MFI-TransSW+: Efficiently Mining Frequent Itemsets in Clickstreams

Franklin A. de Amorim[1], Bernardo Pereira Nunes[1,3]
Giseli Rabello Lopes[2], and Marco A. Casanova[1]

[1] Department of Informatics - PUC-Rio, Rio de Janeiro, RJ, Brazil
{famorim,bnunes,casanova}@inf.puc-rio.br
[2] Department of Computer Science - UFRJ, Rio de Janeiro, RJ, Brazil
giseli@dcc.ufrj.br
[3] Department of Applied Informatics - UNIRIO, Rio de Janeiro, RJ, Brazil
bernardo.nunes@uniriotec.br

Abstract. Data stream mining is the process of extracting knowledge from massive real-time sequence of data items arriving at a very high data rate. It has several practical applications, such as user behavior analysis, software testing and market research. However, the large amount of data generated may offer challenges to process and analyze data at nearly real time. In this paper, we first present the MFI-TransSW+ algorithm, an optimized version of MFI-TransSW algorithm that efficiently processes clickstreams, that is, data streams where the data items are the pages of a Web site. Then, we outline the implementation of a news articles recommendation system, called ClickRec, to demonstrate the efficiency and applicability of the proposed algorithm. Finally, we describe experiments, conducted with real world data, which show that MFI-TransSW+ outperforms the original algorithm, being up to two orders of magnitude faster when processing clickstreams.

Keywords: Datastream; frequent itemsets; data mining.

1 Introduction

Data stream mining is the process of extracting knowledge from massive real-time sequence of possibly unbounded and typically non-stationary data items arriving at a very high data rate [2]. The mining of frequent itemsets in data streams has several practical applications, such as user behavior analysis, software testing and market research. Nevertheless, the massive amount of data generated may pose an obstacle to processing them in real time and, hence, to their analysis and decision making. Thus, improvements in the efficiency of the algorithms used for these purposes may bring substantial benefits to the systems that depend on them.

The problem of mining frequent itemsets in data streams can be defined as follows. Let $I = \{x_1, x_2, ..., x_n\}$ be a set of items. A subset X of I is called an itemset. A transactional data stream is a sequence of transactions, $D = (T_1, T_2, ..., T_N)$, where a transaction T_i is a set of items and N is the total number of transactions in D. The number of transactions in D that contains X is called the support of X and denoted $sup(X)$. An itemset

X is frequent iff $sup(X) \geq N \cdot s$, where $s \in [0, 1]$ is a threshold, defined by the user, called the minimum support. The challenge of mining frequent itemsets in data streams is directly associated with the combinatorial explosion of the number of itemsets and the maximum memory space needed to mine them. Indeed, if the set of items I has n elements, the number of possible itemsets is equal to $2^n - 1$. As the length of a data stream approaches a very large number, the probability of items to be frequent becomes larger and harder to control with limited memory.

A *clickstream* over a Web site is a special type of data stream where the transactions are the users that access the Web site and the items are the pages of the Web site.

MFI-TransSW [8] is an efficient algorithm, in terms of processing and memory consumption, for finding frequent itemsets in sliding windows over data streams, using bit vectors and bit operations. However, the lack of a more efficient sliding window step and the fact that MFI-TransSW algorithm was designed to process a finite sequence of transactions opens up new opportunities for improvement. In this paper, we introduce the MFI-TransSW+ algorithm, an optimized version of the MFI-TransSW to process clickstreams in real time. Additionally, we briefly describe ClickRec, an implementation of a news articles recommendation system based on MFI-TransSW+. Finally, we present experiments, using real data, to evaluate the performance for clickstreams of the original MFI-TransSW and the proposed MFI-TransSW+ algorithm, as well as to validate the recommendations obtained with ClickRec.

The key contributions of the paper are twofold: a new strategy to maintain bit vectors, implemented in the MFI-TransSW+ algorithm, which leads to a substantial performance improvement when processing real-world clickstreams; a new recommendation strategy for news articles, based on measuring metadata similarity, which proved to double the recommendation conversion rate, when compared with the baseline recommendation algorithm adopted by a news portal.

