Efficient Decision Tree Construction for Mining Time-Varying Data Streams

Yingying Tao and M. Tamer Özsu
University of Waterloo
Waterloo, Ontario, Canada
{y3tao, tozsu}@cs.uwaterloo.ca

Abstract

Mining streaming data has been an active research area to address requirements of applications, such as financial marketing, telecommunication, network monitoring, and so on. A popular technique for mining these continuous and fast-arriving data streams is decision trees. The accuracy of decision trees can deteriorate if the distribution of values in the stream changes over time. In this paper, we propose an approach based on decision trees that can detect distribution changes and re-align the decision tree quickly to reflect the change. The technique exploits a set of synopses on the leaf nodes, which are also used to prune the decision tree. Experimental results demonstrate that the proposed approach can detect the distribution changes in real-time with high accuracy, and re-aligning a decision tree can improve its performance in clustering the subsequent data stream tuples.

1 Introduction

Traditional DBMSs are successful in many real-world applications where data are modeled as persistent relations. However, for many recent applications, data arrive in the form of streams of elements, usually with high arrival rate. Techniques for traditional database management and data mining are not suited for dealing with such rapidly changing streams and continuous queries that run on them. Examples of such applications include financial marketing, sensor networks, Internet IP monitoring, and telecommunication [4, 5, 22, 23].

In the past few years, significant research has been done in processing and mining data streams, with the goal of extracting knowledge from different subsets of one data set and integrating these generated knowledge structures to gain a global model of the whole data set [8].

Clustering is an important data/stream mining technique. It groups together data with similar behavior Many applications such as network intrusion detection, marketing investigation and data analysis require data to be clustered.

Use of decision trees is one of the most popular clustering techniques. Compared to other clustering techniques such as K-means [11, 15], decision tree models are robust and flexible. There are many decision tree construction algorithms that generally construct a decision tree using a set of data as training examples, where leaf nodes indicate clusters, and each non-leaf node (called a decision node) specifies the test to be carried out on a single-attribute value. New data can be clustered by following a path from the root to one leaf node.

Most decision tree generation algorithms make the assumption that the training data are random samples drawn from a stationary distribution. However, this assumption does not hold for many real-world data stream applications. Typically, fast data streams are created by continuous activities over a long period of time, usually months or years. It is natural that the underlying processes generating them

can change over time, and thus the data distribution may show important changes during this period. Examples include applications for monitoring stock price, network bandwidth usage, foreign currency exchange rate, holiday effect, and so on. This issue is referred to as *data evolution*, *time-changing data*, or *concept-drifting data* [1, 13, 17, 26].

1.1 Motivation

The distribution change over data streams has great impact on most stream mining algorithms. A data stream mining model built previously may not be efficient or accurate when the data evolve, and some characteristics observed before may no longer hold. Hence, techniques for detecting distribution changes in data streams and adjusting the existing decision tree to reflect these changes are required.

A naive extension is to rebuild the decision tree from scratch when a distribution change is detected. Since it may take a long time to recollect training samples for rebuilding a decision tree, this solution is not practical. An alternative solution is to reconstruct the decision tree incrementally, so that the tree can be adaptive to the changes. Some previous approaches adjust an existing decision tree when the distribution of the stream changes by replacing the affected leaf nodes by subtrees to maintain accuracy [10, 13, 14]. However, this type of approach may lead to serious inefficiency in clustering.

Consider a simple scenario. One import company in US uses a data stream management system to monitor all its transactions with Canada. The company wants to monitor its transactions and the exchange rate when each transaction is made. Hence, the decision tree should be built using currency exchange rate as the criteria for each decision node in the tree. Assume the data stream starts at a time when the exchange rate is about 1:1.6 US dollar to Canadian dollar. If the US dollar gets cheaper over time, leaf nodes in the original decision tree will be replaced by sub-trees to reflect this change. Over time, we may have a decision tree similar to the one shown in Figure 1.

