Chapter 5

Mining in Very Fast Stream Data

Due to high requirement of memory for stream data motivate us to design new, computation and memory efficient algorithms to provide approximate result with higher accuracy.

Traditional databases store sets of relatively static records with no pre-defined notion of time, unless time stamp attributes are explicitly added. The detail of decision tree construction, Entropy function and criteria for splitting the data set is discussed in chapter 4. While data stream model adequately represents commercial catalogers or repositories of personal information, many current and emerging applications require support for on-line analysis of rapidly changing data streams. Limitations of traditional DBMSs in supporting streaming applications have been recognized, prompting research to augment existing technologies and build new systems to manage streaming data [GoO03].

Current research typically focuses on on-line techniques, which are based on a "recently" observed portion of the stream. There are, however, applications, which requires storing and accessing the history of a data stream. Examples include stock market information systems and scientific data warehouses that continuously collect very large amount of information about natural phenomena (earth surface temperature, air pressure, etc.) or complex experiments (e.g., particle physics experiments at CERN). In order to discover trends or patterns of interest, historic data is maintained and frequently accessed as well. Hence in addition to on-line monitoring, the
system has to support efficient high-speed archiving and archive access.

One of the main difficulties in mining non-stationary continuous data streams is to cope with the changing stream data. The fundamental processes generating most real-time data streams may change over years, months and even seconds, at times drastically. This changes are also known as concept drifting [WFU03], causes the data-mining model generated from past data, to become less accurate in the classification of new data. It is impossible to mine the rule from the entire data at one time.

The key issue in mining on streaming data is that only one pass is allowed over the entire data. Moreover, there is a real-time constraint, i.e. the processing time is very limited and the memory available to store any summary information is also bounded. For most data mining problems, a one-pass algorithm cannot be very accurate. Data mining algorithm developed for streaming data also serve as a useful basis for creating approximate, but scalable, implementation for very large and disk resident data sets. Second 5.1 will present time stamp in stream data. Section 5.2 will outline predictive data modeling for streaming data. Windowing concept is discussed in 5.3. Section 5.4 concerns the tilted time window concept. The experimental evaluation is discussed in section 5.4, finally the chapter is summarized in section 5.5.

5.1 Time Stamps in Streams

Time stamps in stream data are categorized into two categories. First implicit time series in which the system adds a special field to each incoming tuples, and second, explicit timestamps, in which a data attribute is designated as the time stamp. Explicit timestamps are used when each tuple correspond to a real world event at a particular time that is of importance. In our approach we are using implicit time attribute for mining the knowledge from the database. Table 5.1 shows how the packets are extracted from the database and when packets are received we are including one time factor (duration) with each packet of data.
In order to mine the incoming data stream the existing approach decision tree classifier C4.5 [Qui86] is used. The basic working of decision tree is discussed in chapter 4.

### 5.2 Predictive data model for stream data

Using sampling [JA00] the problem of selecting predicate based on samples taken from the dataset. In the discussion entropy as the gain function. Suppose \( S \) be a sample taken from data set \( D \). The gain function \( g_c \) associated with a potential split point \( c \) for a numeric attribute \( x_i \). If \( p_1 \) and \( p_2 \) are the fractions of data instances with class labels 1 and 2, respectively, \( \bar{p}_1 \) and \( \bar{p}_2 \) are the estimates computed using the sample. Similarly \( p_{IL}, \bar{p}_{IL}, \bar{p}_{IR} \) are the left and right subtree.

\[
\bar{g}_c = g(\bar{p}_L, \bar{p}_R) = \overline{\text{Entropy}(D)}
\]

\[
= -\bar{p}_L(1-\bar{p}_L)log\bar{p}_L - (1-p_{IL})(log(1-p_{IL})) - (1-\bar{p}_L)\bar{p}_{IR}log\bar{p}_{IR} - (1-\bar{p}_{IR})(log(1-\bar{p}_{IR}))
\]

The value of \( \bar{g}_c \) serves as the estimate of \( g_c \). We do not need to compute \( \overline{\text{Entropy}(D)} \), since the main interest in the relative values of the gain values associated with different split points.