The remainder of the paper is organized as follows. Section 2 reviews the literature. Section 3 presents the MFI-TransSW and the MFI-TransSW+ algorithms. Section 4 overviews the ClickRec recommender system and its implementation. Section 5 describes the experiments. Finally, Section 6 concludes the paper.

2 Related Work

The algorithms that mine frequent itemsets may be classified into those that adopt fixed windows and those that use sliding windows [4]. They may also be classified according to the results they produce into exact and approximate algorithms. The approximate algorithms may be further sub-divided into false positives and false negatives.

Agrawal et al. [1] proposed the A-Priori algorithm, one of the most important contributions to the problem of finding frequent itemsets. The A-Priori algorithm is based on the observation that, if a set of items i is frequent, then all subsets of i are also frequent.

Manku and Motwani [11] developed two single-pass algorithms, called Sticky-Sampling and Lossy Counting, to mine frequent itemsets in fixed windows. They also developed a method, called Buffer-Trie-SetGen (BTS), to mine frequent itemsets in data streams. Both algorithms have low memory usage, but they generate approximate results, with false positives.

Li et al. [9] proposed a single-pass algorithm, called DSM-FI, and Li et al. [10] defined the DSM-MFI algorithm to extract all frequent itemsets (FI) and maximal frequent itemsets (MFI) from a data stream. These algorithms use a data structure based on prefix-trees to store the items as well as auxiliary data.

Yu et al. [13] proposed false positive and false negative algorithms based on the Chernoff Bound [5] to mine frequent itemsets from high speed data streams. The algorithms use two parameters to eliminate itemsets and control memory usage.

Lee et al. [7] proposed a filtering algorithm to incrementally mine frequent itemsets in a sliding window. Chang et al. [3] proposed an algorithm based on BTS, called SWFI-stream, to find frequent itemsets in a transaction-sensitive sliding window. Chi et al. [6] proposed an algorithm, called Moment, to process data streams and extract closed frequent itemsets inside a transaction-sensitive sliding window. Li et al. [8] developed the MFI-TransSW algorithm to mine frequent itemsets in sliding windows over data streams, using bit vectors and bit operations.

The MFI-TransSW+ algorithm, proposed in this paper, processes clickstreams in sliding windows. It adopts a version of the A-Priori algorithm to find all frequent itemsets, without generating false positives or false negatives. Lastly, it maintains a list of users of the current window and circularly updates the bit vectors when the window moves, which result in expressive performance gains, when processing clickstreams, as discussed in Section 5.

3 The MFI-TransSW and the MFI-TransSW+ Algorithms

3.1 The MFI-TransSW Algorithm

The MFI-TransSW algorithm mines frequent itemsets in data streams. MFI-TransSW is a *single pass mining* algorithm in the sense that it reads the data only once and uses bit vectors to process the data that fall inside a sliding window and to extract the frequent itemsets, without generating false positives or false negatives.

Each item X that occurs in a transaction in the sliding window is represented as a bit vector Bit_X, where each position $Bit_X(i)$ represents the ith active transaction in the window and is set to 1 if the transaction includes X. For example, suppose that the window has size $W = 3$ and that the 3 active transactions are $T_1 = \{a, b\}$, $T_2 = \{b\}$ and $T_3 = \{a, b, c\}$. Then, the bit vectors corresponding to the items in the window W would be $Bit_a = 101$, $Bit_b = 111$ e $Bit_c = 001$. MFI-TransSW has three main stages, described in what follows: (1) *window initialization*; (2) *window sliding*; and (3) *frequent itemsets generation*.

The first stage processes the transactions in the order they arrive. For each transaction T and each item X in T, the initialization stage creates a bit vector Bit_X for X, if it does not exist, initialized with 0's. The bit corresponding to T in Bit_X is set to 1.

