Efficient Itemset Generator Discovery over a Stream Sliding Window

Chuancong Gao, Jianyong Wang
Tsinghua National Laboratory for Information Science and Technology
Department of Computer Science and Technology
Tsinghua University, Beijing 100084, China
gaocc07@mails.tsinghua.edu.cn, jianyong@tsinghua.edu.cn

ABSTRACT

Mining generator patterns has raised great research interest in recent years. The main purpose of mining itemset generators is that they can form equivalence classes together with closed itemsets, and can be used to generate simple classification rules according to the MDL principle. In this paper, we devise an efficient algorithm called StreamGen to mine frequent itemset generators over a stream sliding window. We adopt a novel enumeration tree structure to help keep the information of mined generators and the border between generators and non-generators, and propose some optimization techniques to speed up the mining process. We further extend the algorithm to directly mine a set of high quality classification rules over stream sliding windows while keeping high performance. The extensive performance study shows that our algorithm outperforms other state-of-the-art algorithms which perform similar tasks in terms of both runtime and memory usage efficiency, and has high utility in terms of classification.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data Mining*

General Terms

Algorithms, Experimentation, Performance

Keywords

Stream Data, Sliding Window, Itemset Generator, Feature Selection, Classification

1. INTRODUCTION

Frequent itemset mining is one of the essential data mining tasks. Since it was firstly proposed in [1], various algorithms have been proposed, including Apriori [2] and FP-growth [12] algorithms. Many studies have also demonstrated its application in feature selection and associative classifier construction [18, 9, 14, 17, 3, 26, 8, 24, 5, 6].

If we divide the set of all itemset patterns into a set of equivalence classes, where each equivalence class contains a set of itemset patterns which are supported by the same set of input transactions, the closed itemsets are those maximal ones in each equivalence class. It is evident that the set of closed itemsets is just a subset of all itemset patterns, and thus it is possible to identify some parts of search space which are unpromising to generate any closed itemsets and can be pruned. Thus, closed itemset mining can be potentially more efficient than all itemset mining. Due to the concise representation and high efficiency, many algorithms for mining frequent closed itemsets have been proposed [20, 21, 29, 19, 25, 11].

In each equivalence class of itemset patterns, if we call the minimal ones itemset generators, similarly we get that the set of all itemset generators is a subset of all itemset patterns, and itemset generator mining can be potentially more efficient than all itemset pattern mining too. It is also evident that the average length of itemset generators tends to be smaller than that of all itemset patterns (or closed itemset patterns). Since one of the important applications of frequent itemset mining is to be used for feature selection and associative classifier construction. According to the Minimum Description Length (MDL) Principle, generators are preferable in tasks like inductive inference and classification among the three types of itemset patterns (namely, all itemset patterns, closed itemset patterns,

Recently stream data became ubiquitous. One popular form of such kind of data is a sequence of transactions arriving in order continuously. They usually come at a high speed and have a data distribution that often evolves with time. Due to the unique characteristics of the stream data, it is not feasible to simply adapt the algorithms originally designed for static datasets to stream data. Hence several efforts have been devoted to frequent itemset mining and closed itemset mining over stream data [23, 28, 7, 13]. However, to our best knowledge, there exists no algorithm which mines frequent itemset generators over a stream sliding window, while such an algorithm is very useful in building associative classifiers over stream data.

In this paper, we introduce an efficient algorithm, StreamGen, to mine frequent itemset generators over sliding windows on stream data. It adopts the FP-Tree structure to concisely store the transactions of the current window, and devises a novel enumeration tree structure to keep all the mined generators and their border to the non-generators. In the meantime, some optimization techniques are also proposed to accelerate the mining process. To demonstrate its utility, we further extend StreamGen to directly mine classification rules over a stream sliding window. The experimental study shows that StreamGen is efficient and achieves high classification accuracy.

The contributions of the paper are summarized as follows.

- We devise the first algorithm on mining frequent itemset generators over sliding windows on stream data, StreamGen.
- We propose a novel enumeration tree structure and explores some effective optimization techniques to enhance the efficiency of the StreamGen algorithm.
- We extend StreamGen and devise an algorithm to directly mine classification rules on a sliding window.
- An extensive performance study was conducted, which shows that StreamGen is very efficient, outperforms other state-of-the-art algorithms performing similar tasks to StreamGen, and achieves high accuracy in classifying categorical data.

The remainder of this paper is organized as follows. In Section 2, we introduce the related work. In Section 3, we present the problem statement. The details of StreamGen are discussed in Section 4. Section 5 describes the extended algorithm to mine classification rules directly over a stream sliding window. The empirical results are shown in Section 6, and we conclude the paper in Section 7.

2. RELATED WORK

ZIGZAG [23] is an algorithm designed for mining all frequent itemsets over a sliding window. The algorithm supports batch update, and hence outperforms other algorithms updating one transaction at a time when the batch size is large. [28] also discusses frequent itemset mining from transactional data streams. [7] proposes an algorithm called MOMENT to mine frequent closed itemsets over a stream sliding window. It adopts the FP-Tree structure to compress transactions in the current window, and an enumeration tree to maintain the mined closed itemsets. While CFI-Stream proposed in [13] is another algorithm which only keeps closed itemsets in its enumeration tree to further compress the storage and accelerate the mining process. To our best knowldge, currently there is no algorithm which mines frequent itemset generators over a stream sliding window, although there exist several frequent itemset generator mining algorithms for static dataset, such as GR-Growth [15], DPM [16], and an algorithm for incremental mining of itemset generators, such as [27].

One important application of frequent itemset mining is feature selection for building classification models. There are several pieces of work which try to directly mine a set of itemset patterns for classification. The HARMONY algorithm [26] tries to directly mine k best rules for each transaction, and use them for building a rule-based classifier. [8] proposes another algorithm to mine top-K associative classification rules on gene data. [5] proves that information gain should be preferred to confidence in mining classification rules, and proposes an algorithm using information gain to select rules. [6] further devises an algorithm called DDPMine to directly mine rules using a sequential covering paradigm. There is no algorithm which directly mines a set of itemset generators for classification. [10] tries to mine sequential generators for classifying sequential data and achieves good accuracy.

3. PROBLEM STATEMENT

Given a set of items I={i_1, i_2, . . ., i_l}, a transaction database D consists of a set of transactions and a transaction is a tuple $<tid$, $T>$, where tid is the transaction identifier (or time stamp), and $T \subseteq I$. An itemset (i.e., a set of items) S is said to be contained in a transaction $<tid, T>$ if $S \subseteq T$ holds. The number of transactions containing itemset S is called the *absolute support* of S, and the percentage of transactions that contain S is called the *relative support*. In the following we will use *support* to denote *absolute support* and *relative support* interchangeably when there is no confusion, and use sup_S to denote the support of itemset S.

Given a user specified minimum support threshold sup_{min}, we have the following definitions.

