Privacy-Preserving Classification for Data Streams

Yabo Xu, Ke Wang
School of Computing Science
Simon Fraser University
{yxu,wangk}@cs.sfu.ca

Ada Wai-Chee Fu
Department of Computer Science
The Chinese University of Hong Kong
adafu@cse.cuhk.edu.hk

Rong She, Jian Pei
School of Computing Science
Simon Fraser University
{rshe, jpei}@cs.sfu.ca

ABSTRACT
In a wide range of applications, multiple data streams need to be examined together in order to discover trends or patterns existing across several data streams. One common practice is to redirect all data streams into a central place for joint analysis. This "centralized" practice is challenged by the fact that data streams often are private in that they come from different owners. In this paper, we focus on the problem of building a classifier in this context and assume that classification evolves as the current window of streams slides forward. This problem faces two major challenges. First, the many-to-many join relationship of streams will blow up the already fast arrival rate of data streams. Second, the privacy requirement implies that data exchange among owners should be minimal. These considerations rule out all classification methods that require producing the join in the current window. We show that Naïve Bayesian Classification (NBC) presents a unique opportunity to address this problem. Our main contribution is to adopt NBC to solve the classification problem for private data streams.

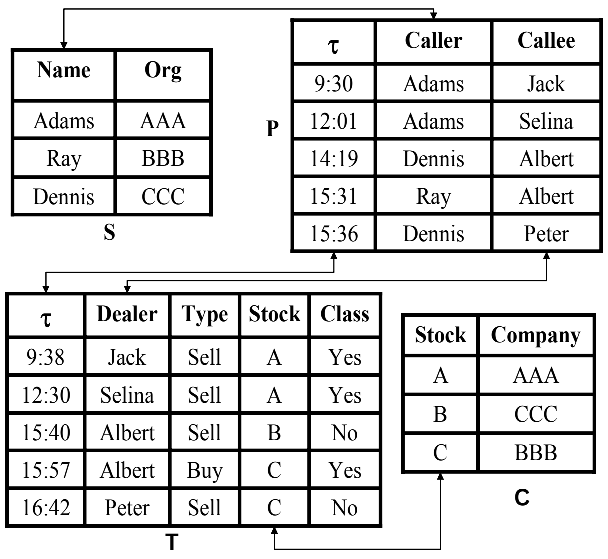
1. INTRODUCTION
With today's information explosion, data not only are stored in large amount but also grow rapidly over time. *Data streams* are such examples, including internet traffic streams, stock trading streams, and telephone call streams. Data streams are characterized as being unbounded, continuously arriving at a high rate, and being scanned once [2]. To benefit from the information and knowledge contained in data streams, often *several* related data streams need to be examined together to discover trends or patterns that exist across different data streams. For example, stock streams and news streams are related, traffic report streams and car-accident streams are related, sensor readings of different types are related. In this paper, we focus on building classification models from such data. Our insight is that classification patterns often are jointly determined by the co-occurrence of certain conditions in several related streams. We illustrate this point by a simplified example.

1.1 Motivating Example
One application involving data streams is to monitor financial or trading transactions for suspicious behaviors [38]. In stock markets, "favorable trading" refers to stock transactions that are favorable to the engaging party, i.e., selling before a stock plunges or buying before a stock goes up. In order to build classification models to identify "favorable trading", the stock trading stream that records all trading transactions must be examined. However, stock transactions are *not* isolated or independent events; they are related to other data streams, e.g., phone calls between dealers and managers/staffs of public companies. Thus it is necessary to consider related data streams together. For example, a classification algorithm may need to look at the following related data:

Trading stream:	**T** (τ, Dealer, Type, Stock, Class)
Phone call stream:	**P** (τ, Caller, Callee)
Company table:	**C** (Company, Stock)
Person table:	**S** (Name, Org)

where τ is the timestamp, "Type" is either "sell" or "buy", "Class" ("yes"/"no") refers to the class label of being favorable trading or not. To compute the training set, a SQL query can be used to extract information from the above data as follows:

```
SELECT  *
FROM    P, S, T, C
WHERE   S.Name=P.Caller AND P.Callee=T.Dealer
        AND P.τ<T.τ AND T.Stock=C.Stock
```



S

Name	Org
Adams	AAA
Ray	BBB
Dennis	CCC

P

τ	Caller	Callee
9:30	Adams	Jack
12:01	Adams	Selina
14:19	Dennis	Albert
15:31	Ray	Albert
15:36	Dennis	Peter

T

τ	Dealer	Type	Stock	Class
9:38	Jack	Sell	A	Yes
12:30	Selina	Sell	A	Yes
15:40	Albert	Sell	B	No
15:57	Albert	Buy	C	Yes
16:42	Peter	Sell	C	No

C

Stock	Company
A	AAA
B	CCC
C	BBB

Table 1. Related streams / tables

Class	Caller	Org	Callee	Dealer	Type	Stock	Company
Yes	Adams	AAA	Jack	Jack	Sell	A	AAA
Yes	Adams	AAA	Selina	Selina	Sell	A	AAA
Yes	Ray	BBB	Albert	Albert	Sell	B	CCC
Yes	Ray	BBB	Albert	Albert	Buy	C	BBB
No	Dennis	CCC	Albert	Albert	Sell	B	CCC
No	Dennis	CCC	Albert	Albert	Buy	C	BBB
No	Dennis	CCC	Peter	Peter	Sell	C	BBB

Table 2. The join stream

Essentially, this query performs a join on the related data and each joined tuple represents a connection between a phone call and the trading ensued from this call. The join result is then used to train the classifier.

Table 1 shows a snapshot of data. The join relationship is indicated by the arrows connecting the join attributes. Note that the join between P and T is "many-to-many". For example, "Albert" was called twice and traded twice, generating four tuples in the join stream in Table 2 (The timestamps for each join record are ignored because of the space.), the rule "Org=Company → Class=Yes" holds in 3 out of 4 tuples that have "Org=Company", i.e., with 75% confidence. This suggests that after getting a call, the trading on the caller's company stock tends to be more favorable.

This example illustrates that classification of certain behaviors (i.e., favorable trading) depends on information contained in several correlated streams and examining such streams together likely produces more accurate classifiers than examining any single input stream alone. The join is a common operation to combine several streams into a single stream and the training data for classification is defined by this "join stream". Moreover, classification rules evolve as data streams evolve. New favorable trading rules may emerge as a reaction to evade from being identified by existing rules. In order to capture this change, the classifier needs to adapt quickly to the changed data distribution. A solution to this problem faces two major challenges.

- **Privacy preservation**. In the above example, since trade and phone call streams involves trading secrets and individual's privacy, apparently neither the trading company nor the phone service company is willing to disclose their local sensitive data. The common approach of redirecting all streams into a central place immediately violates this privacy constraint. In the literature, privacy-preserving data mining and stream data mining have been studied separately. The traditional privacy-preserving data mining techniques focus on static data and are not applicable to data streams with unbounded data size and continuous arrival of new data. On the other hand, most prior work on stream data mining assumes either a single stream or several streams but no privacy issue, and focuses on the processing speed of stream data. More details in Section 2.
- **Blow-up of the join stream**. As input streams arrive in a fast pace, the classifier must evolve quickly when new structures emerge and old ones are out-of-date. However, the join of multiple input streams is an expensive operation, in fact, much slower than the arrival rate of input streams. Furthermore, the "many-to-many" join relationship, as shown in Table 1, could generate the result join stream that is much larger than input streams. Any method that explicitly generates the join stream will suffer from thid blow-up of data arrival rates and is unlikely to be able to keep pace with the incoming source streams.