When the number of transactions reaches the size of the window, the window becomes *full* and the second stage starts. The sequence of transactions is treated as a queue, in the sense that the first and oldest transaction is discarded, opening space for the new transaction, which is added at the end of the queue. Let O be the oldest transaction and T be th new transaction. The bit vectors are updated accordingly: a *bitwise*

left shift operation is applied to each bit vector, which eliminates the first bit, that is, the information about the oldest transaction O. The information about the new transaction T is registered at the end of the bit vectors: for each bit vector Bit_X, the last bit of Bit_X is set to 1, if T contains X, and set to 0, otherwise. If T contains an item for which no bit vector exists, a new bit vector is created as in the initialization stage. At this point, a cleaning operation eliminates all bit vectors that contain only 0's. This process continues as long as new transactions arrive or the stream is interrupted.

The third stage is an adaptations of the A-Priori algorithm [1] using bit vectors and may be executed at any time at the user request or whenever it becomes necessary to generate the frequent itemsets. This stage successively computes the set FI_k of frequent itemsets of size k. FI_1 contains all items X such that $sup(X)$, the *support* of X, defined as the number of bits set to 1 in Bit_X, is greater than a minimum threshold. CI_2 contains all pairs X, Y of frequent items in FI_1 and FI_2 contains all pairs X, Y in CI_2 such that $Bit_{XY} = Bit_X \wedge Bit_Y$ has at least as many bits set to 1 as the minimum threshold. This process continues to generate CI_k and FI_k until no more frequent itemsets are found.

3.2 The MFI-TransSW+ Algorithm

In this section we describe the version of MFI-TransSW we propose, called MFI-TransSW+, which is optimized to process clickstreams. Our experiments indicated that MFI-TransSW+ is 2 orders of magnitude fasters than MFI-TransSW when processing clickstreams (see Section 5).

Processing Clickstreams. A *clickstream* over a Web site is a special type of data stream, where the transactions are the users that access the Web site and the items are the pages of the Web site [12]. The records of a clickstream usually have the fields: (1) *Timestamp*, the time the click was executed; (2) *User*, the user ID that executed the click; (3) *Page*, the page accessed; and (4) *Action*, the action executed by the user, such as a page access, vote, etc. (since we consider only page accesses, we will omit this field).

Recall that, when the window moves and a new transaction arrives, the MFI-TransSW algorithm applies the shift left operation to all bit vectors, including the bit vectors that do not correspond to the items of the new transaction. However, in a clickstream, the set of pages a user accesses is typically a small fraction of the pages all users access in the window (see Section 5). Hence, the way MFI-TransSW updates bit vectors becomes inefficient. MFI-TransSW+ therefore adopts new data structures to optimize processing the bit vectors, as discussed in what follows.

From now on we refer to users, pages and pagesets, rather than transactions, items and itemsets, respectively.

Data Structures. We assume that users have a unique ID (UID) and that the sliding window has size w. Then, all bit vectors will also have size w. We introduce two new data structures. The *list of UIDs*, $LUID$, is a circular list, with maximum size w, that contains the $UID's$ of all users that occur in the current window, in the order they arrive. The position of each user in $LUID$ indicates its position in the bit vectors. The *list of*

bit vectors per user, $LBVPU$, stores, for each user u, the IDs of the bit vectors that have the bit corresponding to u set to 1.

These data structures are updated as follows. Suppose that the next record of the clickstream indicates that user u accessed page p.

Case 1. User u is in $LUID$, that is, u has already been observed in the current window.

Case 1.1. Page p has been accessed before.
Then, there is a bit vector Bit_p already allocated to page p. We set $Bit_p(j) = 1$, where j is the position of u in $LUID$, and add the ID of Bit_p to $LBVPU$, associated with user u, if not already in $LBVPU$.

Case 1.2. Page p has never been accessed before.
We allocate a bit vector Bit_p for page p, initialize Bit_p with 0's, set $Bit_p(j) = 1$, where j again is the position of u in $LUID$, and add the ID of Bit_p to $LBVPU$, associated with user u.