DEFINITION 1. *An itemset S is frequent if and only if $sup_S \geq sup_{min}$.* □

DEFINITION 2. *A frequent itemset generator (or shortly generator) S is a frequent itemset where there is no itemset S^* such that $S^* \subset S$ and $sup_{S^*} = sup_S$.* □

DEFINITION 3. *An unpromising itemset is a frequent non-generator itemset.* □

COROLLARY 1. *A frequent itemset S is unpromising iff $\exists S^*$ such that $|S^*| = |S| - 1$ and $sup_{S^*} = sup_S$.* □

The main task of this work is *to mine the complete set of frequent itemset generators from the most recent sliding window of M transactions in a transactional data stream*. To show the utility of itemset generator mining, we will also discuss how to mine generator-based classification rules over a sliding window. Figure 1 shows a running example of transactional data stream with a sliding window size of 4.

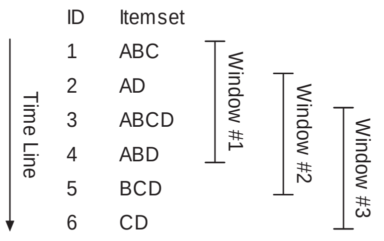


Figure 1: A Running Example of Transactional Stream Data.

4. THE STREAMGEN ALGORITHM

We introduce the StreamGen algorithm in details in this section. First, we present and prove some common properties which will be used in the algorithm design. Then, we introduce the FP-Tree structure for storing transactions in the current sliding window in Section 4.1, the enumeration tree structure in Section 4.2, the ADD operation in Section 4.3, and the REMOVE operation in Section 4.4, respectively. Finally, we will discuss how to combine the ADD and REMOVE operations to get the integrated StreamGen algorithm for mining itemset generators over a stream sliding window in Section 4.5.

Based on the definitions in Section 3, we have the following properties.

THEOREM 1. *A frequent itemset S is a generator iff there exists no subset S^* such that $|S^*|$=$|S|$−1 and sup_{S^*}=sup_S.*

PROOF. **Sufficiency**. Assume there does not exist a subset S^* such that $|S^*|$=$|S|$−1 and sup_{S^*}=sup_S, we prove that itemset S must be a generator.

Suppose S is not a generator, then there must exist a subset S^{**} that sup_S=$sup_{S^{**}}$. According to assumption that there does not exist a subset S^* such that $|S^*|$=$|S|$−1 and sup_{S^*}=sup_S, we can conclude that $|S^{**}|<|S|-1$, $sup_{S^{**}}=sup_S$, and there exists at least one subset S^* such that $S^{**}\subset S^*\subset S$ and $|S^*|=|S|-1$. However, according to the Apriori principle, we also know that $sup_{S^{**}}\geq\ sup_{S*}\geq\ sup_S$ must hold. Hence we further conclude that $sup_{S^*}=sup_S$, which contradicts the assumption.

Necessity. It can be easily derived from Definition 2. □

THEOREM 2. *Given a generator S, any subset of S would be also a generator.*

PROOF. Assume S is a generator and has a non-generator subset S^*. Then there must exist a generator subset S^{**} of S^* such that $sup_{S^{**}}=sup_{S^*}$. We could further conclude that for the itemset $U=S-(S^*-S^{**})$ we have $U\subset S$ and $sup_U=sup_S$, thus, we have that S is not a generator, which contradicts the assumption. □

THEOREM 3. *Given an unpromising itemset S, any superset of S must be either unpromising or infrequent.*

PROOF. It can be easily derived from Theorem 2 and the Apriori property. □

The above three theorems were used in several studies such as [4]. Theorems 2 and 3 help define the border between generators and non-generators, and form the foundation for the enumeration tree used in our algorithm. While with Theorem 1 we can check whether an itemset S is a generator or not by simply checking all the itemsets which are a subset of S and have a length of $|S|-1$.

THEOREM 4. *Given an itemset S and its superset $S^* = S \cup \{x\}$, then for any itemset Y we have either $|S^* \cap Y| = |S \cap Y|$ or $|S^* \cap Y| = |S \cap Y| + 1$.*

PROOF. The proof of this theorem is obvious. □

The new property shown in Theorem 4 can be easily proved. With theorem 4, we could know whether the state of an itemset S will change to another or not when a new transaction Y arrives, and it also helps the intersection calculation which will be introduced in Section 4.3.

THEOREM 5. *Given two itemsets S and T, we have $S \subseteq T$ if and only if $|S \cap T| > |S| - 1$.*

PROOF. **Sufficiency**. Assume $|S \cap T| > |S| - 1$, we have $|S \cap T| = |S|$, thus $S \subseteq T$ holds.

Necessity. Assume $S \subseteq T$, we have $|S \cap T| = |S|$, thus $|S \cap T| > |S| - 1$ holds. □

Theorem 5 is used later in both the ADD and REMOVE operations introduced in Sections 4.3 and 4.4.

4.1 FP-Tree

Like the Moment algorithm [7], we also adopt a variant of the FP-Tree structure [12] to help maintain a concise representation of the transactions in the current sliding window. The use of FP-Tree structure not only reduces the need for memory, but also accelerates some operations such like support counting, etc.

The FP-Tree structure adopted in our algorithm is a variant of the CET tree used by Moment. That is, it keeps not only the frequent items, but also the infrequent items for further processing. Besides, the items are not sorted by supports, but in lexicographic order to avoid re-sorting the items each time as the sliding window moves. Note that with an FP-Tree structure it is no longer necessary to keep any transactions in the sliding window, and all the information could be retrieved or calculated easily from FP-Tree.

Figure 2 shows the FP-Tree built from the first sliding window of the running example shown in Figure 1. We notice that a transaction ID table is used to map all those transactions in the current sliding window to those FP-Tree nodes which contain the first item in the sorted itemset of each transaction.

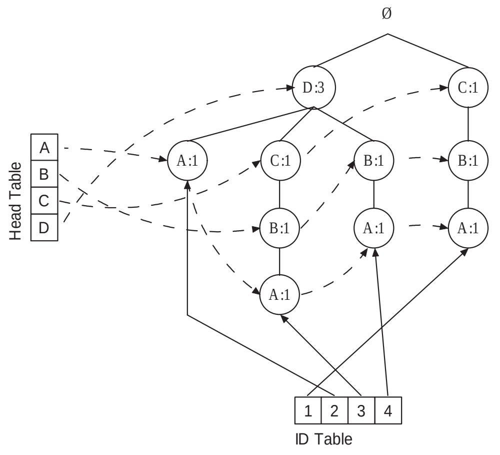


Figure 2: The FP-Tree of the First Sliding Window.

4.2 The Enumeration Tree

In the algorithm we use an enumeration tree to help maintain the whole relationship between itemsets and the mined generators. The idea of using enumeration tree was inspired by [7]. There are three types of tree nodes in the enumeration tree.