In order to maximize the gain value associated with split points. To find the

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Duration</th>
<th>Bytes</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1.0.2</td>
<td>16.2.37</td>
<td>12</td>
<td>20K</td>
<td>http</td>
</tr>
<tr>
<td>18.6.7.1</td>
<td>12.4.0.3</td>
<td>16</td>
<td>24K</td>
<td>http</td>
</tr>
<tr>
<td>13.9.4.3</td>
<td>11.6.8.2</td>
<td>14</td>
<td>20K</td>
<td>http</td>
</tr>
<tr>
<td>15.2.2.9</td>
<td>17.1.2.1</td>
<td>19</td>
<td>40K</td>
<td>http</td>
</tr>
<tr>
<td>12.4.3.8</td>
<td>14.8.7.4</td>
<td>26</td>
<td>58K</td>
<td>http</td>
</tr>
<tr>
<td>10.5.1.3</td>
<td>13.0.0.1</td>
<td>27</td>
<td>100K</td>
<td>ftp</td>
</tr>
<tr>
<td>11.1.0.6</td>
<td>10.3.4.5</td>
<td>32</td>
<td>300K</td>
<td>ftp</td>
</tr>
<tr>
<td>19.7.1.2</td>
<td>16.5.5.8</td>
<td>18</td>
<td>80K</td>
<td>ftp</td>
</tr>
</tbody>
</table>

Table 5.1: Stream dataset
best split point using the estimate of gain. Let \( \overline{g(x_i)} \) be the estimate of gain that can be obtained from attribute \( x_i \) such that

\[
\overline{g(x_i)} - \max_{j \in \{1..m\} - \{i\}} g(x_i) \geq \epsilon
\]  

(5.2)

where \( \epsilon \) is a very small positive number. The above test is called statistical test that is used to infer that \( x_i \) is likely to satisfy the original test for choosing the best attributes.

\[
g(x_i) \geq \max_{j \in \{1..m\} - \{i\}} g(x_i)
\]  

(5.3)

To describe the confidence of statistical inference, a parameter \( \alpha \) is used, it is the probability that the original test holds if the statistical test holds and should be closed to 1 as possible, \( \epsilon \) can be viewed as a function of \( \alpha \) and sample size \( |s| \) i.e. \( \epsilon = f(\alpha, |S|) \)

Domino et al. [DH00] uses the Hoeffding bound to construct the function and the formula is

\[
\epsilon_n = \sqrt{\frac{R^2 \ln(1/(1 - \alpha))}{2 \times |S|}}
\]  

(5.4)

Where \( R \) is the speed of gain function. In this context, where there are two classes and entropy is used to measure the impurity, \( R = 2 \). Based upon the probabilistic bound on the splitting condition for each node, Domingos and Hulten derive the result based on hoeffding tree.

### 5.3 Sliding Window Approach

Maintaining windowing is one of the solutions for streaming dataset, due to huge set of data in stream. Many algorithms proposed for mining on stream data. A subset of the training set, called window, is chosen at random and a decision tree is constructed using only this window. While computing on infinite stream data
is trivial and complex, doing so in a sliding window of size $N$ requires $(N)$ space. Consider a sequence of non-increasing values, in which the oldest item in any given window is the maximum and must be replaced whenever the window moves forward. Thus, the fundamental problem is that as new items arrive, old items must be simultaneously evicted from the window [GoO03]. The proposed Sliding window is based on the concept "make decisions based only on recent data of sliding window size $w$, an element arriving at time $t$ expires at time $t + w$".

5.3.1 Aggregate Window

Aggregation over the sliding window is one of the technique of computation in limited memory by dividing the window into small portions (called basic windows) and only storing an aggregate and a time stamp for each portion. When the time stamp of the oldest basic window expires, its synopsis is removed, a fresh window is added to the front, and the aggregate is incrementally recomputed.

This method has been used to compute correlations between streams and to find frequently appearing items, but results are refreshed only after the stream fills the current basic window, if the available memory is small, the refresh interval is large. Moreover, some statistics may not be incrementally computable from a set of synopses.