Case 2. User u is not in $LUID$, that is, u has not been observed in the current window.

Case 2.1. Assume that the current window is not full.
We add u to the end of $LUID$ and update the data structures as in Cases 1.1 and 1.2.

Case 2.2. Assume that the current window is full.
We do not apply a shift left operation to all bit vectors, but update each bit vector and $LUID$ as circular lists with the help of a pointer k to the last position used. That is, the next position to be used is $((k + 1) \bmod w)$. We first set to 0 position k of the bit vectors associated, in the $LBVPU$ list, with the old user at position k of $LUID$ and remove the old user and all its bit vector IDs from $LBVPU$. The new user u is added to $LUID$ in position k, replacing the user that occupied such position. Next, we update the data structures as in Cases 1.1 and 1.2 for the new user at position k of $LUID$. Finally, we eliminate all bit vectors that contain only 0's.

Pseudo-code of the MFI-TransSW+ Algorithm. Algorithm 1 outlines the pseudo-code of MFI-TransSW+.

Lines 1 to 12 just repeat how the data structures are updated. Lines 1 to 3 initialize the data structures. Lines 4 to 12 process the next record of the clickstream. Line 4 reads the next record from the clickstream CDS, setting user u and page p. Line 4 verifies if user u already exists in $LUID$. If this is the case, Line 6 updates the data structures as in Case 1. Lines 8 to 12 are executed if user u does not exist in $LUID$. Line 8 verifies if $LUID$ is not full. If $LUID$ is not full, Line 9 updates the data structures as in Case 2.1. If $LUID$ is full, Lines 11 and 12 are executed. Line 11 moves pointer k to the next position to be updated. Line 12 updates the data structures as in Case 2.2.

Lines 13 to 21 generate frequent pagesets for the current window as in the original algorithm. Line 13 generates all frequent pagesets of size $j = 1$; it counts the number of bits 1 of the bit vectors and stores in FI_1 those that have more bits set to 1 than the minimum support $(s \cdot w)$ for the current window. Lines 14 to 21 are executed for each possible size j of pagesets. Line 14 initializes the loop with $j = 2$ and increments j by 1 until no new frequent pagesets is found. Line 15 generates CI_j from the list of frequent pagesets of size $j - 1$, as in the original algorithm. To compute the support of the new

candidates, Line 16 *ANDs* the bit vectors of the pages in each pageset in CI_j. Lines 17 to 21 identify which candidates are frequent pagesets. Line 17 initializes FI_j, the list of frequent pagesets of size j. Line 18 selects a candidate c_j from CI_j. Line 19 verifies if the support of c_j is greater than or equal to $(s \cdot w)$, the minimum support. If this is the case, Line 20 adds c_j to FI_j. Lastly, Line 21 adds all pagesets in FI_j to the set *FI-Output* of all frequent pagesets.

Algorithm 1: The optimized MFI-TransSW+ algorithm.

```
input : CDS - a clickstream
        s - a user-defined minimum support, ranging in the interval [0, 1]
        w - a user-defined window size
output: FI-Output - a set of frequent page-access-sets
   /* Initialize the data structures.                                    */
1  k ← 0;
2  FI-Output ← ∅;
3  LUID, LBVPU ← NULL;
   /* Process the next record of the clickstream.                        */
4  for (u, p) ∈ CDS do
5      if u ∈ LUID then
6          Update the data structures for (u, p) as in Case 1;
7      else
8          if LUID ≠ FULL then
9              Update the data structures for (u, p) as in Case 2.1;
10         else
11             k ← (k + 1) mod w ;
12             Update the data structures for (u, p) as in Case 2.2;
   /* Generate frequent pagesets, at user's request.                     */
13 FI_1 ← { frequent pagesets of size 1 };
14 for (j←2; FI_{j−1} ≠ NULL; j++) do
15     Generate CI_j from FI_{j−1} ;
16     Execute bitwise AND to compute the support of all pagesets in CI_j;
17     FI_j ← ∅;
18     for c_j ∈ CI_j do
19         if |sup(c_j)| ≥ w · s then
20             FI_j ← FI_j ⋃{c_j};
21     FI-Output ← FI-Output ⋃ FI_j;
```

4 ClickRec Recommender System

In this section, we describe ClickRec, a news articles recommendation system that processes clickstreams generated by users of a news portal and recommends news articles based on mining frequent pagesets using the MFI-TransSW+ algorithm (Section 3).