1.2 Contributions and Paper Outline

We consider several private data streams owned by different sites. One data stream, called *target stream*, contains class labels. The current window of the training data is defined by the join of input streams in their current window. Such joins are called *sliding-window join* [2] and the join result defines a new stream called "*join stream*". The specification of window can be either tuple-based or time-based [2]. As the window of input streams slides forward, so does the window of the join stream and the classifier must be updated to adapt to the change of window. In practice, only some portion of the data is labeled whereas the remaining is not. For each window, the unlabeled portion will be classified by the classifier built in the previous window, and at the same time, the labeled portion will be used to train the classifier in the current window [31].

Due to the privacy requirement and blow-up of join, however, the join stream cannot be generated explicitly. Hence, the problem we study is to build and update the classifier based on the never-generated join stream, given several private input streams. The construction and update of the classifier must not reveal private information to other sites. This problem is referred to as the *secure join stream classification* (Secure-JSC) hereinafter. Existing classification methods [19][4][31] cannot be applied to the Secure-JSC problem because they deal with a single stream and requires the join stream to be explicitly given.

Our insight is that the independence assumption of Naïve Bayesian Classifier (NBC) [12] provides a unique opportunity to address the requirements for Secure-JSC: for a given class label, variables are assumed to be independent of each other. Research shows that NBC is reliable even when this assumption is violated [11][17][16]. The reason for this reliability is that the most likely class label predicted by NBC is typically correct though the estimated probability may be distorted by the independence assumption. In other words, the top ranked class label often is correct though the estimated probability for ranking class labels was distorted. This reliability has been echoed by the popularity and success of NBC in both research works and practical applications. For most data stream applications, some degree of inaccuracy is tolerable, especially so because the data arrive very fast and there is only the time to scan the data once.

To adopt NBC to Scure-JSC problem, however, we must compute the information required by NBC on the *join stream* without generating the join and exchanging private information across sites. Our insight is that this information can be obtained by computing some "blow-up summary" from examining each input stream and exchanging this summary with other sites. The "blow-up summary" can be computed by examining each input tuple in the current window twice, independent of the number of tuples it joins in other streams. The benefit of this approach is twofold: eliminate the need of collecting private data streams in a central place and avoid the expensive join. Though we consider NBC, the idea of "blow-up summary" is applicable to other classification algorithms that require similar statistics such as decision trees.

The rest of this paper is organized as follows. In Section 2, we review related works. In Section 3, we define the problem and discuss core concepts of NBC. In Section 4, we present our algorithm. We evaluate our method in Section 5. Section 6 concludes the paper.

2. RELATED WORKS

Privacy preserving data mining was first introduced in [25] and [26]. These works opened up a rapidly growing area and various privacy preservation techniques have emerged since then. Most works on privacy preserving data mining assume static data. On the other hand, there is a large body of works on stream data management and mining. But this body of works does not deal with privacy issues. The novelty of our work lies in addressing privacy preservation, data streams and the training data defined by a general join of several streams. Below, we focus on related works which address one ore more of these aspects.

In data stream management [2], sliding-window join is proposed to answer queries involving the join of multiple data streams, such as the join size, sum [1] [9], join-distinct [14]. Their focus was on how to compute these statistics of the join under resource constraints and techniques such as sampling [6] or load-shedding [5] [18] are used to reduce the cost of join. These works assume either a single stream or multiple streams but no privacy issue. In the Secure-JSC problem, as we explained in the previous section, it is prohibited to first compute the join of multiple streams and then build the classifier. Thus these techniques cannot be applied.

Most stream mining algorithms consider a single stream and simple statistics such as average and standard deviation. Classification on data streams was considered in [10][13][19][4][31]. Other mining problems that involve multiple streams are clustering [15][3], correlation analysis [20], sequential patterns [7]. None of these works consider the privacy issue. Neither do they involve a general join among streams; thus, they do not deal with the blow-up of data arrival rates caused by a many-to-many join.

[22] presents a secure construction of decision tree classifiers from vertically partitioned data, where the join is given by the one-to-one relationship implied by the common key identifier for all partitions. This is not applicable to the general many-to-many join relationship. Recently, [21] proposed a secure construction for decision tree classifiers over distributed tables with the general many-to-many join relationship. Both works consider static data, not stream data.

There are only a few studies that cover both data streams and privacy preservation. [36][37] focus on the problem of private search over data streams. Their goal is to protect the privacy of the query over data stream, not the data stream itself. A more related work is [23]. It preserves the privacy of data streams by adding randomized noises. No join relationship is involved among streams. This approach cannot be applied to the Secure-JSC problem since the data obfuscation does not preserve the join relationship among streams. The condensation approach in [33] could be applied to data streams because of its support for incremental update. Since anonymized records are randomly generated in a way to preserve aggregated statistics, it is not clear that this method could preserve the join relationship if applied to multiple data streams. All these works do not consider the classification problem.

3. PROBLEM STATEMENT

3.1 Secure Join Stream Classification

Consider n data streams S_1, …, S_n, distributed among n sites. *Secure Join stream classification* refers to the problem where a classifier needs to be built such that (1) the training instances are defined by a sliding-window join over all data streams; (2) no site learns private information about other data streams. The sliding-window join over S_1, …, S_n is specified by a join condition, a window specification and window update specification [2][32].

In this paper, we consider a join condition in the form of a conjunction of equality predicates $S_i.A=S_j.B$ ($i \neq j$), where each of $S_i.A$ and $S_j.B$, called *join attributes,* represents one or more attributes from S_i and S_j. Since $S_i.A$ and $S_j.B$ are allowed to contain more than one attribute, we need to consider at most one predicate $S_i.A=S_j.B$ between each stream pair S_i and S_j. In the *join graph,* there is an edge between S_i and S_j if there is a predicate $S_i.A=S_j.B$ in the join condition. We consider join conditions for which the join graph is connected and contains no cycle. Many joins in practice are in fact acyclic, such as chain joins and star joins over the star/snowflake schemas [34].

The window and update specification can be time-based or tuple-based. Our method only depends on the set of tuples in the current window, not on how the window is specified and updated. The term "window" refers to the collection of current windows of all input streams. One of $S_1,\ldots,S_n$, called *target stream*, contains the class column. The task is to build a classifier each time the window updates. This means that the classifier must be rebuilt whenever the window on any input stream slides forward. The speed of fastest-sliding window determines the rate of classifier updates.

In the current window, the training set is the set of tuples defined by the sliding-window join. Importantly, the training set is *not* explicitly given, rather, is specified by the input streams and the sliding-window join. Some tuples in the input streams do not contribute any tuple in the join. Such tuples are *dangling*. We do not assume that dangling tuples are removed beforehand; in fact, the removal of dangling tuples is not straightforward due to the privacy requirement.