Suppose $X$ be a series of data of length $n$ is represented in $N$ space by a vector. Suppose the data reference strings are $X = x_1, x_2, x_3, \ldots, x_n$.

\[
X_i = \frac{N}{n} \sum x_i
\]  

(5.5)

To reduce the data from $n$ dimensions to $N$ dimensions, the data is divided into $N$ equalize window frame. If the data arrives $X = \langle -1, -2, -1, 0, 2, 1, 1, 0 \rangle$ the aggregate value $(\text{mean}(-1, -2, -1, 0), \text{mean}(2, 1, 10)) = (-1, 1)$.

For the sub sequence sliding window the data is aggregated by insertion on new incoming data and distraction of last expired record. That can be represented using
formula presented by Keogh [KCP00] which is applicable for stream data also.

\[ X_{ji} = x_{j-i} - \frac{N}{n} x_{\frac{s}{2}(i-1)+1} + \frac{N}{n} x_{\frac{s}{2}i+1} \]  

(5.6)

As the paper [KCP00] the first feature extracted had its value calculated by the mean of \( S(-1,-2,-1,0) = -1 \). In sliding window the value 1 is removed and +2 is added rather than calculating the whole sequence in window.

e.g. \(-1 - \frac{3}{8}(-1) + \frac{3}{8}(2) = -\frac{1}{4}\)

Many infinite stream algorithms uses tilted window model. In order to mine in sliding window if a window of size \( N \) requires \((N)\) space and a sequence of non-increasing values, in which the oldest item in any given window is the maximum and must be replaced whenever the window moves forward (called new batch). Thus, the fundamental problem is that as new items arrive, old items must be simultaneously evicted from the window [Go003].

Many infinite stream algorithms uses tilted window model. In order to mine in sliding window if a window of size \( N \) requires \((N)\) space and a sequence of non-increasing values, in which the oldest item in any given window is the maximum and must be replaced whenever the window moves forward (called new batch). Thus, the fundamental problem is that as new items arrive, old items must be simultaneously evicted from the window [Go003].

Query 1: "Count the number of bids for each auction site in the past 4 minutes and update that result every 1 minute."

```
SELECT auction-site, count(*)
FROM bids
[WHERE time stamp RANGE 4 minutes SLIDE 1 minute]
GROUP-BY auction-site
```

Query 1 is called a time-based sliding-window aggregate query, and it calculates
the count of bids from each auction site for each 4-minutes sub-stream that starts every 1-minute, with respect to the time stamp attribute of the data. In Query 1 shows above a window specification is introduced with three parameters: RANGE specifies the window size, SLIDE specifies how the window moves, and WATTR specifies the windowing attribute on which that the RANGE and SLIDE parameters are defined. The window specification of Query 1 breaks the bid stream into window extents (4-minute overlapping sub-streams). This query result is a descriptive model of data set, it is not appropriate for predictive model of data set.

<table>
<thead>
<tr>
<th>Window extent</th>
<th>WID</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:02 PM - 12:06M</td>
<td>6</td>
</tr>
<tr>
<td>12:03 PM - 12:07PM</td>
<td>7</td>
</tr>
<tr>
<td>12:04 PM - 12:08 PM</td>
<td>8</td>
</tr>
<tr>
<td>2:05 PM -12:09 PM</td>
<td>9</td>
</tr>
<tr>
<td>12:06 PM - 12:10 PM</td>
<td>10</td>
</tr>
<tr>
<td>12:07 PM - 12:11 PM</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 5.2: Window extent

Algorithm 17 Sliding window algorithm
Input: Window size w, Double ended Queue DQ
Output: Rule

1. DQ← Null;
2. Rear ← Null; Front ← Null // Initialize the front and Rear pointer
3. If (Rear = w) then {
4. Front ++ // Delete one item from front the item which have the highest distance }
5. DQ[rear++] ← New Incoming stream // if not full condition arises continue insert new data with rear pointer.
6. end
In above algorithm a double ended queue (DQ) is used, in order to maintain the data on specified window. When a new data arrives on dequeue recently upcoming stream is inserted from rear pointer and highest distance data stream is removed from the front pointer of queue.

