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

Massimo Quadrana [1], Albert Bifet [2], Ricard Gavaldà [3]

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.

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 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 [18] for batch itemset mining.

After thorough examination of several of the algorithms in the literature, for reasons to be explained we decided to implement the IncMine algorithm of Cheng *et al.* [2]. 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 [5] of the

[1] U. Politècnica de Catalunya (BarcelonaTech) and Politecnico di Milano, e-mail: max.square@gmail.com
[2] Yahoo! Research, e-mail: abifet@yahoo-inc.com
[3] U. Politècnica de Catalunya (BarcelonaTech), e-mail: gavalda@lsi.upc.edu

CHARM algorithm [14,15]. 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.

Massive Online Analysis framework [1,17] is a data stream mining framework developed by the U. Waikato. Implementing on top of MOA ensured portability and maintainability, as well as not having to implement from scratch the basic stream processing primitives. Let us note that there is already a MOA extension, due to M. Jarka, implementing the MOMENT method [3] for frequent closed itemset mining. However, it is reported in [2] 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 over the processing, and particularly how it 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.

Some discussions and results omitted in this version can be found in the technical report [9]. The software described here is available from the MOA project site [17] as a MOA extension since September 2012.

2. Background: The Frequent Itemset Mining Problem

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.

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*. An itemset of size k is called a k-itemset. We denote by $\mathscr{I}_k$ the set of all k-itemsets, i.e, subsets of $\mathscr{I}$ of size k. 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, $\boldsymbol{t}$(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 $minsupp(X, \mathscr{D})$, is the number of transactions in $\mathscr{D}$ that contain X, or $minsupp(X, \mathscr{D}) = |\boldsymbol{t}(X)|$. Fix some user-defined minimum support threshold *minsupp*. Then X is said to be *frequent* in $\mathscr{D}$ if $minsupp(X, \mathscr{D}) \geq minsupp$. When there is no confusion about D and *minsupp* we will drop then and simply say "X is frequent" and write its support as $minsupp(X)$.

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, ECLAT [13], and FP-GROWTH [6] 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 re-

dundant, 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 (with the supports) can be computed from the first (with the supports). To be precise, an itemset is frequent if and only if it is a subset of some frequent closed itemset. This is also true if we omit the "frequent" condition everywhere. Consequently, algorithms that obtain the frequent closed sets directly, without internally actually 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 [8], CHARM [14], and CLOSET+ [11].

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, $minsupp(X,\mathscr{D})$ to this scenario in the natural way, e.g. $minsupp(X,W)$.

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.

2.1. 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 [3], 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 [6], to store the information of *all* transactions in the sliding window, with no pruning of infrequent itemsets.

CLOSTREAM, proposed by Yen *et al.* in [12], maintains the *complete set* of closed itemsets over a *transaction-sensitive* sliding window *without any support information*. It uses an *update per transaction* policy. 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.* [7], 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 [2], proposes the *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*.

CLAIM was proposed by Song *et al.* in [10] 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*.

Let us then 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 inexact 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.

3. IncMine and our Implementation

We decided to implement on top of the MOA framework for a number of reasons, including: 1) it is the most complete public framework for stream mining, with a fast-growing

user base. 2) 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. 3) 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 is assumed: any kind of itemset stream that can be passed to MOA via its API should work correctly. 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.

IncMine [2] is an algorithm for incremental update of *frequent closed itemsets* (*FCIs*) over a high-speed data stream. pseudocode and more detail can be found in [9] as well as the original paper. We departed from, or completed, the description in [2] in the following three points:

1. Window type. We decided to implement a *transaction-sensitive* window instead of the *time-sensitive* window proposed in [2], 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 [14], which is available as part of the Sequential Frequent Pattern Mining framework [5], 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. 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 [2]. Luckily, the problem has been extensively used, for example in the Information Retrieval field. Culpepper *et al.* in [4] provide a survey of algorithms for efficient multiple set intersection for inverted indexing. We adopted the *Small Versus Small* approach, which is considered efficient in many cases. Essentially, the intersection is computed by proceeding from smallest to largest set. This tends to produce smallest intermediate results, therefore to be the most efficient processing order.

4. Experiments

We evaluated our implementation with both synthetic and reality-based data streams. In this section we first explain how we generate these data streams. We then report the performance of IncMine under different types of input, i.e. streams with and without drift, 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 achieved throughput; memory usage discussion can be found in [9].

At the time of testing the only Java implementation of MOMENT we could find was the one already running on top of MOA, due to M. Jarka. 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 MOMENT's author. 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.

4.1. 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 Zaki's *IBM Datagen* software [16]. 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, `z` is the number of genarated transactions (in thousands), `u` is the average length of the maximal pattern, `v` is the correlation among patterns, and `[ ]` denotes optional parameters.

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 [16], which is 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 algorithm.