Notice the problem here: as the data stream continues, the most recent data will fall in the clusters (leaf nodes) at the lowest level, i.e. the two leaf nodes under decision node "rate >1.2". As the tree gets deeper, clustering gets increasingly inefficient.

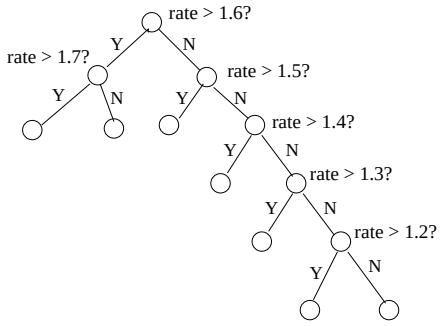


Figure 1: Example of a decision tree based on exchange rates

In this paper, we propose a decision tree-based approach for mining time-changing data streams. The approach can be summarized as follows. We continuously monitor the decision tree, and if it becomes unacceptable (to be defined later), we re-align the tree in real time to improve its efficiency. We maintain a synopsis at each leaf node indicating how many data elements fall in this cluster in certain time periods. If most of the recent data fall in certain leaf nodes (clusters), while other leaf nodes are barely touched, then this could imply a distribution change. Hence, we re-align the tree to reflect this change.

This approach is independent of the underlying tree construction technique. Re-alignment does not affect the accuracy of a decision tree, since we do not modify any gain functions on the decision nodes, while the efficiency will be increased for concept-drifting data. The overhead introduced in our technique is the time to re-align the tree, and the memory used for keeping synopsis on leaf nodes.

1.2 Summary of Contributions

The contributions of this paper are as follows:

- We propose a new method of detecting distribution changes based solely on timestamps and sliding windows; hence, the change can be detected in real-time with very little overhead.
- We propose a decision tree re-aligning technique that is independent of the underlying decision tree construction ap-

proach. Our technique adds adaptivity of changes to the existing decision tree, and can considerably improve the performance of the decision tree when distribution changes.

- Our heuristics-based approach provides a novel way for tree-pruning.
- We can improve any two of the three major criteria for decision trees (accuracy, efficiency, and tree-size) with no impact on the third one.

The rest of the paper is organized as follows. Section 2 discusses our technique for detecting distribution changes in data streams. In Section 3, an algorithm for re-aligning a decision tree is proposed. Section 4 describes an approach that can prune a decision tree effectively. The experimental results are presented in Section 5. In Section 6, we discuss the related work on change-detection and decision tree construction over time-changing streams. We conclude the paper in Section 7.

2 Detecting Changes

A data stream S is an unbounded sequence of elements $\langle s, t\rangle$, where s is data, and $t \in T$ is a monotonically increasing timestamp indicating the arrival time of the element. s can have different forms, such as a relational tuple, or a set of items, depending on the underlying application that generates S.

Let D_s be a decision tree for clustering S. Let d_j $(j = 1, 2, ..., q)$ be the decision nodes in D_s, and c_i $(i = 1, 2, ..., m)$ be the leaf nodes representing different clusters. Each element in S will be clustered by D_s following a certain path constructed by a set of decision nodes and one leaf node. For each leaf node c_i in D_s, we maintain a synopsis containing the following:

- τ_i - The timestamp of the last element that falls in this node; i.e. when a new element $\langle s_k, t_k\rangle$ is placed in c_i, set $\tau_i = t_k$.
- θ_i - Total number of elements that are within this cluster represented by c_i.
- ϕ_i - The average value of timestamps of all elements that fall in c_i. ϕ_i represents the "time density" of c_i. We will use this value for detecting distribution changes in the data stream.

  Since data streams are unbounded, ϕ_i has to be maintained incrementally. Every time a new element $\langle s_k, t_k\rangle$ falls in c_i, we set

  $$\phi_i' = \frac{\phi_i * \theta_i + t_k}{\theta_i'}$$

  where ϕ_i' is the new time density and $\theta_i' = \theta_i + 1$ is the updated total number of elements that are in the cluster represented by c_i.