- **Infrequent Node**: An infrequent node represents an infrequent itemset. Note that in our algorithm we only keep those infrequent nodes which do not have infrequent uncles (i.e., the nodes which have the same parent node with the parent node of the corresponding infrequent node).
- **Unpromising Node**: An unpromising node represents an unpromising itemset. Like the infrequent node we only keep those unpromising nodes which do not have unpromising uncles.
- **Generator Node**: A generator node represents a frequent generator itemset. In our algorithm we keep all the frequent generators in the current sliding window.

Note that we do not need the gateway node used in [7], since according to Theorem 2 there is no non-generator node whose itemset is a subset of the itemset of some generator nodes. Furthermore, the existence of infrequent and unpromising nodes forms the border of the generators to the non-generators in the enumeration tree.

To accelerate the checking operation on whether an itemset is a generator or not, we adopt a hash table structure whose key is the sum of items in one itemset, and whose value is a pointer to the enumeration tree node which contains the itemset. Each node in the same level is stored in a same hash table. Hence it is very useful to focus on the desired nodes by only checking one hash table whose level is smaller by one than the level of current node.

As described above, instead of pruning all infrequent nodes and unpromising nodes from the enumeration tree, we only prune those infrequent nodes and unpromising nodes which have an infrequent or an unpromising uncle node. The reason we adopt this strategy is that the ‘complete pruning’ may cost a lot when either adding or removing a transaction, and the saved time by ‘complete pruning’ could not even offset the time used in ‘complete pruning’ itself.

Figure 3 shows the enumeration tree built from the first sliding window of the running example. An ellipse with solid line style indicates a generator node, an ellipse with dotted line style indicates an unpromising node, and a rectangle with dotted line style indicates an infrequent node. Note that the empty set ∅ is treated as a valid itemset in this paper, and the minimum absolute support threshold here is set at 2.

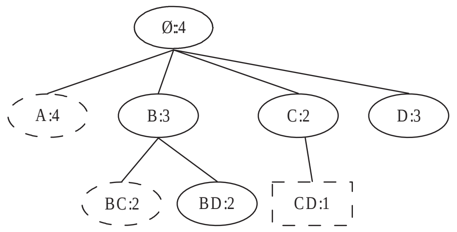


Figure 3: The Enumeration Tree of the First Sliding Window.

4.3 The ADD Operation

The ADD operation in StreamGen mines and maintains the set of frequent generators when a new transaction arrives. Before we introduce the details of the ADD operation, we first prove some properties which are helpful in understanding the algorithm.

THEOREM 6. *During the ADD operation, a generator node would never change its type.*

PROOF. Given a generator node n whose corresponding itemset is $itemset_n$, we have that for any itemset $S \subset itemset_n$, $sup_S > sup_n$ holds. If the new transaction is a superset of $itemset_n$, the support of $itemsets_S$ and all its subsets would all increase by one, hence n would remain a generator node. While if the new transaction is not a superset of $itemset_n$, only some subsets of $itemset_n$ would get their support increased by one and thus n still remains a generator node. □

This theorem indicates that we only need to care about the support update of a generator node, but do not need to consider the change of its node type since it would never happen.

THEOREM 7. *If the itemset of an unpromising node is a subset of the new transaction, the node would not change its type.*

PROOF. A node n whose itemset is $itemset_n$ is unpromising means there exists a subset S of $itemset_n$ satisfying $sup_S = sup_{itemset_n}$. If $itemset_n$ is a subset of the new transaction, the support of both $itemset_n$ and its subset S will be all increased by one, hence node n remains an unpromising node. □

THEOREM 8. *Given a new transaction T, if an unpromising node n whose itemset is $itemset_n$ becomes a generator node it must satisfy $|itemset_n \cap T| = |itemset_n| - 1$.*

PROOF. If $|itemset_n \cap T| < |itemset_n| - 1$, then there does not exist any subset S of $itemset_n$ satisfying $|S| = |itemset_n| - 1$ and gets its support increased by one. Hence n would remain unpromising according to Theorem 1. If $|itemset_n \cap T| > |itemset_n| - 1$, we could know that its type would also not change according to Theorem 7. □

THEOREM 9. *Given a new transaction T, if a node n whose itemset is $itemset_n$ satisfies $|itemset_n \cap T| < |itemset_n| - 1$, the state of node n and all its descendant nodes will remain unchanged.*

PROOF. We could easily prove that the node type would not change according to Theorems 6 and 8. Furthermore, we can prove that all its descendants would also satisfy the condition of this theorem according to Theorem 4. Hence the state of node n and all its descendant nodes will remain unchanged. □

THEOREM 10. *When an infrequent node becomes a generator node, all its newly added child nodes will be either infrequent or unpromising.*

PROOF. When an infrequent node n becomes a generator node, it must satisfy $sup_n = sup_{min}$. Hence any new frequent child node of n, n^*, must satisfy $sup_n = sup_{n^*}$ and thus is an unpromising node; and all its infrequent children are infrequent nodes. □

Theorem 10 is very useful since it guarantees that we do not need to check the new child nodes if they are generator nodes, which saves much time.

After introducing the properties related to itemset generators, we will discuss how to explore these properties to design an efficient generator mining algorithm (i.e., the ADD operation) upon receiving a new transaction.

Before we elaborate on the ADD operation, we first introduce a sub-procedure, $explore(n)$, which is shown in Algorithms 1 and describes how to explore a node n. Note that the algorithm calls a function of $newChild(n, i)$ which is used to create a new child node of node n with an itemset of $itemset_n \cup \{i\}$. The algorithm creates the child node by merging the itemset of node n with the lexicographically largest item of the itemset of one of n's siblings which have the same parent as node n. It also re-creates the child nodes which were pruned out due to the existence of an infrequent or unpromising uncle node, and hence extends the border of the enumeration tree. Note that we only need to re-generate the child nodes with only one level lower, since Theorem 10 assures that all the newly added child nodes of a generator node which are infrequent nodes would become either infrequent or unpromising. When an unpromising node n becomes a generator node after receiving a new transaction T, it must satisfy $|itemset_n \cap T| = |itemset_n| - 1$ according to Theorem 8. Hence any new frequent child node n^* would satisfy $|itemset_{n*} \cap T| = |itemset_{n*}| - 1$ or $|itemset_{n*} \cap T| < |itemset_{n*}| - 1$ according to Theorem 4. If we have $|itemset_{n*} \cap T| < |itemset_{n*}| - 1$, the new child can be safely skipped according to Theorem 9. If we have $|itemset_{n*} \cap T| = |itemset_{n*}| - 1$ and the new child node is a generator, it will be traversed later.