5.3.2 Decision tree with sliding window

In proposed system we have used sliding window over the stream data. Due to continuously changes in stream data, old data is eliminated and new data is incorporated. Each time count is incremented when new data arrives and old obsolete data is decremented. The increases in sub tree as the new data arrived; when alternate more accurate node arrives it replaces the old data. The approach uses the information gain. It is the enhancement over the existing C4.5 algorithm by replacing the old sub tree with new sub tree when a best attribute is found.

The total size of the data is much larger than the available memory. It is not possible to store and re-read all data from memory. Thus, a single pass algorithm is required, which also needs to meet the real-time constraints, i.e. the computing time for each item should be less than the interval between arrival times for two consecutive items.

Jin et al. [JA00] presents high level algorithm for construction of decision tree on streaming data. This algorithm is the basis for extension to our approach. The algorithm uses two queues, DQ and AQ. DQ for double ended queue, AQ stands for active queue, denotes the set of decision tree nodes that we are currently working on expanding.

Q is the set of decision nodes that have not yet been split, but are not currently being processed. This distinction is made because actively processing each node requires additional memory. For example, we need to store the count and time associated with each distinct value of attributes. The DQ is constructed from the AQ by including as many nodes as possible, till sufficient memory is available.
Initially the tree is treated as null tree.

The input to the algorithm is a stream of data, denoted by D. Time \( t \) is successively obtained from the stream. If the node belongs to time \( t \) it is added to AQ. The tree is constructed from AQ. Continuously the data is added to DQ. If sufficient information is loaded into AQ is spitted. The algorithm terminated when DQ and AQ both are empty.

The algorithm, as presented here, is different from Jin approach by using double-ended queue. Using double queue data is extracted one end and data is deleted from other end for a fixed size of stream and it is also maintaining the timing information of incoming streams. The memory requirement for processing a set of nodes is one of the issues we are optimizing in our work. If the memory requirements for processing a given node in the tree are reduced, more nodes can be fit into the set.

Other than the memory requirement, this algorithm also exposes a number of other issues in decision tree construction on streaming data.

The structure for stream data

```c
Struct stream_data
{
  data;
  time intime;
};
```
Algorithm 18 Stream tree with sliding window

Queue DQ, AQ
Node node,
Time intime;

1. AQ ← Null; DQ ← Null // Initialize null value to Active Queue and Dequeue
2. Add (root , AQ) // Add root node to Active
3. While not Null (AQ) not Null (DQ)
4. {
5. t ← D.get();
6. node ← classify(root, t);
7. if node ∈ AQ
8. add (node.sample, tree);
9. if node Satisfy stopping criteria
10. {
11. remove(node, AQ);} //remove data that obsolete from time window
12. if node.enough.Sample available = true then {
13. sp = find split_point(D) // Find the split point
14. node.create() ← split(sp, node1, node2);
15. remove(node, AQ);
16. add((node1, node2), DQ); }
17. While enough memory(AQ, DQ)
18. get(node, Q);
19. add(node, AQ)
20. } // End of procedure

5.4 Tilted time window

If the window is incremented and move onto stopping condition is called tilted time window. In stream data analysis, both the recent data and historical data are required, but the recent data are usually more important than the historical
data. People are often interested in recent changes rather than traditional records. Limited memory space is available, it is impossible to store the history in full scale. The most recent time can be registered in finest levels of granularity and the most distant time recorded in coarseness granularities. Data is aggregated in and transferred to coarser granularity for the distant time period, we can substantially save storage space and computational time, and also preserve the important information.

5.4.1 Natural Tilted-Time Window

The natural tilted-time window can be illustrated using following example. When four quarters are accumulated, they merge together to constitute one hour. After 24 hours are accumulated, one day is built. In the natural tilted-time window, at most tilted windows need to be maintained for a period of one month \([\text{GHP03]}\). For mining time based stream, we can maintain incremental tilted time window in order to answer time-based queries.

Figure 5.3 shows such a tilted-time window: the most recent 4 quarters of an hour, then the last 24 hours, and 31 days. Based on this model, one can compute frequent item sets in the last hour with the precision of a quarter of an hour, the last day with the precision of an hour, etc. This model registers only units of time, with an acceptable trade-off of lower granularity at distant times.