The full algorithm for detecting distribution changes is given in Algorithm 1.

Each time a new element $\langle s_k, t_k\rangle$ falls in a leaf node c_i, we calculate the timestamp distance $distance_i = t_k - \phi_i$. This value is compared to a threshold value γ. If $distance_i < \gamma$, it means that a large portion of newly arrived elements fall in this cluster, which may imply a distribution change. Hence, we mark this leaf node c_i for re-aligning (lines 19-25 in Algorithm 1). Threshold γ is a predefined value. The larger the γ, the earlier a change in the stream can be detected. However, the risk for false alarms is also higher, that may cause more frequent re-alignment, leading to a higher overhead. In Section 5.2, we will further discuss issues related to setting γ.

For a data stream with very high arrival rate, calculating the timestamp distances of all leaf nodes each time a new element arrives would be expensive and unnecessary. Existing sampling techniques can be adopted for this case. We do not discuss this issue further in this paper.

For a stream that has continued over a long period, historical data can severely affect the accuracy and efficiency of the change detection technique. For example, even when most of the new elements fall in one leaf node c_i, if c_i contains a large number of historical data, its timestamp distance may still be larger than the threshold γ. Therefore, to eliminate the effect of historical data in a cluster, we apply a time-based sliding window R over the data stream S. Window R contains a substream of S that arrives within time $\langle t, t + \Delta\rangle$. Only elements that are within this sliding window are considered in the calculation of ϕ for each leaf node. When a data element $\langle s_k, t_k\rangle$ expires from R as R moves forward, the synopsis for leaf node c_i that $\langle s_k, t_k\rangle$ belongs to isupdated. The new ϕ_i' is updated as $\phi_i' = \frac{\phi_i * \theta_i - t_k}{\theta_i'}$, where $\theta_i' = \theta_i - 1$

Algorithm 1 Detecting changes in data stream

```
1: INPUT: Data stream S
2:        Decision tree D_s
3:        Sliding window R
4: OUTPUT: Modified decision tree D'_s

5: for all new element ⟨s_k, t_k⟩ of S that will fall in leaf node c_j do
6:   if c_j is replaced by a subtree with leaf nodes c'_j and c''_j then
7:     //Set synopsis for c'_j and c''_j
8:     τ'_i = τ''_i = t_k;
9:     θ'_i = θ''_i = 0;
10:    φ'_i = φ''_i = 0;
11:    Set c'_j and c''_j to be unmarked;
12:  else
13:    // Update synopsis
14:    τ_j = t_k;
15:    θ_j ++;
16:    φ_j = (φ_j * θ_j + t_k) / θ_j;
17:    if θ_j > MinClusterSize then
18:      // Check distribution changes
19:      distance_j = t_k − φ_j;
20:      if distance_j < γ then
21:        if the re-aligning mark of c_j is unmarked then
22:          //Start re-aligning
23:          Mark c_i for re-aligning;
24:          Call Algorithm 2;
25:        end if
26:      else if the re-aligning mark of c_i is marked then
27:        Reset c_i to be unmarked;
28:      end if
29:    end if
30:    for all ⟨s_i, t_i⟩ belongs to leaf node c_i that are moved out of R do
31:      //Remove historical data
32:      θ_i − − ;
33:      φ_i = (φ_i * θ_i − t_i) / θ_i;
34:    end for
35:  end if
36: end for
```

is the updated total number of elements that are in the cluster of c_i (lines 30-34).

The time interval Δ of R is a pre-defined value based on application requirements. Some applications, such as network monitoring, may require larger Δ, while other applications, such as time-critical real-time ones, may prefer a smaller Δ. Since, to detect changes, we do not need to store the actual values of the elements within the window, the time interval of R can be very large if needed. We will further discuss issues related to Δ setting in Section 5.2.