We tested ClickRec with three different recommendation approaches: (1) treat the news articles as the items; (2) treat the metadata (e.g. tags, semantic annotations and even editorial category) of the news articles as the items; (3) treat the metadata of the news articles as the items, but adopting a similarity metric over metadata to create recommendations. Due to space limitations, we discuss only the third approach, which produced the best results. Indeed, during our experiments (see Section 5), we identified that most of the users (55% ∼ 70%) accessed only one news article (these users are commonly called *bounce users*), which limits the practical application of the first approach. After several experiments using real clickstreams from a news portal, the third approach obtained the best results.

The third approach works as follows. First, frequent pagesets are mined in a window W. Then, when a user u accesses an article a (that is, a page), the approach compares the metadata of a with the metadata of the most frequent pagesets, searching for the most similar set F. Finally, the first n news articles described by metadata most similar to F are recommended to u.

For example, suppose that a user accesses a news article having as metadata the semantic annotations $S = \{<barcelona>, <neymar>, <messi>\}$. Assume that $F = \{<barcelona>, <neymar>, <messi>, <BBVA_league>\}$ was identified, in the last window, as the frequent itemset most similar to S. Then, the recommendation approach searches for other news articles described by metadata most similar to F. Similarity metrics such as TF-IDF were applied for this task.

ClickRec was implemented in Python, using Redis, a key-value database, to store the bit vectors and other data structures. It has three main modules: *Data Streams Processor*, *Frequent Itemsets Miner* and *Recommender*.

The *Data Streams Processor* module continuously processes clickstreams, maintaining a window of users. Using MFI-TransSW+, it creates or updates the bit vectors, and related structures, of the users belonging to the selected window. This module receives as input the window size (described in number of users).

The *Frequent Pagesets Miner* module can be invoked whenever the transaction window becomes full or after regular time intervals. When this module is activated, the current window is processed to mine the frequent pagesets by the application of stage 3 of MFI-TransSW+, using as input the bit vectors generated by *Data Streams Processor* module.

The *Recommender* module uses frequent pagesets mined by *Frequent Pagesets Miner* module as input to generate recommendations of news articles for a user. This module is invoked whenever a user click on a news article (the current news article information is also an input for this module) and generates associated recommendations in real time. This module is responsible for implementing the recommendation approaches mentioned earlier in this section.

5 Experiments and Results

This section reports two experiments conducted with the MFI-TransSW+ algorithm. The first experiment compares the performance of the original MFI-TransSW and the

proposed MFI-TransSW+ algorithm, whereas the second experiment validates the recommendations provided by ClickRec. A user tracking system was used to collect clickstreams from one of the largest news Web sites in Brazil. In total, 25M of page views were collected per day. The experiments and analysis were performed using one-hour time frames containing, on the average, 1M of page views.

The experiments were performed on a standard PC with 2,5GHz Intel Core i5 Processor and 16GB RAM, using the implementations described in previous sections.

5.1 Comparison between MFI-TransSW and MFI-TransSW+

To compare the performances of the original MFI-TransSW and the optimized MFI-TransSW+, we used a one-hour time frame containing 1M of page views. The experiment was divided into two parts. In the first part, a number of users were processed up to the point the window reached its full size. In the second part of the experiment, the clickstreams were processed until reaching a total of 10k window slides. The performance of both algorithms was only measured in the second part, since both algorithms adopt different approaches. Table 1 shows the results for different window sizes and Figure 1 shows a significant performance gain between both algorithms. The gain in performance of the MFI-TransSW+ grows linearly with the increase of the window size. For a window size of w=10,000, MFI-TransSW+ is up to 900 times faster than the original algorithm, that is, 2 orders of magnitude faster.