Privacy Model. All sites are assumed to be honest, curious, but not malicious [35]. Intuitively, this means that a site may collect intermediate information received from other sites, but will follow the specified computation as expected. Our privacy model can be described by three types of attributes in each window:

- *Non-private class column*: the class column can be revealed to all sites. This assumption was made previously in [22]. When all sites collaborate to build a classifier, we assume these sites are willing to share the information on class labels
- *Semi-privat*e join attributes: for a join predicate $S_i.A=S_j.B$, the join attributes $S_i.A$ and $S_j.B$, are semi-private in that the sites of S_i and S_j are willing to share their join values that they both have, i.e., $S_i.A\cap S_j.B$, but not any other join values, i.e., $(S_i.A\cup S_j.B)-(S_i.A\cap S_j.B)$. This model was adopted in the literature for secure join and intersection in [24] .
- *Private non-join attributes*: the values of all non-join attributes must not be revealed to any other sites.

In short, any join values known to the joining sites are not private, but everything else is. We shall use this privacy model to define our notion of privacy-preserving.

3.2 Naïve Bayesian Classifiers

Consider a single table T ($X_1,\ldots, X_n$, Class). "Class" denotes the class column whose domain is a collection of class labels $\{C_1,\ldots, C_m\}$. X_i is a categorical variable. To classify a tuple $x=(x_1,\ldots, x_n)$, the Naïve Bayesian Classifier (NBC) assigns x to the class C_i that maximizes the conditional class probability $P(C_i|x)$ based on the following maximum a posteriori (MAP) hypothesis:

$$\underset{C_i\in Class}{\operatorname{argmax}} P(C_i\mid x)=\underset{C_i\in Class}{\operatorname{argmax}} P(x\mid C_i)P(C_i)$$

where $P(C_i)$ is the class probability and $P(x|C_i)$ is the conditional probability of x given the class label C_i. Under the independence assumption that variables $X_1, \ldots, X_n$ are independent given the class label, NBC estimates $P(x|C_i)$ by

$$P(x\mid C_i)=\prod_{j=1..n}P(x_j\mid C_i)\cdot$$

Once $P(x_j|C_i)$ and $P(C_i)$ are collected from the training data, NBC is able to assign a class label to a new tuple x. NBC requires the variables X_i to be categorical (having a small number of distinct values). Continuous attributes can be first discretized (such as equi-width or equi-depth binning) into a small number of intervals before applying NBC.

To compute $P(x_j|C_i)$ and $P(C_i)$, we only need to compute the *class count matrix* of the form (x_k, $<N_1,\ldots, N_m>$) for each distinct value x_k of X_j, where N_j ($1\leq j\leq m$) is the number of tuples that has the value x_k and the class label C_j. This data structure has a size proportional to the number of distinct values in X_j.

The above discussion assumes a single table T. For Secure-JSC, T will be the join result of the input streams $S_1, \ldots, S_n$ in the current window. Generating T would violate the privacy constraint. The challenge is to compute $P(x_j|C_i)$ and $P(C_i)$ on the join T without generating T. In the next section, we present such a method.

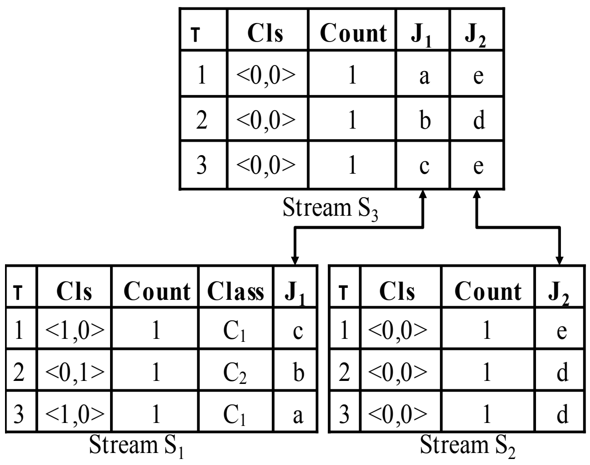
4. OUR APPROACH

We assume that the current window of each input stream can fit in the local memory. The join relationship among streams forms an acyclic join graph, which is a rooted tree. Any stream may be regarded as the root. As our method involves propagation of information along the edges of the tree, we call this tree *propagation tree*.

Let us consider the site for an input stream S_i. Instead of generating the join stream, the site maintains an entry of (*Cls*, *Count*) for each tuple *t* in the current window of S_i. *Cls* is a *class vector* in the form of $<N_1,\ldots, N_m>$ where *m* is the number of classes and N_i records the number of occurrences of *t* associated with the class label C_i in the never-generated join stream. *Count* is the number of occurrences of *t* in the join stream. Intuitively, the entry (*Cls*, *Count*) for *t* stores all information about *t* in the current window of the join stream. Thus, instead of keeping every join tuple involving t, we keep *t* only once and store its number of occurrences and class labels in those occurrences. The size of this data structure is proportional to the window size of S_i. Importantly, having *Cls* for each tuple *t* in the current window, the site of S_i is able to compute $P(x_j|C_i)$ and $P(C_i)$ for the all values x_j in the current window. The challenge is computing *Cls* without performing the join.

To compute the class vectors *Cls,* we propagate the "blow-up effect" of join. The propagation proceeds in two phases. In the phase of *bottom-up propagation*, *Cls* and *Count* are propagated from the leaf nodes to the root. The propagation along an edge blows up *Cls* and *Count* according to the join condition on the edge. The detail will be presented shortly. On reaching the root, the *Cls* for the root reflects the join of all input streams. Next, in the phase of *top-down propagation*, we propagate *Cls* from the root to all leaf nodes. When reaching all leaf nodes, *Cls* in each stream have reflected the join effect of all streams. The algorithm is distributed in that each node (site) in the tree performs the propagation as described; there is no central place to collect all data. This approach circumvents the computation of the sliding-window join, thus addresses both the privacy and efficiency requirements.

Now we explain the propagation at each site in details. First, we extend arithmetic operations to class vectors *Cls*: given an operator "$\circ$" and two *Cls*'s $V_1=<a_1,\ldots,a_m>$ and $V_2=<b_1,\ldots,b_m>$, $V_1\circ V_2=<a_1\circ b_1,\ldots,a_m\circ b_m>$. For example, $<4,3>/<2,3>=<2,1>$.



τ	Cls	Count	J$_1$	J$_2$
1	<0,0>	1	a	e
2	<0,0>	1	b	d
3	<0,0>	1	c	e

Stream S$_3$

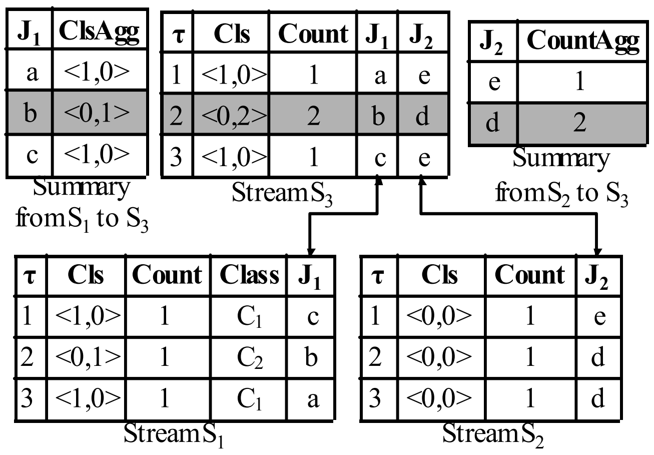
τ	Cls	Count	Class	J$_1$
1	<1,0>	1	C$_1$	c
2	<0,1>	1	C$_2$	b
3	<1,0>	1	C$_1$	a