If a cluster represented by c_i contains only historical data, i.e., none of the elements it contains is within R, then this cluster will have $\theta_i = 0$ and $\phi_i = 0$. Notice one problem here: when a new element $\langle s_k, t_k \rangle$ falls in c_i, c_i's synopsis will be updated as $\tau_i = t_k, \theta_i = 1$, and $\phi_i = t_k$. Since $distance_i = \phi_i - t_k$ will then be 0, a distribution change will be flagged. This problem is caused by the danger of making a decision with very few samples. The problem arises when a leaf node is replaced by a subtree (as will be discussed later), since new leaf nodes contain no elements yet. To solve this problem, we use a minimum cluster size parameter to indicate the minimum number of data elements that a cluster must contain in order to trigger change detection on this cluster.

Once a leaf node c_i is marked, $distance_i$ may stay below the threshold γ for a while. After a certain time period, there are two possibilities:

- The total number of elements that fall in c_i is very high, in which case c_i will be replaced by a subtree with one decision node and two leaf nodes c'_i and c''_i (as in [6, 10, 14]). We then set $\tau'_i = \tau''_i = t_k, \theta'_i = \theta''_i = 0, \phi'_i = \phi''_i = 0$, where t_k is the timestamp of the last element that was placed in c_i before the replacement. Re-alignment flags for c'_i and c''_i are set to *unmarked* (lines 6-11).
- $Distance_i$ is no longer less than γ. This may indicate that the distribution change has ended, i.e., the new distribution has stabilized. Hence, we reset the re-aligning flags for c_i to *unmarked* (lines 26-27).

Algorithm 1 has complexity $O(m)$ for each new data element, where m is the total number of out-dated data elements (i.e., the elements that move out of R) when a new data element arrives. If the arrival rate of the data stream is stable, then each time a new element arrives only one data element will be removed from R. For this case, the complexity of Algorithm 1 is

$O(1)$. For a data stream with an unstable arrival rate, m can be greater than 1, indicating the arrival rate is decelerating, since there are more out-dated data than new data. Hence, although Algorithm 1 may take longer for this case, it is still practical, because when the arrival rate is lower, clustering process does not need to be that efficient.

3 Tree Re-alignment

The purpose of re-aligning a decision tree D_s is to move more frequently visited leaf nodes higher in the tree. By doing this, the overall efficiency of the clustering process is improved, since most recent elements need to follow shorter paths (i.e. pass fewer decision nodes) to be clustered. For example, for the decision tree shown in Figure 1, recall that most of the recent data elements are in clusters "rate $\leq$ 1.2" and "1.2 $<$ rate $\leq$ 1.3 ". Any element that needs to reach either of the clusters needs to pass 5 decision nodes (including root). Total number of decision nodes for reaching both leaf nodes is 10. However, if we re-align the tree to the form shown in Figure 2, the total number of decision nodes for reaching both of the leaf nodes is 3, with 1 for cluster "rate $\leq$ 1.2" and 2 for cluster "1.2 $<$ rate $\leq$ 1.3". Although to reach the leaf nodes under decision node "rate $>$ 1.7", 6 decision nodes need to be passed, since these leaf nodes are barely visited after the distribution change, the efficiency of this decision tree improves.

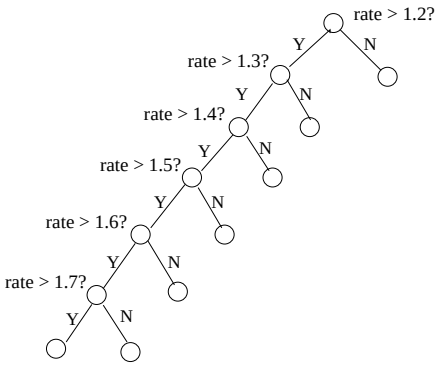


Figure 2: Re-aligned decision tree

The problem of re-aligning D_s can be transformed into a problem similar to the optimal weighted binary search tree construction problem. We can assign each leaf node a weight that reflects the number of recent elements this node has. The higher the weight, the more recent elements this leaf node contains; whereas the lower the weight, the more historical elements are within this cluster. An optimal decision tree with the highest weight can then be constructed using dynamic programming. For the applications where efficiency is the major concern, a suboptimal decision tree re-alignment technique can be applied.