Note that the substantial difference in performance between the algorithms occurs because MFI-TransSW+ performs much less operations than MFI-TransSW in each window move. For instance, in a window of size $w = 10.000$, we may have a total of 20k bit vectors. So, when the window reaches its full size, the original algorithm executes a bitwise shift for every bit vector, whereas the optimized algorithm only performs the clean-and-update operation.

Table 1: Runtime comparison between MFI-TransSW and MFI-TransSW+.

Window Size	Runtime (seconds)	
	MFI-TransSW	MFI-TransSW+
1.000	41,45	0,40
2.000	136,73	0,63
3.000	272,23	0,95
4.000	395,54	1,17
5.000	533,10	1,28
6.000	761,30	1,59
7.000	996,09	1,91
8.000	1.295,16	2,07
9.000	1.484,10	2,22
10.000	1.928,76	2,36

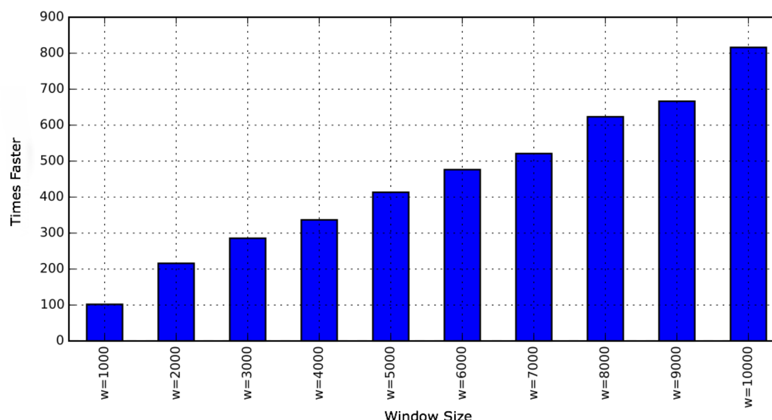

Fig. 1: Performance of the MFI-TransSW and MFI-TransSW+ algorithms.

5.2 Validation of ClickRec recommendations

A preliminary analysis of the clickstreams over a period of one-hour indicated that the vast majority of users of the news portal under analysis were bounce users, that is, users that access only one page during its navigation. Figure 2 shows that the percentage of users that accessed only one page is much higher than those that accessed two or more pages. This analysis was conducted over two different editorial categories (A and B, as in Figure 2) and shows similar user navigation behaviors.

The low number of pages accessed by users is a problem for frequent itemsets mining algorithms as there will be only a few to be mined. To solve this problem, instead of using user page views as input to our system, we opted to use the third metadata approach described in Section 4 to mine the frequent itemsets. As mentioned in Section 4, all news articles have at least one semantic annotation, which were used in this experiment.

To run this experiment, we divided the clickstreams into pairs of two consecutive periods of one hour each. The first clickstream period is used to create the window and generate recommendations. The second clickstream period is used to extract a sample of users and measure the effectiveness of the recommendations. The sample consists only of users who have accessed more than one page during the second clickstream period.

Note that we need users who accessed more than one page because the first page accessed in the first period of the experiment will be used as input to the ClickRec recommendation module. Based on the first page accessed, ClickRec must recommend up to 10 news articles to a given user. If the user accessed one or more of the recommended pages, then the recommendation is considered successful.

Briefly, the following steps are performed for each pair of consecutive hours:

– **Loading:** Load a window with the first clickstream hour and generate the frequent itemsets. This step is equivalent to the steps 1 and 2 of ClickRec.