Stream S$_1$

τ	Cls	Count	J$_2$
1	<0,0>	1	e
2	<0,0>	1	d
3	<0,0>	1	d

Stream S$_2$

Figure 1. Example with 3 streams at initialization



J$_1$	ClsAgg
a	<1,0>
b	<0,1>
c	<1,0>

Summary from S$_1$ to S$_3$

τ	Cls	Count	J$_1$	J$_2$
1	<1,0>	1	a	e
2	<0,2>	2	b	d
3	<1,0>	1	c	e

Stream S$_3$

J$_2$	CountAgg
e	1
d	2

Summary from S$_2$ to S$_3$

τ	Cls	Count	Class	J$_1$
1	<1,0>	1	C$_1$	c
2	<0,1>	1	C$_2$	b
3	<1,0>	1	C$_1$	a

Stream S$_1$

τ	Cls	Count	J$_2$
1	<0,0>	1	e
2	<0,0>	1	d
3	<0,0>	1	d

Stream S$_2$

Figure 2. After bottom-up propagations

4.1 Initialization

Initially, for each tuple in the target stream, its *Cls*, <N_1, …, N_m>, is determined as follows: N_i=1 if the class label is C_i or otherwise N_j=0. *Count* is initialized to 1. For any tuple in any other stream, its *Cls* is initialized to all zeros <0,…,0> and *Count* is 1. This initialization does not require a separate scan of streams and can be combined with the bottom-up propagation discussed in the following subsection.

Example 1. Consider an example with 3 streams with initial *Cls* and *Count* shown in Figure 1. The join relationships are specified by the arrows: S$_1$ and S$_3$ join on J$_1$, and S$_2$ and S$_3$ join on J$_2$. S$_1$ is the target stream containing two classes. S$_3$ is the root of the propagation tree. The root can be arbitrarily selected. We will show later that choosing the input stream with largest window size as the root can optimize the cost of scan of input streams.

4.2 Bottom-Up Propagation

This is the phase where the information of *Cls* and *Count* are propagated from leaf nodes to the root in a bottom-up order. Consider a parent node S$_P$ and a child node S$_C$ with the join predicate S$_P$.J$_1$= S$_C$.J$_2$. The propagation from a child to the parent is based on the following observation.

Observation 1: Given a tuple *t* in S$_P$, if *t* joins with *k* tuples in S$_C$, *t* will occur *k* times in the join between S$_P$ and S$_C$. These occurrences can be represented by blowing up *Cls* and *Count* of *t* using the aggregated *Cls* and *Count* of the *k* joining tuples in S$_C$. And if S$_P$ has *n* child nodes (*n*>1), the *Cls* and *Count* of *t* in S$_P$ will be blown up by all children to reflect the join with all children streams.

To explain Observation 1 precisely, we define the *blow-up summary* from S$_C$ to S$_P$ as the set {(*v*, *ClsAgg*, *CountAgg*)}. *v* is a distinct join value in S$_C$, *ClsAgg*=∑Cls and *CountAgg*=∑*Count*, where ∑ is over all tuples in Sc containing the value v. Since the target stream can be anywhere in the tree, there are two cases in the bottom-up propagation from children to a parent node S$_P$:

- If the target stream is not in S$_P$'s subtree, we blow up only *Count* at S$_P$ since *ClsAgg* is always zero for all child nodes of Sp (recall Cls is initialized to all-zero for a non-target stream);
- If the target stream is in S$_P$'s subtree, exactly one of the child of S$_P$ has non-zero *ClsAgg* and we blow up both *Cls* and *Count* at S$_P$.

The following lemma gives the computation for blow-up following the above observation and discussion .

Lemma 1. Assume that a parent node S$_P$ has *n* child nodes. For each tuple *t* in S$_P$ with the join values $v_1,\ldots,v_n$, where v_i is the join value between S$_P$ and the *i*th child, let (v_i, $ClsAgg_i$, $CountAgg_i$) denote the blow-up summary from *i*th child. Then

$$t.Count = \prod_{j=1..n} (CountAgg_j).$$

If some $ClsAgg_i$ ($1 \le i \le n$) is non-zero,

$$t.Cls = (ClsAgg_i) * \prod_{j=1..n, j \ne i} (CountAgg_j) \qquad \blacksquare$$

Based on this lemma, to compute *Count* and *Cls* at S$_P$, each child node S$_C$ propagates its blow-up summary to the parent S$_P$. After receiving blow-up summaries from all child noes, S$_P$ scans its tuples once and updates *Count* and *Cls* of each tuple t as in Lemma 1. In addition, S$_P$ creates the blow-up summary from S$_P$ to its own parent (if any) in the same scan.

Example 2. The bottom-up propagation for Example 1 is shown in Figure 2. S$_1$ and S$_2$ are scanned (locally) to produce blow-up summaries to propagate to S$_3$. On receiving the summaries, S$_3$ blows up *Cls* and *Count* of its tuples. For example, consider the tuple *t* in S$_3$ as gray scaled in Figure 2 (with J$_1$=b, J$_2$=d). *t* has two corresponding summary entries: (b,<0,1>,1) from S$_1$ and (d,<0,0>,2) from S$_2$. *t.Count*=1*2=2, *t.Cls*=<0,1>*2=<0,2>. These results indicate that *t* occurs in the join twice, both having the class label C$_2$, which is exactly the same information as in the join stream.

4.3 Top-Down Propagation

At the end of bottom-up propagation, *Cls* in the root stream reflects the join of all streams. However, *Cls* in other streams has not reflected the joins performed at their ancestors. Thus we need to propagate in the top-down fashion to push the correct join information to all non-root streams. The propagation is based on the following observation.

Observation 2: For a parent node S_P and a child node S_C, if a tuple *t* in S_C joins with some tuple in S_P that has the join value *v*, so do *all* tuples in S_C that have this join value *v*. We can view all such tuples as an "*equivalence class*" on the join value *v* in S_C, denoted as $S_C[v]$. Similarly, $S_P[v]$ contains all tuples in S_P that have the join value *v*. *Cls* of the $S_C[v]$ tuples must be rescaled to reflect all joins not reflected so far at S_C. The rescaling must satisfy the following properties: (1) the relative share of any tuple in $S_C[v]$ remains unchanged because every tuple in $S_C[v]$ will join every tuple in $S_P[v]$, (2) the aggregated $\sum Cls$ in $S_C[v]$ after rescaling is the same as the aggregated $\sum Cls$ in $S_P[v]$.

To perform the top-down propagation, we define the *rescaling summary* from S_P to S_C as the set {(*v*, *ClsAgg*)}, where *v* is a join value in S_P and *ClsAgg* is the aggregated class vector of all $S_P[v]$ tuples.