We used MOA Release 2012.03 [17] 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* of IncMine is fixed to 500 transactions, while its *window size* 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 $[0.1, 1]$, in order to evaluate its influence on the false negative rate. Thus we evaluate the effects of the variation of the relaxation rate on the *precision* and recall 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 [5]. The results, in terms of recall and precision averaged over all windows, are as follows for IncMine: Precision is always 1, as any false-negative algorithm should have. 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.

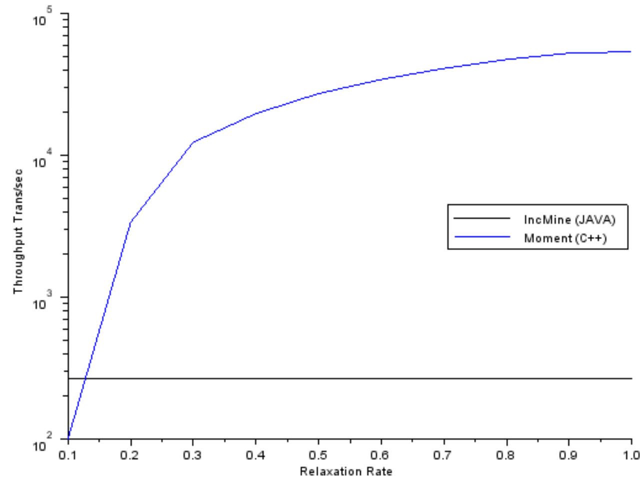

Figure 1. Throughput in trans/sec for different values of r ($\sigma = 0.1$). The minimum support used for MOMENT is equal to 500. Observe the logarithmic scale in the y axis.

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).

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 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 very good 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 we average the throughput obtained

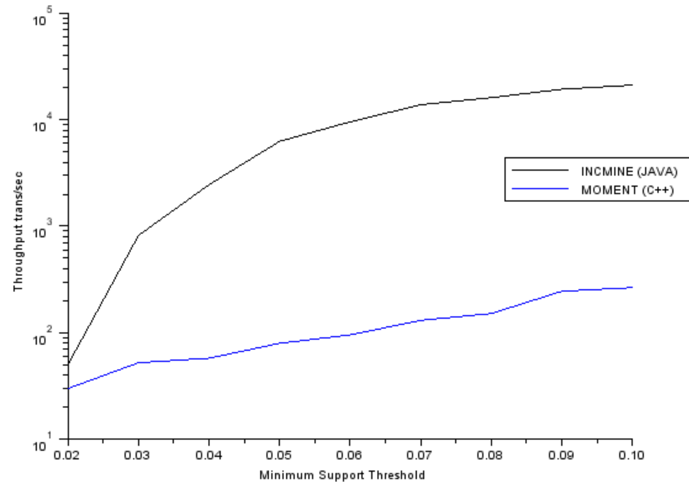

Figure 2. Throughput in trans/sec for different values of σ ($r = 0.5$). The minimum support used for MOMENT is equal to $\sigma \cdot 5000$. Observe the logarithmic scale in the y axis.

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 [2] performed similar tests comparing their C++ implementation of IncMine using CHARM author's code [14] 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.

4.2. 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 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 to true set of FCI of this concept. Since the number of FIs varies every transaction, we decide by convention that a concept is reached when the size of the difference set is lower that 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 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

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 3. Reaction time for $window_size \in [10, 100]$.

correctly the quality of the reaction to every kind of concept drift. Reaction times are presented in Figure 3, as a function of window size. We find them remarkably small compared to typical results in evaluating reaction time in stream mining.

We also used datasets with gradual drift by smoothly merging several datasets. Using MOA's "sigmoidal drift" 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 now, depending on the nature of the stream, drift can last from some hundreds of transactions to several thousands, and this amplifies the differences between small and longer windows. We noted additionally the expected fact that smaller windows, though faster to detect drift, are less stable in the sense that they often exhibit spikes due to random fluctuations in the data. Longer windows react more slowly, but provide more stable processing. Quantitative results can be found in [9].

4.3. 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 [19], 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. For the latter, see [9] for the details of the transformation to a transactional, binary database. 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.

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. Detailed results of the evaluation on this dataset can be found in [9].

5. 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 possibilty of parallelizing this or another method for closed itemst mining on streams, in order to increase the throughput. As already mentioned, currently MOA does not support parallel or distributed processing.

References

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

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

[3] 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).

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

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

[6] 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).

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

[8] 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.

[9] M. Quadrana, A. Bifet, R. Gavaldà. An Efficient Closed Frequent Itemset Miner for the MOA Stream Mining System (extended version). Technical Report LSI-13-9-R, Departament LSI, U. Politècnica de Catalunya, `http://www.lsi.upc.edu/dept/techreps/techreps.html`.

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

[12] 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.

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

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

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

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

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

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

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