Let p_i be the weight of leaf node c_i ($i = 1, 2, ..., m$). Initialize $p_i = 1$. If a leaf node c_i contains only historical data, i.e. all elements within it are outside R, we reduce p_i by half. Each time R slides through a full length (i.e., every Δ time period) while no new data element arrives in c_i, we continue to reduce its weight p_i by half. Therefore, the smaller is p_i, the "older" is c_i. Figure 3 gives an example of how the weight should be adjusted for an out-dated leaf node over time.

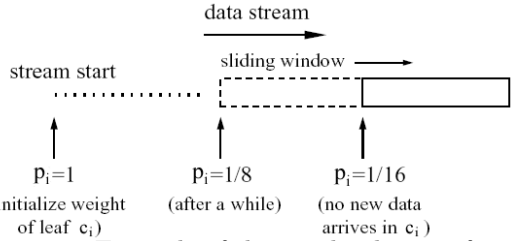


Figure 3: Example of the weight change of one out-dated leaf node

Every time c_i is changed from *unmarked* to *marked*, we increment p_i by 1. By analyzing these weights attached to each leaf node, we can determine which leaf nodes have received recent data, and which have not. Based on different applications, re-alignment strategy can be either eager or lazy. For eager strategy, every time the weight of one leaf node is increased (which indicates a distribution change), the decision tree is re-aligned. We can also apply a lazy strategy where the decision tree will only be re-aligned periodically, or after a certain number of leaf nodes are changed from *unmarked* to *marked*.

Notice that p_i only changes when c_i changes from *unmarked* to *marked*. For any leaf node c_j already *marked*, its weight will not increase every Δ time period, and the re-alignment procedure will not be invoked. This is because once a distribution change starts, it may take

a while until the new distribution stabilizes. Hence, it takes time for $distance_j$ to become greater than γ again, or until c_j is replaced by a subtree, as discussed in Section 2. There is no need for re-aligning the tree before the new distribution stabilizes. If c_j is replaced by a subtree with new leaf nodes c'_j and c''_j, then we set $p'_j = p''_j = p_j$, but do not start re-aligning the new decision tree immediately.

Definition 1. For two decision trees D_s and D'_s with decision nodes d_j, d'_j $(j = 1, 2, ..., q)$ and leaf nodes c_i, c'_i $(i = 1, 2, ..., m)$, respectively, we say D_s and D'_s are *functionally equivalent* if and only if:

- D'_s is constructed using exactly the same decision nodes and leaf nodes as D_s, i.e., $\{d_1, ..., d_q\} = \{d'_1, ..., d'_q\}$ and $\{c_1, ..., c_m\} = \{c'_1, ..., c'_m\}$, and
- if a data element $\langle s_k, t_k \rangle$ in S, if it falls in leaf node c_i following D_s, then it will fall in the same leaf node following D'_s.

Functionally equivalent trees have the same number of decision nodes and leaf nodes, and the gain functions on decision nodes are identical. They produce exactly the same results for clustering a data stream. However, their efficiency may be different. Our goal for tree re-alignment is to find the most efficient functionally equivalent tree to the current decision tree D_s. To measure the efficiency of a decision tree, we introduce the concept of the *weight* for decision trees.