![Figure 5.2: Tilted time window with Quarter](image)

Tilted window required to maintain for one month = 31 + 24 + 4 = 59 units
Tilted window required to main data for one year = 365 + 31 + 24 + 4 = 424 units
The natural tilted time window is implemented using circular queues. Each queue have a specific time granularities. Example six granularities minutes, quarter, hour, day, month and year to store the aggregates from the finest level to the coarsest level. Each granularities required a separate queue.

Example: Suppose data is recorded from 9:00 time. When time reaches 9:16 data from 9:01-9:15 is accumulated and propagated to Quarter queue. This technique compresses the data without loss of important information, and make possible to long run analysis on data stream, because it reduces the size of data stream.

Example: Suppose in Natural tilted window have four granularity levels

1. Minute in the first 15 minutes
2. Quarter in the first hour
3. Hour in the first 24 hours
4. Days in the first month

To perform above task four circular queues is maintained:

1. Queue for minute : 15 items
2. Queue for quarter : 4 items
3. Queue for hour : 24 items
4. Queue for day : 31 items

Each queue has a tracking pointer that makes a flag on last updated slot. The minute queue has a time stamp to track the last updated in minute. The hour queue has a time stamp to track the last updated hours.
Example 2: The management of stream in tilted window is as follows.

Minute Queue: 

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

A batch data received at 8:05. Count = 3.

Minute Queue: 

| 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Data received at 8:08. Count = 5

Minute Queue: 

| 0 | 0 | 3 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 |

8:08 - 8:05 = 03, reset the counter flag to 3, slots of tracking pointer to 0, advance the tracking pointer 3 slots, and insert the count = 5

New Batch received 8:17, count = 4

Minute Queue: 

| 0 | 4 | 0 | 0 | 3 | 0 | 0 | 5 | 0 | 0 | 0 | 0 |

Quarter Queue: 

| 8 | 0 | 0 | 0 |

8:17 - 8:08 = 09, advances the tracking pointer 9 slot, When the pointer reaches the end of the queue, the counts is summarized and propagated to the next granularity level. Last update time = 8:17.

A new data arrives on 8:37 count = 9

Minute Queue: 

| 0 | 0 | 0 | 0 | 0 | 0 | 9 | 0 | 0 | 0 | 0 | 0 |

Quarter Queue: 

| 8 | 4 | 0 | 0 |

Since 8:37 - 8:17 = 20, advance the tracking pointer 20 slots. When the pointer reaches the end of the queue, summarize the counts and propagate to the next granularity level. Last update time = 8:37.

Data received at 9:06, count 8
• Minute Queue \[0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0\]
• Quarter Queue \[8 4 9 0\]
• Hour Queue \[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0\]

9:06 - 8:37 = 29, when the tracking pointer reaches the end of the queue, summarized count is propagated to next granularities level, when the queue of quarter is reached it summarized an propagation to next granularities level Hours. The Hours queue is updated by \(8 + 4 + 9 = 21\).

In order to reduce the number of tilted window another approach is used is called as Logarithmic Tilted -Time window, which is discussed in section below.

### 5.4.2 Logarithmic Tilted -Time window

The concepts of logarithmic tilted window is proposed by Giannella et. al. [GHP03]. It is based on logarithmic time scale as shown in Figure 5.3 5.4. The current window holds the transaction on current quarter. Then the remaining slots are for the last quarter, the next two quarters are 4 quarter, 8 quarter, 16 quarter, etc, growing at exponential rate of 2. The stream of transactions is broken up into fixed size of batches \(B_1, B_2, \ldots, B_n\). Where \(B_n\)'s is the most current batch and \(B_1\) is the oldest batch.

\[
B(i,j) = \cup_{k=j}^i B_k \forall i > j \quad (5.7)
\]

\(f(i,j)\) denote the frequency of \(I\) in \(B(i,j)\). Record the frequency for item set \(I\).

Figure 5.4: Logarithmic time scale
A logarithmic tilted window is used to store the stream of data. The frequency of stream are represented $f(n, n); f(n - 1, n - 1); f(n - 2, n - 2); f(n - 4, n - 7)$.