Algorithm 1: $explore(n)$

```
  Input  : A node $n$ of the enumeration tree.
1 begin
2     foreach $x \in \{y | parent_y = parent_n, max_{itemset_y} > max_{itemset_n}$, $y$ is a generator$\}$ do
3         $newChild(n, max_{itemset_x})$;
4     foreach $x \in \{y | parent_y = parent_n, max_{itemset_y} < max_{itemset_n}$, $y$ is a generator$\}$ do
5         if not $hasChild(x, max_{itemset_n})$ then
6             $newChild(x, max_{itemset_n})$;
7 end
```

Algorithm 2 gives the details of the ADD operation, from which we see that it uses a right-to-left, top-down updating strategy. The top-down strategy is necessary due to the fact that the checking of whether a node is a generator node or not must examine those nodes which are on a higher level in the enumeration tree and hence those nodes must have been updated before updating the current node. While a right-to-left strategy is used due to the fact the uncle nodes used in Algorithms 1 are all on the right side to the current node and hence we must assure those nodes have already been created if they do not exist in the last sliding window.

The algorithm determines how to deal with the update according to whether $|itemset_n \cap T|$ is equal to, less than, or great than $|itemset_n \cap T| - 1$.

```
Algorithm 2: $add(r, T)$
Input : The root node of the enemuration tree $r$, and the newly
        arrived transaction $T$.
begin
    $enqueue(r)$;
    while $q$ is not empty do
        $n \leftarrow dequeue()$;
        if $|itemset_n \cap T| = |itemset_n \cap T| - 1$ then
            if $n$ is unpromising then
                $identify(n)$;
                if $n$ is unpromising then
                    continue;
                $explore(n)$;
            foreach $cn \in children_n$ in reversed order do
                if $cn$ is not infrequent then
                    $enqueue(cn)$;
        else
            if $|itemset_n \cap T| > |itemset_n \cap T| - 1$ then
                $sup_n \leftarrow sup_n + 1$;
                if $n$ is unpromising then
                    continue;
                if $n$ is infrequent then
                    if $sup_n < sup_{min}$ then
                        continue;
                    $identify(n)$;
                    if $n$ is generator then
                        $explore(n)$;
            else
                foreach $cn \in children_n$ in reversed order do
                    $enqueue(cn)$;
                $m \leftarrow max_{itemset_n}$;
                foreach $i \in \{c|c \in T, c > m\}$ do
                    if $node_{\{itemset_n - \{m\}\} \cup \{i\}}$ is generator
                    then
                        $newChild(n, i)$;
end
```

In addition, there is no need to calculate the itemset intersection every time. Since according to Theorem 4 we can always calculate the intersection incrementally from the last intersection, and hence save a lot of time. Note that in our algorithm we do not prune all the infrequent and/or unpromising nodes whose itemsets are supersets of the itemsets of some infrequent and/or unpromising nodes due to the potential cost of rebuilding those nodes in a later sliding window and the significant pruning cost. The sub-procedure $identify(n)$ is used for updating the type information of current node.

Figure 4 illustrates the enumeration tree built from the first sliding window after adding a new transaction (i.e., the transaction with an ID of 5).

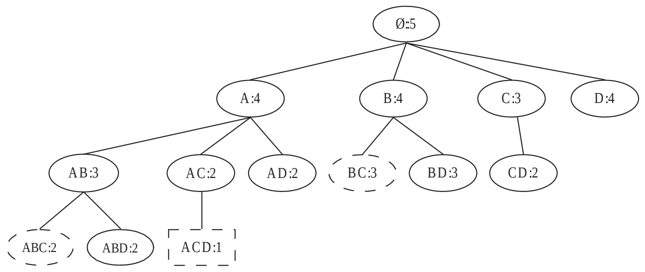


Figure 4: The Enumeration Tree after Adding a Transaction.

Finally, we could easily prove the number of nodes in the enumeration tree does not decrease during the ADD operation.

4.4 The REMOVE Operation

We will introduce the REMOVE operation in this section, which mines and maintains frequent itemset generators upon removing an old transaction.

THEOREM 11. *During the REMOVE operation, an unpromising node would never change its type unless it becomes infrequent or is pruned.*

PROOF. Given an unpromising node n, we know that there exists an itemset S such that $S \subset itemset_n$ and $sup_S = sup_{itemset_n}$, which means a removed transaction T either contains both S and $itemset_n$ or contains neither of S and $itemset_n$. Thus, by removing transaction T, the relationship between S and $itemset_n$ remains unchanged, namely, $S \subset itemset_n$ and $sup_S = sup_{itemset_n}$ still hold. □

THEOREM 12. *By removing a transaction T, a generator node n would not change its type if it satisfies $|itemset_n \cap T| > |itemset_n| - 1$.*

PROOF. Since n is a generator node, there does not exist any subset S such that $|S| = |itemset_n| - 1$ and $sup_S = sup_n$. The condition of $|itemset_n \cap T| > |itemset_n| - 1$ means T contains $itemset_n$, thus, by removing transaction T, not only the support of n but also the support of all its subsets would be decreased by one. Hence the node n would not change its type unless it becomes infrequent. □

THEOREM 13. *By removing a transaction T, a generator node n would become an unpromising node only when it satisfies $|itemset_n \cap T| = |itemset_n| - 1$ and $sup(itemset_n \cap T) = sup_n + 1$.*

PROOF. As a generator node n becomes unpromising after removing transaction T, according to Theorem 1, there must exist at least one subset S such that $|S| = |itemset_n| - 1$ and its support sup_S becomes exactly $sup_{itemset_n}$. As the deletion of T can at most decrease sup_S by one, thus $sup_S = sup_n + 1$ must hold before removing T, and we can derive that the only condition which makes a generator node n becomes unpromising is that $|itemset_n \cap T| = |itemset_n| - 1$ and $sup(itemset_n \cap T) = sup_n + 1$. □

THEOREM 14. *In removing an old transaction T, if a node n whose itemset is $itemset_n$ satisfies $|itemset_n \cap T| < |itemset_n| - 1$, the state of node n and all its descendant nodes will remain unchanged.*

PROOF. The proof is similar to that of Theorem 9. □

After introducing some nice properties of the generator nodes and the unpromising nodes which can be used to enhance the efficiency of mining and maintaining the set of frequent generators during the REMOVE operation, we now turn to the algorithm for the REMOVE operation. Algorithm 4 shows the details of the REMOVE operation. Like the ADD operation, we also adopt a right-to-left and top-down updating strategy here. The algorithm also determines how to deal with the update according to whether $|itemset_n \cap T|$ is equal to, less than, or great than $|itemset_n \cap T| - 1$.

The REMOVE operation invokes a procedure, $clean()$, which is depicted in Algorithm 3. Algorithm 3 is used to clean those nodes some of whose uncle nodes change their types to infrequent or unpromising. In Algorithm 3, the function $cleanChildren(n)$ is used to clean all child nodes of node n, and function $removeChild(n, i)$ is used to remove the child node of node n whose itemset is $itemset_n \cup \{i\}$.