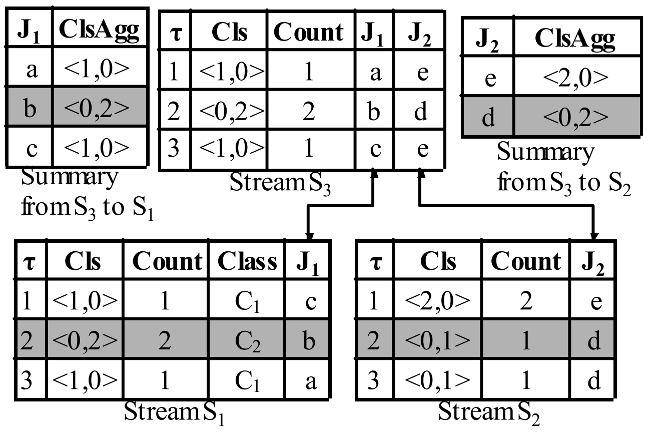
Lemma 2. Let *t* be a tuple in $S_C[v]$ and let (*v*, *ClsAgg*) be a rescaling summary entry from S_P. *t.Cls* is rescaled as follows:

$$t.Cls = ClsAgg * (t.Count / S_C[v].CountAgg)$$

where $S_C[v].CountAgg$ is the aggregated $\sum Count$ over all $S_C[v]$ tuples. ■

The ratio $t.Count/S_C[v].CountAgg$ represents *t*'s share in $S_C[v]$. Based on this lemma, to compute *Cls* at S_C, the parent node S_P propagates its rescaling summary to S_C. On receiving the rescaling summary from S_P, *Cls* in S_C are updated as in Lemma 2. In the same scan, the rescaling summary from S_C to its own children (if any) is computed.

Example 3. The top-down propagation is shown in Figure 3. At the root S_3, the rescaling summaries to S_1 and S_2 are generated while scanning S_3 in the bottom-up propagation. On receiving these summaries, S_1 and S_2 rescale their *Cls*. For example, for the tuple *t* in S_1 as gray scaled in Figure 3, *t.Cls*=<0,1> is rescaled to <0,2>*(1/1)=<0,2>, where (b,<0,2>) is the summary entry corresponding to b, and (1/1) is the share of *t* in its own equivalence class for J_1=b. The result captures exactly the same information about *t* as in the join stream: *t* occurs twice having the class label C_2.



J_1	ClsAgg
a	<1,0>
b	<0,2>
c	<1,0>

Summary from S_3 to S_1

τ	Cls	Count	J_1	J_2
1	<1,0>	1	a	e
2	<0,2>	2	b	d
3	<1,0>	1	c	e

Stream S_3

J_2	ClsAgg
e	<2,0>
d	<0,2>

Summary from S_3 to S_2

τ	Cls	Count	Class	J_1
1	<1,0>	1	C_1	c
2	<0,2>	2	C_2	b
3	<1,0>	1	C_1	a

Stream S_1

τ	Cls	Count	J_2
1	<2,0>	2	e
2	<0,1>	1	d
3	<0,1>	1	d

Stream S_2

Figure 3. After top-down propagations

4.4 Using NBC

We now consider classifying a new instance t=<t_1,…, t_n>, where t_j is the sub-record from S_j. At each site j for S_j, let t_j=<x_1,…x_m>. The site j computes $P(t_j|C_i) = \prod P(x_k|C_i)$ for k=1,…,m, and sends $P(t_j|C_i)$ to a coordinator, which could be any of the participating sites or a third party. After receiving this information from all sites, the coordinator computes $P(t|C_i) = \prod P(t_j|C_i) \times P(C_i)$ for j=1,…,n. The class label C_i that yields the maximum $P(t|C_i)$ is assigned to *t*. $P(C_i)$ is available to every participating site. No private information, as per our privacy model, is revealed by sending $P(t_j|C_i)$ to the coordinator because $P(t_j|C_i)$ is just a numerical value. If an attribute value x_i in a new instance *t* is not found in the training data, this value is simply ignored in the posterior computation.

4.5 Algorithm Analysis

Below, we analyze the algorithm wrt privacy and scalability.

Privacy. In the bottom-up and top-down propagation, only summaries are passed between parent/child pairs. For non-join attributes, no site transmits their values in any form to other sites. For the join attributes, consider a parent node S_P and a child node S_C with the join predicate $S_P.J_1 = S_C.J_2$. The blow-up summary from S_C to S_P contains entries of the form (*v*, *ClsAgg*, *CountAgg*), where *v* is a join value in $S_C.J_2$ and *ClsAgg*/*CountAgg* contains the class/count information. Since *ClsAgg* and *CountAgg* are the aggregate-level information and the class column is non-private, *ClsAgg*/*CountAgg* does not pose a problem. $S_P.J_1$ and $S_C.J_2$ are semi-private, thus *v* can be exchanged between S_P and S_C if $v \in S_P.J_1 \cap S_C.J_2$. This can be ensured by first performing the secure intersection [24] to get $S_P.J_1 \cap S_C.J_2$. Then the blow-up summary from S_C to S_P needs to contain only entries for the join values in the intersection. As for the rescaling summary from S_P to S_C, no secure intersection is needed because all dangling tuples are removed at the end of bottom-up propagation.

Privacy Claim. (1) No private attribute values are transmitted out of its owner site. (2) Semi-private attribute values are transmitted between two joining sites only if they are shared by both sites.■

Scalability. In the bottom-up and top-down propagation, one summary is passed between each parent/child pair and each stream (window) is scanned once. At any time, only the summaries for the edges being examined are kept in memory. The size of a summary is proportional to the number of distinct join values, not the number of join tuples. A summary lookup operation takes a constant time in an array or hash table implementation. The whole propagation is linear in the window size, in fact, scans each input tuple in the current window twice, once at the bottom-up propagation phase and once at the top-down propagation process. Thus, the computation at each window is independent of the join size. This property is important because the join size can be arbitrarily large compared with the window size, due to the many-to-many join relationships. An additional cost is the secure intersection, which is performed during the bottom-up propagation as discussed above. This cost is log linear in the number of distinct join values [24], not the number of tuples in the window.

Scalability Claim. For each window, the cost of rebuilding NBC is proportional to the window size, not the join size. More precisely, each tuple in the window is scanned twice (in memory).

The two scans of the root stream, one in the bottom-up propagation phase and one in the top-down propagation process, can be combined into one. In this case, choosing the input stream of the largest window size (i.e., the most number of tuples) as the root will minimize the cost of stream scans.

5. EMPIRICAL STUDIES

Our approach aims at two goals, namely, privacy preservation and fast processing of join stream classification. The privacy goal is delivered by limiting the information exchanged among sites, as claimed in Section 4. Therefore, in this section we focus on the performance goal. We would like to answer two questions: (1) whether the formulation of Secure-JSC defines a better training space compared with a single stream alone; (2) whether our algorithm scales up to handle high-speed data streams.

We denote our algorithm as *NB_Join*, as it builds a NBC classifier whose training set is defined on the join of multiple streams. We compared it with following alternatives:

- *NB_Target*: NBC based on the target stream alone. In this case, all non-target streams are ignored.

- *DT_Join*: the decision tree classifier (C4.5) on the join stream. To build the decision tree, the join stream is first computed by actually joining the input streams. Note that his approach does not meet the privacy requirement.

- *DT_Target*: the decision tree classifier on the target stream alone.

For each window, we train the classifier using the first 80% of stream tuples within this window and evaluate the classifier using the remaining 20% of stream tuples in the same window. The testing data are generated by the join of the testing samples from all streams.

