in \{10, 20, 50, 100\}$ (in number of segments of size 500).