Algorithm 3: $clean(n)$

```
Input   : A node $n$ of the enemuration tree.
1  begin
2      $cleanChildren(n)$;
3      foreach $x \in \{y|parent_y = parent_n, max_{itemset_y} < max_{itemset_n}, y$ is a generator$\}$ do
4          if $hasChild(x, max_{itemset_n})$ then
5              $z \leftarrow getChildren(x, max_{itemset_n})$
6              if $z$ is generator then
7                  $clean(z)$
8              $removeChild(parent_z, max_{itemset_z})$
9  end
```

Algorithm 4: $remove(r)$

```
Input   : The root node $r$ of the enemuration tree.
1  begin
2      $T \leftarrow$ the oldest transaction in sliding window;
3      $enqueue(r)$;
4      while $q$ is not empty do
5          $n \leftarrow dequeue()$;
6          if $|itemset_n \cap T| = |itemset_n \cap T| - 1$ then
7              if $n$ is generator then
8                  $identify(n)$;
9                  if $n$ is unpromising then
10                     $clean(n)$;
11                 else
12                     foreach $x \in children_n$ in reversed order do
13                         if $x$ is generator then
14                             $enqueue(x)$;
15         else
16             if $|itemset_n \cap T| > |itemset_n \cap T| - 1$ then
17                 $sup_n \leftarrow sup_n - 1$;
18                 if $n$ is infrequent and $sup_n = 0$ then
19                     $removeChild(parent_n, max_{itemset_n})$;
20                 else
21                     if $sup_n < sup_{min}$ then
22                         if $n$ is generator then
23                             $clean(n)$;
24                         set $n$ as infrequent;
25                     else
26                         if $n$ is generator then
27                             foreach $y \in children_n$ in reversed
                               order do
28                                 $enqueue(y)$;
29 end
```

Algorithm 4 shows the details of the $REMOVE$ operation. Like the ADD operation, we also adopt a right-to-left and top-down updating strategy here.

Figure 5 depicts the enumeration tree built from the second sliding window (namely, after adding a new transaction with an ID of 5 and removing an old transaction with an ID of 1).

Similarly, we could prove the number of nodes in the enumeration tree would not increase during the REMOVE operation.

From the above analysis we could easily see that both ADD and REMOVE operations determine how to update the enumeration tree according to whether $|itemset_n \cap T|$ is equal to, less than, or greater than $|itemset_n \cap T| - 1$, where $itemset_n$ is the itemset represented by any a node n in the tree, T is the transaction being added to (or removed from) the current sliding window. Hence, we summarize the transforming matrix in Table 1.

4.5 The Integrated StreamGen Algorithm

After introducing some properties, the FP-tree like structure to store the transactions of the current sliding window, the enumeration tree structure to maintain the set of generators (or potential generators), the ADD operation and the REMOVE operation, we can easily derive the integrated StreamGen algorithm to mine frequent itemset generators over a stream sliding window. The FP-tree like structure and the enumeration tree structure are initialized to empty. After receiving a new transaction, StreamGen performs the ADD operation as shown in Algorithm 2, and if the size of the current sliding window exceeds the user-specified sliding window size, it then performs the REMOVE operation as shown in Algorithm 4. The set of generators for the current sliding window is always maintained and can be found in the enumeration tree. In addition, although the above StreamGen algorithm mines generators over a stream sliding window, we need to point out that we can easily turn it into an incremental algorithm if we do not apply the REMOVE operation, as the ADD and REMOVE operations are totally independent of each other.

Note that since the two operations are not related, we could combine them freely for specific environments. For example, if we only adopt ADD operation, we could easily get an incremental algorithm.

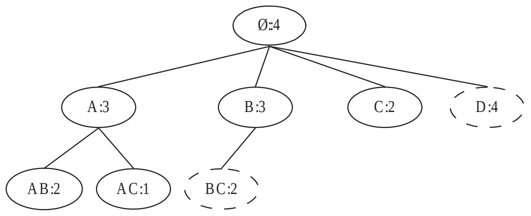


Figure 5: The Enumeration Tree of the 2nd Sliding Window.

	ADD			REMOVE		
Type	$x < y$	$x = y$	$x > y$	$x < y$	$x = y$	$x > y$
G	G	G	G	G	G/U	I/G
U	U	G/U	U	U	U	I/U
I	I	I	I/G/U	I	I	I

Table 1: Transforming matrix for Add and Remove operations ($x = |itemset_n \cap T|$, $y = |itemset_n| - 1$, G = Generator, U = Unpromising, I = Infrequent)

5. EXTENSION FOR MINING CLASSIFICATION RULES

Since one important application of itemset generator mining is to construct concise classification rules, we further extend the StreamGen framework to directly mine generator-based classification rules.

Since the complete set of frequent itemset generators for the current sliding window are maintained in the enumeration tree structure, it is straightforward to retrieve all the generators from the enumeration tree, which can be further used to build classification rules. Algorithm 5 shows the framework StreamGenRules for classification rule construction. It adopts the information gain as a measure of discriminative ability which has been proved better than confidence in [5]. Note that it very easy to change our framework to the confidence-based.

It is very easy to prove that all the itemsets in the same equivalence class have the same information gain (and, the same confidence) due to the fact they have the same supporting set. Hence the generators could represent all the discriminative rules.

In Algorithm 5, the function getGenerators is used to traverse the whole enumeration tree and return all the generators, however, we

Algorithm 5: $StreamGenRules(n)$

```
   Input : The root node $n$ of the enemuration tree.
1  begin
2      $nodes \leftarrow getGenerators(n)$;
3      sort nodes by info-gain;
4      $rules \leftarrow \emptyset$;
5      foreach $cn \in nodes$ do
6          if $\forall r \in rules, r \not\subset cn$ then
7              if cn covers at least one transaction then
8                  $rules \leftarrow rules \cup \{cn\}$;
9                  remove covered transactions;
10                 if no more transactions then
11                     break;
12     return $rules$;
13 end
```

could apply a pruning technique in the function which prunes all those unpromising child nodes whose information gain is smaller than their ancestors and return only a subset of (and sometimes only a small subset of) high quality generators.

The algorithm can be easily adapted to mine a set of rules covering each transaction more than one time. However, the change could cause the loss of pruning power in $getGenerators(n)$ which could prune a great many of unpromising generators. Besides, there is no evidence that covering each transaction more than one times could provide a better accuracy [22].

The StreamGenRules framework seems much like the DDPMine [6] algorithm at the first glance. Both algorithms use information gain as the discriminative measure, and the sequential covering paradigm to select rules. However, there exists a significant difference between StreamGenRules and DDPMine. StreamGenRules tries to find the best rules from the generators which are computed from the full set of transactions in the current sliding window, while DDPMine tries to find the best discriminative rules from those transactions which have not been covered and removed. Hence the rules found by StreamGenRules are globally optimal, while DDPMines finds only those locally optimal rules, thus the rules returned by StreamGenRules tend to have better accuracy in classification. From the experimental results in Section 6.2 we could easily validate that our method can achieve a better accuracy on average. Actually, our algorithm accords with the idea first proposed in [26], which tries to find one most discriminative rule for each transaction.