The exponential growth of data 1, 1, 2, 4, 8, 16..

*When new data arrives (batches) how the tilted time window is updated?*

\[
\begin{align*}
  f(n, n) & \text{ level 0} \\
  f(n - 1, n - 1) & \text{ level 1; } \\
  f(n - 2, n - 3) & \text{ level 2;} \\
  f(n - 4, n - 7) & \text{ level 4; }
\end{align*}
\]

When new data arrives $f(n + 1, n + 1)$ data is shifted to lower level

\[
\begin{align*}
  f(n + 1, n + 1) & \text{ level 0;} \\
  f(n, n) & \text{ level 1} \\
  f(n - 1, n - 1) & \text{ level 2} \\
  f(n - 2, n - 3) & \text{ level 3} \\
  f(n - 4, n - 7) & \text{ level 4.}
\end{align*}
\]

The process requires intermediate window. This intermediate window will replace or be merged with tilted time window when they are full. The process is continues till intermediate window in not full (with frequency $f$), if it is full then shifted to next window.
Algorithm 19 Framework Tilted time window

Input: Window size $n$

Method:

1. Initially the intermediate level for all window is empty
2. When stream data arrive
3. Shift the recently arrived data to next level
4. IF the intermediate window is empty
5. Shift all $n$th data to $(n-1)$ level.
6. Else {
7. If intermediate occurs Shifting stop.
8. $f(n-1, n-1) + f$ is shifted back to same level
9. }
10. end
11. Till the stopping condition not arises.

Example: There are eight batches of data $B1, B2, B3, B4, B5, B6, B7, B8$. The logarithmic tilted time window looks like $f(8, 8); f(7, 7); f(6, 5); f(4, 1)$. The intermediate window is empty. In logarithmic tilted time window there are $\lfloor \log_2(n) \rfloor + 1$ frequencies in each level.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>granularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(8, 8)$</td>
<td>0</td>
</tr>
<tr>
<td>$f(7, 7)$</td>
<td>1</td>
</tr>
<tr>
<td>$f(6, 5)$</td>
<td>2</td>
</tr>
<tr>
<td>$f(4, 1)$</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5.3: Tilted time window at granularity level 0

If a new batch $B9$ arrives it shifted all the data and no merging operation because all intermediate window is empty.
CHAPTER 5. MINING IN VERY FAST STREAM DATA

If a new batch $B_{10}$ arrives when data $f(9,9)$ is shifted from level 1 to 2, since the intermediate window is full, the frequencies of level two are merged.

$f(8,7) = f(8,8) + f(7,7)$. $f(8,7)$ is stored on level 2, replacing $f(6,5)$. Since the intermediate window at level 2 is empty, $f(6,5)$ is put into intermediate window as shown in the table.

| $f(10,10)$ | 0 |
| $f(9,9)$   | 1 |
| $f(8,8)$   | 2 |
| $f(6,5)$   | 4 |
| $f(4,1)$   | 4 |

After the operation

If batch $B_{11}$ arrives the updating on window.

| $f(11,11)$ | 0 |
| $f(10,10)$ | 1 |
| $f(9,9)$   | 2 | Intermediate window
| $f(8,7)$   | 4 |
| $f(6,5)$   | 4 | Put onto intermediate window
| $f(1,1)$   | 4 |
After the merging operation,

<table>
<thead>
<tr>
<th>Window</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(12, 12) )</td>
<td>0</td>
</tr>
<tr>
<td>( f(11, 11) )</td>
<td>1</td>
</tr>
<tr>
<td>( f(10, 9) )</td>
<td>2</td>
</tr>
<tr>
<td>( f(8, 5) )</td>
<td>3</td>
</tr>
<tr>
<td>( [f(4, 1)] )</td>
<td>4 Intermediate window</td>
</tr>
</tbody>
</table>

In this section, we present our time based windowing algorithm for streaming data. We have addressed a novel approach for time based attribute pruning for stream data. The basis of our method is to maintain a sliding window of user defined data size.
CHAPTER 5. MINING IN VERY FAST STREAM DATA

Algorithm 20 Mining Data stream with tilted time window

Input: Data stream D
Queue: Active Queue, Dequeue DQ, Queue for each level of granularities
Output: Decision Tree for time t.