We measure performance by "time per input tuple", i.e., time spent on each window divided by the number of input tuples in the window. The "input tuples" refers to the tuples in the input streams, not the join stream. This measure gives an idea about the data arrival rate that an algorithm is able to handle. For *DT_Join*, because it has to generate the join stream before building the classifier, we measure the join time only and ignore the classifier construction time since the join time dominates. Most of sliding-window join algorithms in literature are not suitable for generating the join stream for *DT_Join* because they focus on fast computing special aggregates [9][14], or producing approximate join results [18] under resource constraints; not the *exact* join result. We implemented the nested loop join algorithm. This choice should not have a major effect because all tuples in the current window are in memory. All programs were coded in C++ and run on a PC with 2GHz CPU, 512M memory and Windows XP.

5.1 Real-life Datasets

For experiments on real-life dataset, we obtained UK road accident data from the UK data archive[1]. It contains information about accidents, vehicles and casualties, in order to monitor road safety and determine policies to reduce the road accident casualty toll. There are three tables: "Accident", "Vehicle" and "Casualty". The characteristics of year-2001 data are shown in Figure 4 where arrows indicate join relationships: each accident involves one or more vehicles; each vehicle has zero or more casualties. Each table can be regarded as a stream that is timestamped by "date of accident". On average, about 600 "Accident" tuples, 700 "Vehicle" tuples and 850 "Casualty" tuples are added every day. The join stream is specified by the equalities between all common attributes among the three input streams. "Casualty" is the target stream with two casualty classes --- class 1: "fatal/serious" (13% of all tuples) and class 2: "slight" (87% of tuples).

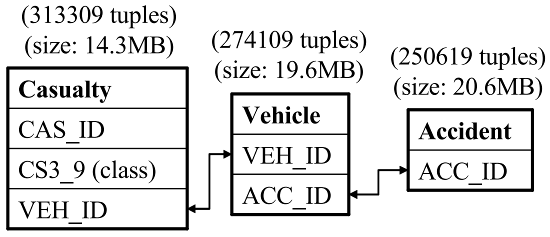


Figure 4. UK road accident data (2001)

5.1.1 Classification Accuracy

Figure 5 shows the accuracy of all classifiers being compared. For all methods, the window size is the same and ranges from 10 to 50 days with no window overlapping.

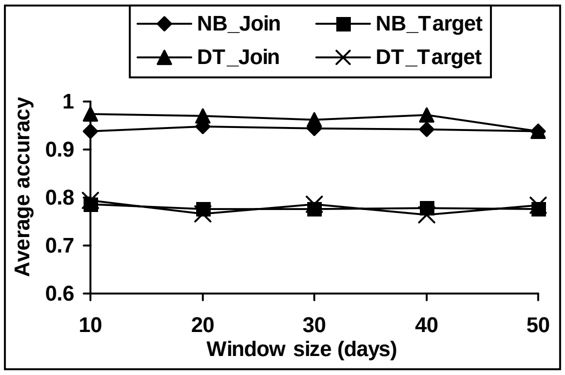


Figure 5. Classifier accuracy

It is apparent that classifiers built on multiple streams are much more accurate. This result confirms that examining correlated streams is advantageous compared with building the classifier on a single stream. In fact, the accuracy obtained by examining the target stream alone is only about 80%, even lower than that obtained by a naïve classifier which simply classifies every tuple as belonging to class 2, since 87% of tuples belong to this class.

On the other hand, the results also show that, with the same training set, naïve Bayesian classifier has a performance comparable to that of the decision tree. Keep in mind that our method *NB_Join* runs directly on the input streams, while the decision tree is built on the join stream. The latter does not meet

[1] http://www.data-archive.ac.uk/

the privacy requirement and has a high join cost. We will examine the efficiency of these two methods in the next set of experiments.

5.1.2 Time per input tuple

Figure 6 compares the time per input tuple. For example, at the window size of 20 days, the join takes about 9.83 seconds whereas *NB_Join* takes only about 0.3 seconds. Therefore, the join time per input tuple is $9.83*10^6/43{,}900=224$ microseconds, where 43,900 is the total number of tuples that arrived in the 20-day window. In contrast, *NB_Join* takes only $0.3*10^6/43{,}900=6.8$ microseconds per input tuple. This means that any method that requires computing the join will be at least 33 times slower than *NB_Join*. As the window size increases, the join time increases quickly due to the increased join cardinality in a larger window; whereas the time per input tuple for *NB_Join* is almost constant. In other words, our approach is linear in the window size, independent of the join stream size. This property makes our approach particularly suitable for multiple correlated streams.

Therefore, though both *NB_Join* and *DT_Join* classifiers exhibit a similar classification accuracy, *NB_Join* is much more efficient than *DT_Join*.

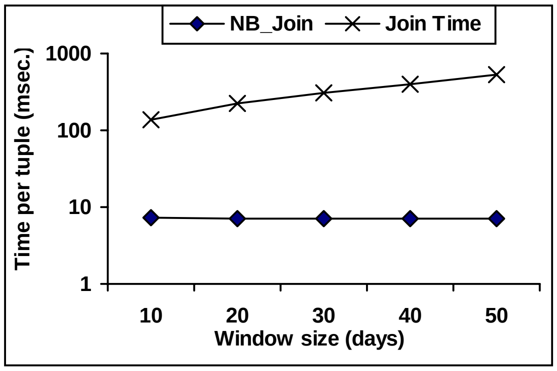


Figure 6. Time per input tuple

5.2 Synthetic Datasets

To further verify our claims, we also conducted experiments on synthetic datasets with various data characteristics. Similar to the experiments on real-life datasets, we want to examine whether the correlation of multiple streams yields benefits for classification under different data characteristics. We also want to evaluate if *NB_Join* can deal with streams with high data arrival rates. As we are not aware of existing data generators to evaluate classification spanning correlated streams, we designed our own data generator.

5.2.1 The Data Generator

We consider the chain join of k streams $S_1, \ldots, S_k$, where S_1 is the target stream. An adjacent pair S_i and S_{i+1} have one join predicate and a non-adjacent pair have no join predicate. All streams have the same number of tuples denoted $|S|$. All join attributes are categorical and have the same domain size D. In addition, all streams have N ranked attributes and N categorical attributes (excluding the join attributes and the class attribute). Categorical values are drawn randomly from a domain of size 20. All ranked attributes have the ranked domain {1,…,10}.

Since our goal is to verify that the classifier built on the join stream is more accurate when there are correlations among streams, the dataset must contain certain "structures" for the class label rather than random tuples. We construct the dataset in which the class label in a join tuple is determined by whether at least q percentage of the ranked attributes have a "high" value. A ranked value is "high" if it belongs to the top half of its ranked domain. Since the ranked attributes are distributed among multiple input streams, to ensure the desired property of the class label, the input streams $S_1,\ldots,S_k$ are constructed as follows.