6. EXPERIMENTAL RESULTS

In this section we evaluate the performance and classification accuracy of our StreamGen algorithm in comparison with several state-of-the-art algorithms. The performance study was conducted on a computer with Intel Core Duo 2 E6550 CPU and 2GB memory installed. A set of UCI datasets were used in the experiments, and Table 2 shows the dataset characteristics. The datasets with two class labels were used in classification accuracy evaluation, while the datasets with multiple class labels, and the horse and mushroom datasets were used in performance test. Note all these datasets are publicly available and have been used widely in evaluating various data mining algorithms.

6.1 Performance Evaluation

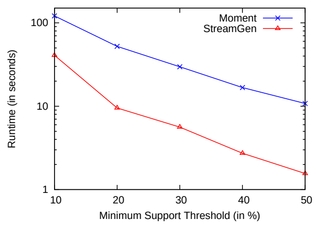
As there is no existing algorithm which mines frequent itemset generators over a sliding window, we evaluate the runtime efficiency of StreamGen algorithm in comparison with three state-of-the-art algorithms which perform a similar task to StreamGen. The first algorithm is Moment which mines frequent closed itemsets over a stream sliding window [7], the second one is DPM which mines frequent itemset generators and frequent closed itemsets in equivalence classes [16], and the third one is DDPMine which directly mines classification rules [6].

Dataset	# Items	# tran.	# Pos.	# Neg.	Avg. Len.
mushroom	116	8,124	4,208	3,916	21.695
horse	89	368	232	136	16.769
adult	128	48,842	11,687	37,155	13.868
breast	45	699	458	241	8.977
hepatitus	55	155	32	123	17.923
pima	40	768	500	268	8
chess	75	3,196	-	-	37
connect	129	67,557	-	-	43
pumsb	2,113	49,046	-	-	74

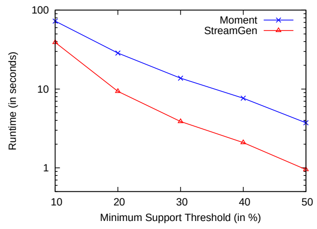
Table 2: Dataset characteristics.

6.1.1 Comparison with Moment

We used four datasets, mushroom, chess, pumsb, and connect-4, to compare the performance of StreamGen with Moment. Figure 6 shows the efficiency comparison on mushroom dataset with a sliding windows size of 4, 000 and 2, 000, respectively.



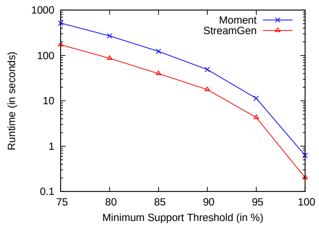
$a)\ dataset = mushroom,$ $window\_size = 2,000$



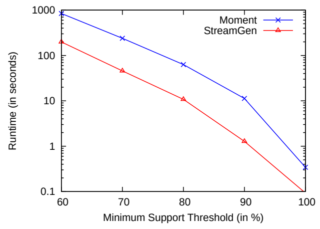
$b)\ dataset = mushroom,$ $window\_size = 4,000$

Figure 6: Runtime comparison with MOMENT

Figure 7 depicts the evaluation results on chess dataset with a sliding window size of 2, 000 and 1, 000, respectively. From the results we could find that our algorithm is significantly faster than Moment algorithm on datasets mushroom and chess.



$a)\ dataset = chess,$ $window\_size = 1,000$

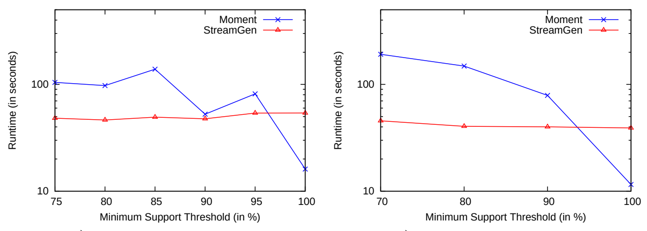


$b)\ dataset = chess,$ $window\_size = 2,000$

Figure 7: Runtime comparison with MOMENT

We also evaluated the algorithm on dataset pumsb, and Figure 8 shows the result on a sliding window size of 10, 000 and 2, 500, respectively. We see that StreamGen outperforms Moment in most cases except when the support is extremely high. We do not provide the result corresponding to the support of 0.6, since Moment ran out of all the available memory (i.e., $2GB$) while our algorithm consumed less than $100MB$ memory.

Figure 9 demonstrates the results on connect-4 dataset with a sliding window size of 60, 000 and 30, 000, respectively. The re-

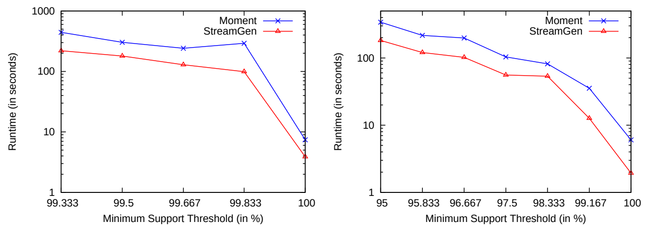


$a)\ dataset = pumsb,$
$window\_size = 2,500$

$b)\ dataset = pumsb,$
$window\_size = 10,000$

Figure 8: Runtime comparison with MOMENT

sults indicate a similar result that our algorithm outperforms Moment in all situations. Note that we do not provide the result on a lower sliding window size as Moment could not finish in an acceptable time.



$a)\ dataset = connect,$
$window\_size = 30,000$

$b)\ dataset = connect,$
$window\_size = 60,000$

Figure 9: Runtime comparison with MOMENT

Table 3 compares the peak memory usage on the datasets used above with the lowest minimum support used in performance test. We see that StreamGen always uses less memory than Moment, thus StreamGen is more memory efficient.

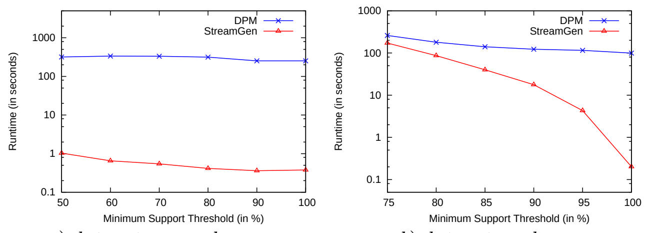
Dataset	$window\_size$	sup_{min}	Moment	StreamGen
mushroom	4,000	0.1	14,476	10,108
mushroom	2,000	0.1	12,504	8,472
chess	2,000	0.6	103,180	31,636
chess	1,000	0.75	34,624	9,176
connect-4	60,000	0.95	141,756	98,236
connect-4	30,000	0.998	73,056	52,372
pumsb	10,000	0.7	1,732,136	75,316
pumsb	2,500	0.75	90,944	23,472

Table 3: Peak memory usage comparsion

6.1.2 Comparison with DPM

To our best knowledge, DPM is the newest and fastest algorithm for mining frequent itemset generators. A naïve way to adapt it to mine frequent itemset generators over a stream sliding window is to run DPM on each new sliding window (In the following we denote this approach by DPM-stream). Our performance comparison between StreamGen and DPM-stream in the data stream setting shows that DPM-stream is not feasible in terms of runtime efficiency. Note that the runtime of DPM-stream is measured only on the sliding windows with a full size, which means we ignored the time period for DPM-stream before the sliding window reaches its full size. If this period is also considered for DPM-stream, the runtime of DPM-stream is even longer and DPM-stream could not terminate in an acceptable time on most datasets.