Definition 2. Let H be the depth of D_s and h_i be the depth of c_i, i.e. the number of decision nodes an element needs to pass before it reaches c_i. We define the *weight* (W) of decision tree D_s as:

$W = \sum_{i=1}^{m}(p_i \times (H - h_i + 1))$

$H - h_i + 1$ is the level of leaf node c_i counting bottom-up. For two leaf nodes c_i and c_j with the same weight, if $h_i < h_j$ (i.e. c_i is at a higher level than c_j), we have $(p_i \times (H - h_i + 1)) > (p_j \times (H - h_j + 1))$. Traditionally, the level of a node is counted top-down. However, in our case, because our goal is to push leaf nodes with higher weight to a higher level, the level is assigned in reverse.

Given two functionally equivalent trees D_s and D'_s with weights W and W', respectively, if $W > W'$, it may imply that leaf nodes with higher weights (i.e., most frequently visited) are aligned at higher levels in D_s than in D'_s. Hence, our goal is to find an equivalent tree of D_s with the highest weight, i.e. the tree D^w_s where

$W^w = \sum_{i=1}^{m}(p_i * (H - h^w_i + 1)) = max(W)$.

It is possible that two functionally equivalent trees have the same weight. To break the tie, we choose the tree with a lower depth H.

For data streams that do not have very high arrival rate and do not require frequent re-alignment (i.e. γ is not set to be very large), we can apply dynamic programming to find optimal weighted binary trees [18, 19]. It has been proven that this dynamic programming approach for finding optimal weighted binary tree has time complexity $O(n^2)$, where n is the total number of nodes in D_s. This dynamic programming approach has been shown to admit an implementation running in $O(n \log(n))$ time [19].

However, for the high speed data streams and streams that change distribution frequently, the quadratic time complexity may not be practical. Thus, we introduce a sub-optimal decision tree re-alignment algorithm derived from [20, 21][1]. This algorithm can find an approximat optimal weighted binary tree with the complexity $O(n \log(n))$.

Let c_i be a leaf node of D_s, and d_j and d'_j be its parent and grandparent nodes. Let c'_i be the direct child of d'_j. That is, c'_i is one level higher than c_i. Starting from the leaf node on the lowest level and going bottom-up, we apply the following heuristics to each leaf node c_i $(i = 1, 2, ...m)$ in D_s.

Heuristics H1: If the weight of c_i is greater than the weight of c'_i, i.e. $p_i > p'_i$, then exchange the position of d_j and d'_j in D_s by performing a single rotation.

By applying this heuristics, c_i along with its parent d_j will be moved to a higher level than c'_i and d'_j, and hence, the weight of the new tree is greater than D_s. This heuristic is repeatedly applied until all the leaf nodes are examined.

Note that the resulting decision tree with a

[1] An algorithm introduced in [25] can further reduce the time complexity to linear, i.e. $O(n)$, however, since the resulting tree may not be efficient in many cases, this algorithm should only be applied when the speed of the data stream is extremely high.

higher overall weight than the original tree may not be a balanced one. A balanced tree may not be a good solution for many data streams with distribution changes. According to our weight function, the leaf nodes on very low levels should have low weights, meaning that their clusters haven't received new data for a long time. Hence, although visiting these leaf nodes may be inefficient, since they are barely visited, the overall efficiency should not be affected severely. Furthermore, we can apply the pruning method (introduced in the next section) to reduce the tree depth by removing these historical clusters.

The full algorithm for assigning weights and eagerly re-aligning the tree is shown in Algorithm 2.

Algorithm 2 Decision tree re-aligning

```
1: INPUT: Decision tree D_s
2: OUTPUT: Re-aligned decision tree D'_s

3: for all leaf node c_i in D_s do
4:    //Initialize weights
5:    p_i = 1;
6:    Call change detection algorithm;
7:    if c_i contains only historical data then
8:       p_i = p_i/2;
9:    else if the re-aligning mark of c_i is set
      from unmarked to marked then
10:      p_i = p_i + 1;
11:      //Start re-aligning D_s
12:      for all leaf node c_k in D_s starting from
         the lowest level do
13:         Find its parent d_j and grandparent
            d'_j;
14:         Find d'_j's direct child c'_k
15:         if p_k > p'_k then
16:            Exchange d_j and d'_j with a single
               rotation;
17:         end if
18:         Move to the leaf node at one level
            higher;
19:      end for
20:   end if
21: end for
```

4 Pruning Decision Tree

There are three major criteria for evaluating decision trees: accuracy, efficiency, and tree size. As we mentioned in Section 1, our approach improves the efficiency of a decision tree for concept-drifting data without affecting its accuracy. In this section, we discuss how the tree size can be reduced effectively by pruning outdated nodes using the synopsis.