- ***Join values***. Each stream S_i consists of D groups: from 1st to Dth group. All tuples in the jth ($1 \le j \le D$) group of S_i join with all tuples in the jth group of S_{i+1}, but not any other tuples. The *jth join group* refers to the set of join tuples produced by the jth groups. The size Z_j of the jth group is the same for all streams $S_1,\ldots,S_k$, and follows Poisson distribution with the mean $\lambda=|S|/D$. The jth join group has the size Z_j^k, with λ^k being the mean. The *blow-up ratio* of the join is defined as $\lambda^k/\lambda=\lambda^{k-1}$, i.e., the ratio between the mean of group size on the join stream and that on input streams.
- ***Ranked values***. We generate ranked attributes such that all join tuples in the jth join group have the same class label. In particular, we ensure that all join tuples in the same group have "high" values in the *same* number of ranked attributes, say hj. To this end, we distribute the number hj among $S_1,\ldots,S_k$ randomly, say $hj_1,\ldots,hj_k$, such that $hj = hj_1+\ldots+hj_k$, and all tuples in the jth group for S_i are "high" in hj_i ranked attributes. hj follows uniform distribution in the range $[0,k*N]$, where $k*N$ is the total number of ranked attributes.
- ***Class labels***. If $hj \ge q*k*N$, for some percentage parameter q, we assign the "Yes" class label to every tuple in the jth group of S_1, otherwise, assign the "No" class label.

Finally, to simulate the "concept drifting" in data streams, we change the parameter q every time after generating W tuples. In particular, for every W tuples we randomly determine a q value in the range [0.25, 0.75) following the uniform distribution. W is called the *concept drifting interval*. Usually W is larger than the window size because not every window leads to a change in classification. A dataset generated as above can be characterized by the parameters (N, $|S|$, D, λ, W), where $\lambda=|S|/D$ is the mean of group size and determines the blow-up ratio of join.

5.2.2 Accuracy

We generated three streams S_1, S_2 and S_3 with the parameter setting N=10, $|S|$=1,000,000, D=200,000, λ=5, W=100,000. Figure 7 shows the accuracy vs the window size with 50% window overlapping. *DT_Join* and *NB_Join* are more accurate than their counterparts on the single stream, while both having similar accuracies.

Figure 8 shows another experiment, where we fixed the window size w at 20,000 and decreased W from 100,000 to 20,000. Since the previous experiments have confirmed that classifiers built on the join stream have a better accuracy, in this experiment we only show the accuracy of *NB_Join* and *DT_Join*. As expected, the accuracy drops slowly as W decreases, since the structure for the class label changes more frequently.

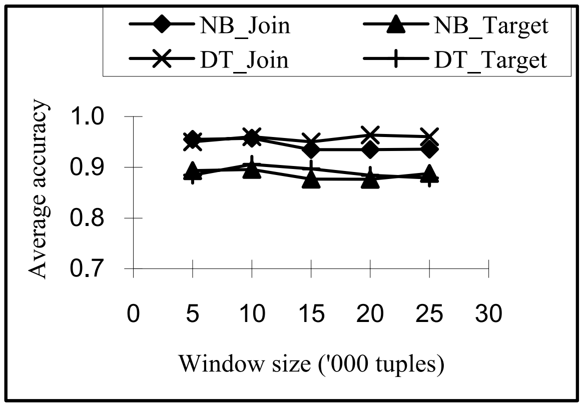


Figure 7. Classifier accuracy vs window size

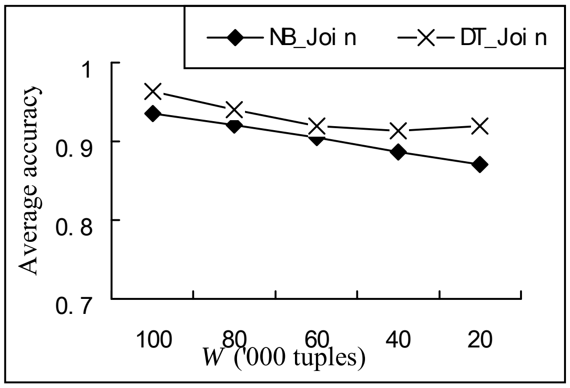


Figure 8. Classifier accuracy vs concept drifting interval

5.2.3 Time per input tuple

Figure 9 shows the time per tuple on the same dataset as in Figure 7. The join time is much larger than the time of *NB_Join.* As the window size increases, the join time increases due to the blow-up effect of join, while *NB_Join* spends almost constant time per tuple for any window size.

Figure 10 shows time per tuple vs. blow-up ratio of join. The parameters are fixed as N=10, $|S|$=1,000,000, D=200,000, W=100,000. For the join of three streams, the blow-up ratio is λ^2. By varying λ from 2 to 7, the blow-up ratio varies from 4 to 49. The window size is fixed at 20,000. Again, *NB_Join* shows a much better performance and is flat with respect to the blow-up of join. This is because it scans the window exactly twice, independent of the blow-up ratio of the join. On the other hand, the join takes more time per tuple with a larger blow-up ratio because much more tuples are generated.

Figure 11 shows time per tuple vs. number of streams. All parameters are still the same as in Figure 9. The window size is fixed at 20,000 tuples. We vary the number of steams from 1 to 5. The blow-up ratio for k-stream join is determined by $5^{(k-1)}$. The comparison of the results is similar to Figure 10.

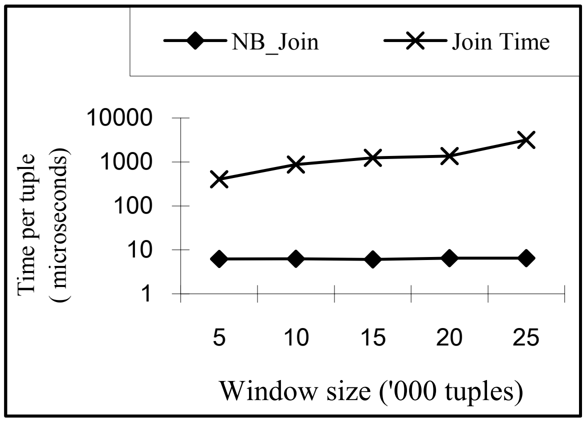


Figure 9. Time per input tuple vs. window size

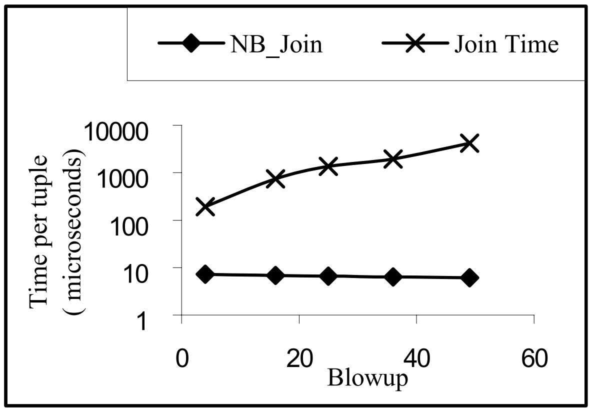


Figure 10. Time per input tuple vs. blow-up ratio

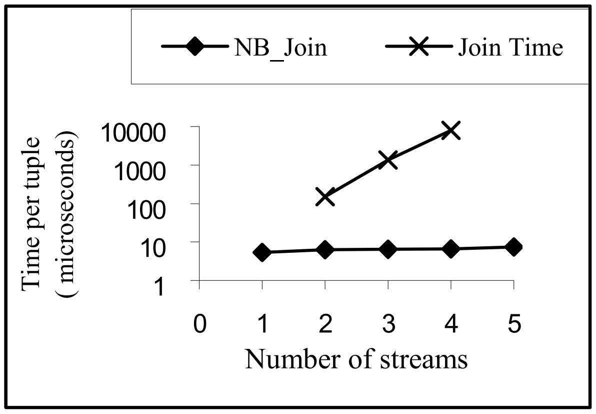


Figure 11. Time per input tuple vs. number of streams

5.3 Discussion

On both real life and synthetic datasets, our empirical studies showed that when the features for classification are contained in several related streams, the proposed join stream classification has significant accuracy advantage over the conventional method of examining only the target stream.