Figure 10 a) shows the comparison results on mushroom dataset with a sliding window size of $4,000$. DPM's performance is very stable when the minimum support varies, while our algorithm could be a little slower when the support becomes lower. However, the advantage of our algorithm is evident. We could see from Figure 10 a) that our algorithm is orders of magnitude faster than DPM. Figure 10 b) shows the results on chess with a sliding window size of $1,000$, which is similar to that on mushroom.



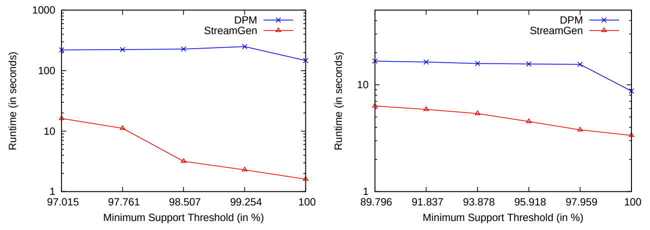
$a)\ dataset = mushroom,$
$window\_size = 4,000$

$b)\ dataset = chess,$
$window\_size = 1,000$

Figure 10: Runtime comparison with DPM

We also evaluated the performance on connect-4 dataset with a sliding window size of $67,000$. We did not provide the result on other window size because DPM is too slow and could not finish in an acceptable time. Note that $67,000$ is almost the biggest window size given the whole size of connect-4 of $67,557$. The result in Figure 11 a) shows our algorithm outperforms DPM significantly.

Figure 11 b) provides the result on pumsb with an extremely large sliding window size of $49,000$ compared with the size of the entire dataset, $49,046$. We do not provide the results on a lower sliding window size as DPM cannot finish in an acceptable time. We see that our algorithm is more efficient than DPM.



$a)\ dataset = connect,$
$window\_size = 67,000$

$b)\ dataset = pumsb,$
$window\_size = 49,000$

Figure 11: Runtime comparison with DPM

6.1.3 Comparison with DDPMine

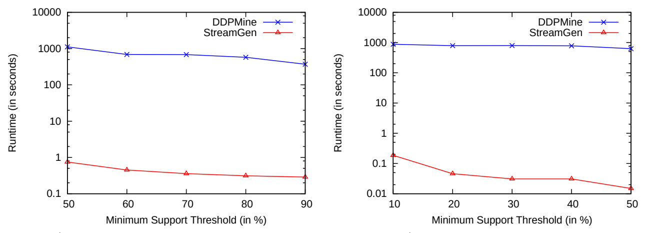
We also compared the efficiency of StreamGen with DDPMine in terms of mining classification rules directly from the datasets. Similar to DPM, we also adapt DDPMine to mine classification rules in stream data setting by running DDPMine on each sliding window having full window size (we denote the DDPMine-based approach by DDPMine-stream). Figure 12 a) provides the comparison results on mushroom dataset with a large sliding window size of $8,000$ for mining classification rules. We could easily find that DDPMine-stream is much slower than StreamGen. Actually, even one round running of DDPMine-stream on one sliding window is much longer than the total time of StreamGen on all sliding windows.

Figure 12 b) shows the results on horse dataset with a sliding window size of 600. We see that DDPMine-stream uses nearly $1,000$ seconds while StreamGen uses less than one second. In fact,

Dataset	StreamGen				DDPMine			
	Accuracy	max. len.	avg. len.	avg. num.	Accuracy	max. len.	avg. len.	avg. num.
breast	**96.708**	3	1.551	23.6	95.28	9	2.448	11.6
adult	**82.146**	3	1.831	13	81.292	14	4.583	7.2
mushroom	**98.918**	3	1.958	9.6	97.184	22	15.592	16.2
hepatitus	**82.006**	4	2.387	15	76.986	8	4.8	5
horse	**81.512**	2	1.389	3.6	81.246	20	4.88	10
pima	74.87	4	1.663	18.4	**75.124**	7	2.435	12.6

Table 4: Classification accuracy and some related statistical information of the mined rules ($sup_{min} = 0.1$).

the total runtime of StreamGen on all sliding windows over the entire dataset is shorter than the runtime of DDPMine-stream on only one sliding window.



$a)\ dataset = mushroom,$ $window\_size = 8,000$

$b)\ dataset = horse,$ $window\_size = 600$

Figure 12: Runtime comparison with DDPMine

6.2 Classification Accuracy Evaluation

Similar to the state-of-the-art classification rule mining algorithm, DDPMine, StreamGen builds a SVM classification model with its features based on those mined association rules. We compared the classification accuracy using the five-fold cross-validation scheme. Table 4 shows the accuracy comparison between StreamGen and DDPMine and some statistical information of the mined rules (e.g., the average pattern length, the maximum pattern length, and the average number of patterns) for datasets breast, adult, mushroom, hepatitus, horse, and pima, with a minimum support of 0.1. During the experiment, DDPMine ran on the entire dataset, while StreamGen ran with a window size equal to the dataset size (That means there is only one sliding window for StreamGen).

We see that StreamGen achieves better accuracy than DDPMine in most cases, which validates that StreamGen is effective in mining classification rules over stream data with high accuracy. We can also observe that the rules mined by StreamGen is much shorter than those mined by DDPMine. We have to concede that the number of rules mined by StreamGen is usually larger than that of DDPMine, which is due to the fact that DDPMine always tries to find an optimum rule for the remaining transactions and thus usually outputs a smaller number of rules in practice. However, the simplicity of the rules is more important since simpler rules can be better understood and explained. Table 5 shows an example of the rules mined by StreamGen and DDPMine in one of the five folds on the mushroom dataset with a support of 0.1. We could easily find that the rules mined by StreamGen are significantly simpler. Note we have tried some other support thresholds (e.g., 0.05) for these datasets, and got similar comparison results.