Most of the decision tree generation and modification techniques over data streams proposed in literature do not consider the tree size as long as the tree can fit in main memory. Consequently, the most popular solution is to start pruning only when a decision tree is too large to fit in memory. This is not necessarily a good strategy. Since historical data, e.g. data that arrived months or years ago, may no longer be useful after the distribution of the stream changes, we can delete clusters containing only out-dated data even when memory is sufficient. This early-pruning can result in a reduction in the size and the depth of the tree, leading to lower overall cost for clustering.

Furthermore, scant attention has been paid so far to the actual pruning algorithm. One common approach is to remove leaf nodes that contain few elements. This approach may not be appropriate for data streams that change over time. For example, assume one leaf node c_1 has more elements than c_2. Based on the common less-elements-pruned-earlier strategy, c_2 will be removed. However, for a data stream whose distribution changes over time, it is possible that most of the elements in c_1 arrive long time ago, while new data are falling in c_2. In this case, a better solution is to prune c_1, since it is less important to the current distribution.

Based on this insight, we propose two heuristics for pruning a decision tree using the synopsis we presented in Section 2 and the weights introduced in Section 3.

Heuristics H2: Prune leaf nodes with $\theta \neq 0$ and *distance* greater than a certain threshold.

Recall that θ is the number of elements that fall in the cluster represented by a leaf node. The greater the *distance*, the "older" the leaf node is. Hence, by pruning leaf nodes with *distance* greater than a certain threshold, historical clusters are deleted from the tree. The appropriate threshold setting is application dependent.

Heuristics H3: Prune leaf nodes with weight p less than a certain threshold ω ($0 \leq$

$\omega \leq 1$).

The higher is ω, the more leaf nodes will be pruned. For example, if ω is set to 1/2, then the first time one leaf node is considered out-dated, it will be pruned immediately. If ω is 1/4, then it takes double the time to make this pruning decision. Hence, when the system resources are limited, ω should be a higher value, whereas if resources are abundant, ω could be smaller.

H2 can be regarded as an eager pruning strategy, and H3 is a lazy pruning strategy. For different scenarios, these heuristics can be re-evaluated:

1. every time a new element arrives, when *distance* and p values of all leaf nodes are recalculated, or
2. when the size, depth, or weight W for D_s is less than a predefined threshold, or
3. at certain time intervals, or
4. when memory is insufficient.

After several leaf nodes are deleted, the decision nodes connected to them will miss one or both of their children. Usually, a decision tree is a full binary tree, and to maintain this property, we consider two cases:

- **Case 1:** one child of a decision node d_a is deleted after pruning.

  Let the children of d_a be c_a and t_a (t_a can be a subtree or a leaf node). Let d'_a be the parent of d_a. Assume c_a needs to be pruned according to H2 or H3. If d_a is the root node (i.e. $d'_a = NULL$), then remove d_a and set the root of t_a to be the new root (illustrated in Figure 4(a)). Otherwise, set the root of t_a to d'_a, and remove d_a (illustrated in Figure 4(b)).

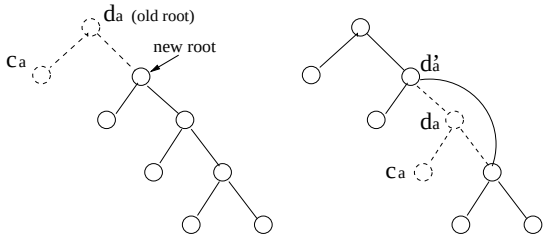


Figure 4: Case 1: one child of d_a is pruned

- **Case 2:** both children of a decision node d_a are deleted after pruning.