Classifier (Node N, Stream D)
{
1. Initialize root node using data set S
2. DQ ← Null, AQ ← Null
3. Initialize queue Q to contain root node
4. S ← D.get();
5. Store stream and time information to AQ increment the rear pointer.
6. While Not receive new data on Active Queue AQ {
7. node ← classify(root, t);
8. If node is not pure {
9. For each attribute A
10. Evaluate splits on attribute A
11. (N1, N2) ← node.Create()
12. Do event()
13. remove( node, AQ)
14. Append N1 and N2 to AQ }
15. else New data Received
16. delete the item from lower granularities to higher level granularities;
17. goto (5);
18. enc:
}

The pseudo code for our mining on stream data is presented in algorithm 20. Here data is collected from stream over time t. We are maintaining a Dequeue DQ. The decision tree is constructed at time t, by taking the data using front pointer and new data is inserted using rear pointer. When the data is inserted to DQ it stores the information and time of data onto Dequeue. During the construction of
tree it creates only for specified time period, if queue is full the data is summarized and propagated to higher level window. Instead of recording whole data stream, simple queue over tilted windows may be computed in limited memory by dividing the window into different level of granularities

5.5 Experimental evaluation

To evaluate the efficiency of our algorithm we have perform the experimental study. In this section, we report our experimental result over the window based mining streams. All the experiments are performed on 500 MHz Pentium IV machine, with 128 main memory, 40 GB hard disk. The data set is taken from machine learning repositories, the comparison is done with existing VFDT algorithm.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Data</th>
<th>Allocation</th>
<th>Time Seconds</th>
<th>Size</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most common</td>
<td>Hypo</td>
<td>8139</td>
<td>.04</td>
<td>2514</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Crx</td>
<td>4524</td>
<td>.00</td>
<td>4524</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Golf</td>
<td>1466</td>
<td>0.0</td>
<td>1466</td>
<td></td>
</tr>
<tr>
<td></td>
<td>monk1</td>
<td>442</td>
<td>0.0</td>
<td>442</td>
<td></td>
</tr>
<tr>
<td></td>
<td>monk2</td>
<td>442</td>
<td>0.0</td>
<td>442</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Soybean</td>
<td>11677</td>
<td>.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vote</td>
<td>446</td>
<td>0.0</td>
<td>446</td>
<td></td>
</tr>
<tr>
<td>VFDT</td>
<td>Hypo</td>
<td>1919243</td>
<td>.05</td>
<td>2514</td>
<td>97/1258</td>
</tr>
<tr>
<td></td>
<td>Crx</td>
<td>284754</td>
<td>.03</td>
<td>4524</td>
<td>90/200</td>
</tr>
<tr>
<td></td>
<td>Golf</td>
<td>11305</td>
<td>.0</td>
<td>1466</td>
<td>Error</td>
</tr>
<tr>
<td></td>
<td>monk1</td>
<td>1199</td>
<td>.0</td>
<td>442</td>
<td>Error</td>
</tr>
<tr>
<td></td>
<td>monk2</td>
<td>1199</td>
<td>.0</td>
<td>442</td>
<td>Error</td>
</tr>
<tr>
<td></td>
<td>Soybean</td>
<td>2189153</td>
<td>.04</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vote</td>
<td>1233</td>
<td>0.0</td>
<td>446</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: Memory usage and execution time

Table 5.4 memory usage and execution time is studied for most common classes and VFDT learner using different dataset.