The main challenge is how such classification can be performed in pace with the high-speed input streams, given that the join stream has an even higher data arrival rate than that of the input streams. To this end, our experiments showed that our proposed algorithm has a cost linear in the size of input streams, independent of the join size. This feature makes our algorithm superior to other alternative methods.

It is worthy of noting that the classifier must be rebuilt each time the window on any input stream slides forward. This is reasonable when there is no overlap or only small overlaps between windows. However, when windows are significantly overlapped, this strategy tends to repeat the work on the overlapped data. In this case, a more efficient strategy may be incrementally updating the NBC by working only on the difference due to the window sliding. We did not pursue in this direction further because even overlapped tuples still need to be joined with *new* tuples in other streams, which means that the scan of overlapped tuples cannot be avoided. Since our algorithm scans the current window only twice, the benefit of being incremental is limited, especially considering the overhead added.

6. CONCLUSIONS

Motivated by real life applications, we considered the classification problem where the training data are coming from several related private data streams. Joining all streams violates the privacy of stream owners and suffers from the blow-up of the join. We presented a solution based on Naïve Bayesian Classifiers. The main idea is rapidly obtaining the essential join statistics without actually computing the join. With this technique, we can build exactly the same Naïve Bayesian Classifier as using the join stream without exchanging private information. The processing cost is linear in the size of input streams and independent of the join size. Empirical studies supported our claim that examining several related streams indeed benefits the quality of classification. Having a much lower processing time per input tuple, the proposed method is able to handle much higher data arrival rate and deal with the general many-to-many join relationships of data streams.

7. REFERENCES

[1] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. Tracking Join and Self-Join Sizes in Limited Storage. *In ACM PODS*, 1999.

[2] B.Babcock, S. Babu, M. Datar, R. Motwani, J. Widom. Model and issues in data stream systems. *In ACM PODS*, Madison, Wisconsin, 2002.

[3] J. Beringer and E. Hullermeier. Online clustering of parallel data streams. In press for *Data & Knowledge Engineering,* 2005.

[4] Y. D. Cai, D. Clutter, G. Pape, J. Han, M. Welge and L. Auvil. MAIDS: Mining alarming incidents from data streams. *In Proc. SIGMOD*, demonstration paper, 2004.

[5] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. *In Proc. VLDB,* 2002.

[6] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On random sampling over joins. *In Proc. SIGMOD,* 1999.

[7] G. Chen, X. Wu, X. Zhu. Sequential pattern mining in multiple streams, *In Proc. ICDM,* 2005.

[8] A. Das, J. Gehrke and M.Riedewald. Approximate join processing over data streams. *In Proc. SIGMOD*, Madison, Wisconsin, 2003.

[9] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. *In Proc. SIGMOD*, Madison, Wisconsin, 2002.

[10] P.Domingos and G. Hulten. Mining high-speed data streams. *In Proc. SIGKDD,* 2000.

[11] Pedro Domingos and Michael Pazzani. On the optimality of the simple Bayesian classifier under zero-one loss. *Machine Learning,* 29:103-130, 1997.

[12] R. O. Duda and P. E. Hart. *Pattern classification and scene analysis*. New York: John Wiley & Sons, 1973.

[13] J. Gama, R. Racha, P.Medas. Accurate decision trees for mining high-speed data streams. *In Proc. SIGKDD,* 2003.

[14] S. Ganguly, M. Garofalakis, A. Kumar and R. Rastogj. Join-distinct aggregate estimation over update streams*. In Proc. ACM PODS*, Baltimore, Maryland, 2005.

[15] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. *In FOCS,* 2000.

[16] D. J. Hand and K. Yu, Idiot's Bayes - not so stupid after all? *International Statistical Review*. 69(3), 385-399, 2001.

[17] Irina Rish. An empirical study of the naive Bayes classifier. *IJCAI 2001 Workshop on Empirical Methods in Artificial Intelligence*, 2001.

[18] U. Srivastava, J. Widom. Memory-limited execution of windowed stream joins. *In Proc. VLDB,* 2004.

[19] H. Wang, W. Fan, P. Yu and J. Han. Mining concept-drifting data streams using ensemble classifiers. *In Proc. SIGKDD*, 2003.

[20] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. *In Proc. VLDB,* 2002.

[21] K. Wang, Y. Xu, R. She, P. Yu. Classification Spanning Private Databases. *AAAI*, 2006.

[22] W. Du and Z. Zhan. Building decision tree classifier on private data*. ICDM Workshop on Privacy, Security and Data Mining*, 2002

[23] F. Li, J. Sun, S. Papadimitriou, G. Mihala and I. Stanoi. Hiding in the Crowd: Privacy Preservation on Evolving Streams through Correlation Tracking. *In Proc. ICDE*, 2007.

[24] R. Agrawal, A. Evfimievski and R. Srikant. Information sharing across private databases. *In Proc. SIGMOD*, 2003.

[25] R. Agrawal, and R. Srikant. Privacy-preserving data mining. In *Proc. SIGMOD* 2000.

[26] Y. Lindell AND B. PINKAS, B. 2000. Privacy preserving data mining. In *Proc*. *CRYPTO* 2000.

[27] L. Sweeney. *k*-Anonymity: A Model for Protecting Privacy, International Journal on Uncertainty, Fuzziness and Knowledge-based Systems, 10(5), 2002.

[28] J. Vaidya and C. W. Clifton. Privacy preserving association rule mining in vertically partitioned data. In *SIGKDD*, 2002.

[29] K. Chen and L. Liu. Privacy preserving data classification with rotation perturbation. In *ICDM*, 2005.

[30] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. *l*-Diversity: Privacy beyond *k*-anonymity. *ICDE* 2006.

[31] C. Aggarwal, J. Han, J. Wang, and P. Yu. A Framework for On-Demand Classification of Evolving Data Streams. *IEEE TKDE*, Vol. 18, No. 5, May 2006, PP:577-589

[32] L. Golab and M. Tamer Ozsu. Processing sliding window multi-joins in continuous queries over data streams. *In Proc. VLDB*, 2003

[33] C. Agarwal and P. Yu. A condensation Approach to Privacy Preserving Data Mining. *In Proc*. *EDBT*, 2004.

[34] M. Levene and G. Loizou. Why is the snowflake schema a good data warehouse design? *Information Systems* 28(3), 2003.

[35] O. Goldreich. Secure multi-party computation. Working Draft, Version 1.3, June 2001.

[36] J. Bethencourt, D. Song, and B. Waters. Constructions and Practical Applications for Private Stream Searching. *In IEEE Symposium on Security and Privacy*, 2006.

[37] R. Ostrovsky, W. Skeith, Private Searching on Streaming Data. *In CRYPTO*, 2005.

[38] Bank data sifted in secret by US to block terror. The New York Times, June 23, 2006.