StreamGen	DDPMine
38	17 39
12 25	5 7 8 11 13 15 16 17 18 19 20 26
13 25	8 17 18
7 67	5 7 9 13 14 15 16 17 18 19 20 40 41 46 53 54
66	2 7 9 11 13 14 15 16 17 18 19 20 21 38 40 44 53 54 76
7 68	2 7 9 11 13 14 15 16 17 18 19 20 28 38 40 44 53 54 76
11 18	2 7 9 11 13 14 15 16 17 18 19 20 32 38 40 53 54 65 76
6 18 37	2 7 9 11 13 14 15 16 17 18 19 20 22 32 38 40 53 54 76
4 53	2 7 9 11 13 14 15 16 17 18 19 20 28 32 38 40 46 53 54 76
	2 7 9 11 13 14 15 16 17 18 19 20 21 32 38 40 45 46 53 54 76
	2 7 9 11 13 14 15 16 17 18 19 20 21 32 34 38 40 46 48 53 54 76

Table 5: One example of the classification rules mined by StreamGen and DDPMine ($dataset = mushroom$, $sup_{min} = 0.1$).

7. CONCLUSIONS

Many previous studies have shown that mining itemset generators is very meaningful from the classification point of view according to the MDL principle. In this paper, we explore a new and challenging problem of mining frequent itemset generators over a data stream sliding window. We devise a novel enumeration tree structure to help maintain the information of the mined generators and the border between generators and non-generators. We also propose some effective optimization techniques and develop the StreamGen algorithm. The comprehensive performance study shows that StreamGen outperforms several state-of-the-art algorithms in terms of efficiency and classification accuracy.

8. ACKNOWLEDGEMENTS

This work was supported in part by National Natural Science Foundation of China under Grant No. 60873171 and 60833003, National Basic Research Program of China under Grant No. 2006CB303103, Basic Research Foundation of Tsinghua National Laboratory for Information Science and Technology (TNList), a HP Labs Innovation Research Program award, a research award from Google, Inc., and the Program for New Century Excellent Talents in University under Grant No. NCET-07-0491, State Education Ministry of China.

9. REFERENCES

[1] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, D.C., 1993. ACM Press.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of 20th International Conference on Very Large Data Bases*, pages 487–499, Santiago de Chile, Chile, 1994. Morgan Kaufmann.

[3] M.-L. Antonie and O. R. Zaïane. Text document categorization by term association. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*, pages 19–26, Maebashi City, Japan, 2002. IEEE Computer Society.

[4] Y. Bastide, N. Pasquier, R. Taouil, G. Stumme, and L. Lakhal. Mining minimal non-redundant association rules using frequent closed itemsets. In *Proceedings of the First International Conference on Computational Logic*, pages 972–986, London, UK.

[5] H. Cheng, X. Yan, J. Han, and C.-W. Hsu. Discriminative frequent pattern analysis for effective classification. In *Proceedings of the 23rd International Conference on Data Engineering*, pages 716–725, Istanbul, Turkey, 2007. IEEE.

[6] H. Cheng, X. Yan, J. Han, and P. S. Yu. Direct discriminative pattern mining for effective classification. In *Proceedings of the 24th International Conference on Data Engineering*, pages 169–178, Cancún, México, 2008. IEEE.

[7] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz. Catch the moment: maintaining closed frequent itemsets over a data stream sliding window. *Knowl. Inf. Syst.*, 10(3):265–294, 2006.

[8] G. Cong, K.-L. Tan, A. K. H. Tung, and X. Xu. Mining top-k covering rule groups for gene expression data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 670–681, Baltimore, Maryland, USA, 2005. ACM.

[9] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *Proceedings of the Fifteen ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 43–52, 1999.

[10] C. Gao, J. Wang, Y. He, and L. Zhou. Efficient mining of frequent sequence generators. In *Proceedings of the 17th International Conference on World Wide Web*, pages 1051–1052, Beijing, China, 2008. ACM.

[11] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Frequent Itemset Mining Implementations, Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations*, Melbourne, Florida, USA, 2003. CEUR-WS.org.

[12] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Min. Knowl. Discov.*, 8(1):53–87, 2004.

[13] N. Jiang and L. Gruenwald. Cfi-stream: mining closed frequent itemsets in data streams. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 592–597, Philadelphia, PA, USA, 2006. ACM.

[14] J. Li, G. Dong, and K. Ramamohanarao. Making use of the most expressive jumping emerging patterns for classification. *Knowl. Inf. Syst.*, 3(2):131–145, 2001.

[15] J. Li, H. Li, L. Wong, J. Pei, and G. Dong. Minimum description length principle: Generators are preferable to closed patterns. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, Boston, Massachusetts, USA, 2006. AAAI Press.

[16] J. Li, G. Liu, and L. Wong. Mining statistically important equivalence classes and delta-discriminative emerging patterns. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 430–439, San Jose, California, USA, 2007.

[17] W. Li, J. Han, and J. Pei. Cmar: Accurate and efficient classification based on multiple class-association rules. In *Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 369–376, San Jose, California, USA, 2001. IEEE Computer Society.

[18] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *Proceedings of the Fourteen ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 80–86, 1998.

[19] G. Liu, H. Lu, W. Lou, and J. X. Yu. On computing, storing and querying frequent patterns. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 607–612, Washington, DC, USA, 2003. ACM.

[20] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proceedings of the 7th International Conference on Database Theory*, pages 398–416, Jerusalem, Israel, 1999. Springer.

[21] J. Pei, J. Han, and R. Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, 2000.

[22] J. R. Quinlan and R. M. Cameron-Jones. Foil: A midterm report. In *Machine Learning: ECML-93, European Conference on Machine Learning*, pages 3–20, Vienna, Austria, 1993. Springer.

[23] A. Veloso, W. M. Jr., M. de Carvalho, B. Pôssas, S. Parthasarathy, and M. J. Zaki. Mining frequent itemsets in evolving databases. In *Proceedings of the 2002 SIAM International Conference on Data Mining*, Arlington, VA, USA, 2002. SIAM.

[24] A. Veloso, W. M. Jr., and M. J. Zaki. Lazy associative classification. In *Proceedings of the 6th IEEE International Conference on Data Mining (ICDM 2006)*, pages 645–654, Hong Kong, China, 2006. IEEE Computer Society.

[25] J. Wang, J. Han, and J. Pei. Closet+: searching for the best strategies for mining frequent closed itemsets. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 236–245, Washington, DC, USA, 2003. ACM.

[26] J. Wang and G. Karypis. On mining instance-centric classification rules. *IEEE Trans. Knowl. Data Eng.*, 18(11):1497–1511, 2006.

[27] L. Xu and K. Xie. An incremental algorithm for mining generators representation. In *PKDD 2005, 9th European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 701–708, Porto, Portugal, 2005. Springer.

[28] J. X. Yu, Z. Chong, H. Lu, and A. Zhou. False positive or false negative: Mining frequent itemsets from high speed transactional data streams. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 204–215, Toronto, Canada, 2004. Morgan Kaufmann.

[29] M. J. Zaki and C.-J. Hsiao. Charm: An efficient algorithm for closed itemset mining. In *Proceedings of the 2002 SIAM International Conference on Data Mining*, Arlington, VA, USA, 2002. SIAM.