The following experiments are performed with CVFDT algorithm, the source code is taken from http://www.csc.liv.ac.uk/~frans/. The memory allocation and execution time for hypo, crx, golf, soybean dataset are studied.
CHAPTER 5. MINING IN VERY FAST STREAM DATA

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Dataset</th>
<th>Allocation</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVFDT</td>
<td>Hypo</td>
<td>1990513</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>Crx</td>
<td>267888</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>golf</td>
<td>9197</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>Soybean</td>
<td>692808</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>vote</td>
<td>1685</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 5.5: Performance with CVFDT data set

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>data set</th>
<th>Memory allocation</th>
<th>Execution time</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sliding window</td>
<td>Hypo</td>
<td>170449</td>
<td>0.07</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td></td>
<td>407705</td>
<td>0.07</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td></td>
<td>809953</td>
<td>0.07</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1587249</td>
<td>0.05</td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td>Crx</td>
<td>1990513</td>
<td>0.05</td>
<td>5000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1990513</td>
<td>0.01</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1990513</td>
<td>0.01</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1990513</td>
<td>0.00</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1990513</td>
<td>0.00</td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1990513</td>
<td>1685.01</td>
<td>5000</td>
</tr>
<tr>
<td></td>
<td>golf</td>
<td>9197</td>
<td>0.0</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9197</td>
<td>0.0</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9197</td>
<td>0.0</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9197</td>
<td>0.0</td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9197</td>
<td>0.0</td>
<td>5000</td>
</tr>
<tr>
<td></td>
<td>Soybean</td>
<td>209768</td>
<td>0.00</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td></td>
<td>504892</td>
<td>0.00</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td></td>
<td>692704</td>
<td>0.01</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>692704</td>
<td>0.01</td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>692704</td>
<td>0.01</td>
<td>5000</td>
</tr>
<tr>
<td></td>
<td>vote</td>
<td>1581</td>
<td>0.0</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1581</td>
<td>0.0</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1581</td>
<td>0.0</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1581</td>
<td>0.0</td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1581</td>
<td>0.0</td>
<td>5000</td>
</tr>
</tbody>
</table>

Table 5.6: Performance after applying sliding Window

Analysis. In this section, we analyze the scalability of both the algorithm. That is, we change the window size, the memory allocation is reduced. Data is treated like it is stored on different buckets which reduces the memory requirement. When the size of window is small the required memory is less, when the size of window is increases, the memory requirement is approximate equal to very fast stream data algorithm (CVFDT).
Analysis. The experimental result shows that as the size of window is small the memory usage is less as the window size increases the memory requirement is increasing. For a limit range of window size (1000) the memory usage is constant, there is no improvement in memory requirement.
Analysis. The experimental result shows that as the size of window is small the memory usage is less as the window size increases the memory requirement is increased.
Analysis. The experimental result shows that the memory requirement is same for all size of window, as the resultant windowing size is applicable if the data is of large size.

5.5.1 Findings

If the data distribution is stable, mining a data stream is largely same as mining a large dataset, as the size of data is large it requires large computational time which makes the algorithm inefficient. By windowing approach it is not based on sample, thus their is less chances of loss of accuracy and the model is prepared using very small dataset, that reduces the execution time, but if the window is large, it take large dataset, that increases the computational time, thus the intermediate size windowing is suitable for classification and prediction.

5.6 Summary

1. Predictive data modeling for streamed data is presented in this chapter. We
have proposed a 'windowing' technique for mining stream data. The windowing technique is useful to construct classifier model on very large data sets (stream data). A subset of the training set, called window, is chosen at random and a decision tree is constructed using only this window. In windowing technique we have proposed two different methods.

2. Sliding window method.

In sliding window approach for construction of decision tree uses the aggregation value of data, in order to improve the accuracy. Aggregation of data is slided the first contents is removed from the queue and new item is inserted from rear pointer of queue. It uses the time stamps, when the time stamps of the oldest basic window expires, its synopsis is removed, a fresh data is added to the front and the aggregated is incrementally computed. We have also used tilted and Logarithmic time window.

1. Titled time window: If the window is incremented and move onto stopping condition is called tilted time window. Data is aggregated and transferred to higher granularity level for the distant time period, that substantially save the storage space and computational time.

* Natural Tilted-Time Window: It is maintained using circular queue. Each queue has a specific time granularities. Data is transferred over different granularity.

* Logarithmic Tilted-Time Window: This window is basically used in order to reduce the exponential growth. The stream of transaction is broken into batches.
The experimental result shows that it reduces the memory usage approximated 1/2. It depends on the size of windows. In the series of time based data, mining on time series data is discussed in next chapter.