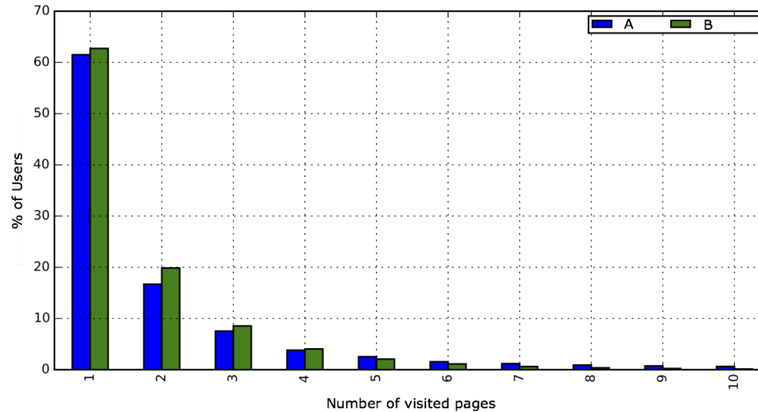

Fig. 2: Number of pages views per user.

- **Sampling:** Extract a sample of 10K users from the second clickstream hour.
- **News Recommendation:** Generate, for each user, recommendations according to his page accessed in the first clickstream period. This step is equivalent to the step 3 of the ClickRec system.
- **Validation:** Validate if the user accessed a page recommended to him. If the user accesses the recommended page, we consider that the recommendation approach succeeded.

Figures 3 and 4 show the results for two distinct editorial categories (A and B). In the worst scenario, ClickRec was able to make correct recommendations for 20% of the users. Although 20% seems to be low, the baseline algorithm provided by the news portal achieved a maximum conversion rate of 10% of the users that actually clicked in the recommended articles (unfortunately, the method and the portal name cannot be disclosed).

We also need to take into account the difficulty in measuring the correct recommendations. Indeed, a user may not access a recommended page either because it is irrelevant or because the page recommendation is not easily recognized due to design issues of the portal. Moreover, specifically in news portals, users tend to access only the top news articles, overlooking related articles.

6 Conclusions

In this paper, we presented the MFI-TransSW+ algorithm, an efficient mining algorithm to process clickstreams, that incorporates a new strategy to maintain bit vectors. Moreover, we described a news articles recommendation system, called ClickRec, that incorporates the MFI-TransSW+ algorithm and features a new recommendation strategy for news articles, based on measuring metadata similarity.

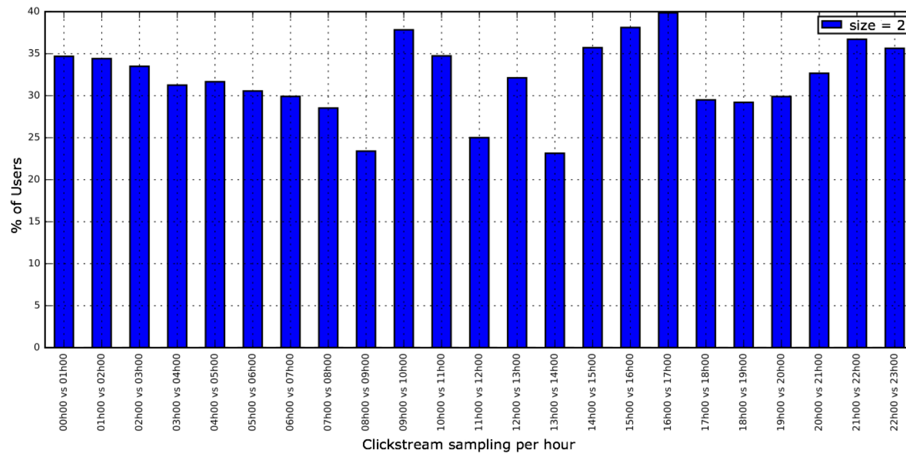

Fig. 3: ClickRec: Number of correct recommendations for the A editorial category.

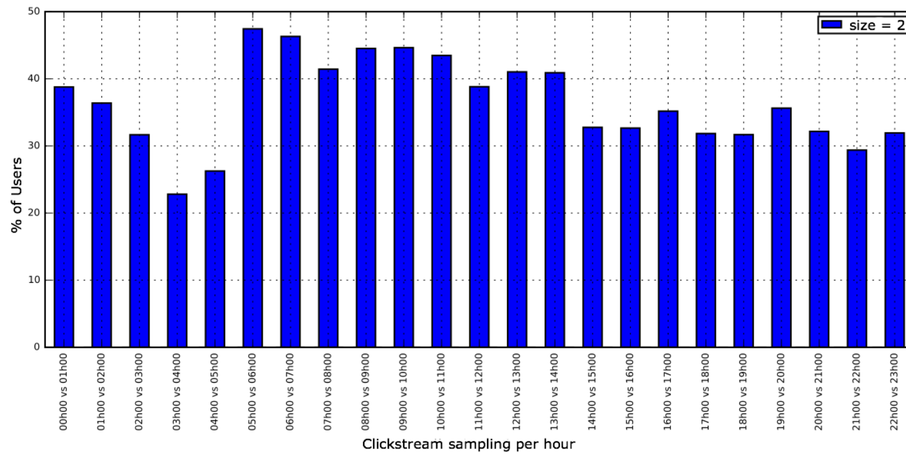

Fig. 4: ClickRec: Number of correct recommendations for the B editorial category.

Finally, we performed experiments to validate the contributions of the paper. A comparison between the original and the proposed MFI-TransSW+ algorithm showed a substantial performance gain of the proposed algorithm, when processing clickstreams. In the experiments, for window sizes of 1K and 10K, the proposed algorithm was 100 to 900 times faster than the original MFI-TransSW algorithm. Furthermore, in the experiments with real data, ClickRec obtained much better conversion rates than the baseline algorithm currently adopted by the news portal.

References

1. R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. *VLDB 1994*, pages 1–32, 1994.
2. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. *ACM PODS 2002*, pages(2002-19):1, 2002.
3. J. H. Chang and W. S. Lee. A Sliding Window Method for Finding Recently Frequent Itemsets over Online Data Streams. *Journal of Information Science and Engineering*, 20(4):753–762, 2004.
4. J. Cheng, Y. Ke, and W. Ng. A survey on algorithms for mining frequent itemsets over data streams. *Knowledge and Information Systems*, 16(1):1–27, 2008.
5. H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics*, pages 493–507, 1952.
6. Y. Chi, H. Wang, S. Y. Philip, and R. R. Muntz. Catch the moment: maintaining closed frequent itemsets over a data stream sliding window. *Knowledge and Information Systems*, 10(3):265–294, 2006.
7. C.-H. Lee, C.-R. Lin, and M.-S. Chen. Sliding window filtering: an efficient method for incremental mining on a time-variant database. *Information Systems*, 30(3):227–244, 2005.
8. H.-F. Li and S.-Y. Lee. Mining frequent itemsets over data streams using efficient window sliding techniques. *Expert Systems with Applications*, 36(2):1466–1477, 2009.
9. H.-F. Li, S.-Y. Lee, and M.-K. Shan. An efficient algorithm for mining frequent itemsets over the entire history of data streams. In *Proc. of First International Workshop on Knowledge Discovery in Data Streams*, 2004.
10. H.-F. Li, S.-Y. Lee, and M.-K. Shan. Online mining (recently) maximal frequent itemsets over data streams. In *RIDE-SDMA 2005*, pages 11–18. IEEE, 2005.
11. G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB 2002*, pages 346–357, 2002.
12. A. L. Montgomery, S. Li, K. Srinivasan, and J. C. Liechty. Modeling online browsing and path analysis using clickstream data. *Marketing Science*, 23(4):579–595, 2004.
13. J. X. Yu, Z. Chong, H. Lu, Z. Zhang, and A. Zhou. A false negative approach to mining frequent itemsets from high speed transactional data streams. *Information Sciences*, 176(14):1986–2015, 2006.