  Let the children of d_a be c_a and c'_a. If c_a and c'_a are to be deleted, replace d_a by a new leaf node c_b. Set its synopsis as $\tau_b = max(\tau_a, \tau'_a), \theta_b = 0, \phi_b = 0$. The weight p_b of new leaf c_b is reset to 1.

The total cost of the new decision tree may not be optimal after pruning. However, since pruning process only modifies historical leaf nodes, recently visited leaf nodes remain on their current levels. Hence, it is not necessary to re-align the tree after pruning.

The full algorithm for pruning a decision tree is given in Algorithm 3. Algorithm 3 has complexity $O(m)$ upon each re-evaluation, where m is the total number of leaf nodes in D_s.

5 Experiments

In this section, we present a series of experiments to evaluate the performance of the proposed techniques. All experiments are conducted on a PC with 3GHz Pentium 4 processor and 1GB of RAM, running Windows XP. All algorithms are implemented in C.

For each data stream used in the experiments, we generate its decision tree using the CVFDT algorithm [13]. A decision tree generated by CVFDT algorithm is adaptive to distribution changes in the stream by replacing leaf nodes with subtrees. In [13], CVFDT is evaluated with categorical attributes only. Hence, we additionally implemented the technique presented in [14], so that numerical attributes can be used as classifiers on the decision tree.

For a decision tree using numerical attributes as classifiers, when one leaf node contains too many data elements and needs to be split, it is not clear to see what the splitting point (i.e., the numerical value on the decision node of the new subtree that replaces this leaf node, see Figure 1 for example) is. In such cases, we apply a gain function to find the best splitting point. The gain function is constructed using Hoeffding bound [12]. In our experiments, for each leaf node, we start the splitting procedure after at least 5,000 data elements fall in it. The α value used to calculate Hoeffding bound is set to $1 - 10^{-6}$. These two parameters are set exactly the same as in the experiments of [14].

Algorithm 3 Pruning a decision tree

```
1: INPUT: Decision tree D_s
2: OUTPUT: Modified decision tree D'_s

3: if re-evaluation start then
4:   for all leaf node c_a in D_s do
5:     if c_a satisfies H2 or H3 then
6:       //Depend on which heuristic is adopted
7:       //H1 and H2 cannot be used at the same time
8:       Find c_a's direct parent d_a;
9:     end if
10:    if d_a is the root of D_s then
11:      Set d_a's another child d'_a as new root;
12:      Remove c_a and d_a;
13:    else if d_a's another child is a leaf node c'_a then
14:      Create a new leaf node c_b;
15:      τ_b = max(τ_a, τ'_a);
16:      θ_b = 0;
17:      φ_b = 0;
18:      p_b = 1;
19:      Replace d_a with c_b;
20:      Delete c_a and c'_a;
21:    else
22:      //d_a's another child is a subtree
23:      Find d_a's another direct child d_b;
24:      Find d_a's direct parent d_c;
25:      Set d_b's direct parent to d_c;
26:      Delete d_a and c_a;
27:    end if
28:  end for
29: end if
```

5.1 Change Detection Evaluation

To evaluate the effectiveness of our change detection technique, we adopt the same data streams as used in the experiments of [17], and compared our experimental results with their results[2]. Each data stream contains 2,000,000 points with only one numerical attribute for each data element. The distribution changes occur every 20,000 points. Hence, there are 99 true changes in each data stream.

The arrival speed of this data stream is set to be stable, with one tuple per unit time. This is for the purpose of gaining control over the size of the sliding window R, since a time-based sliding window will be equal to a count-based one if the speed of the stream is stable. However, keep in mind that our techniques do not require the data stream to have an even speed. In these experiments, the time interval Δ of R is set to 500 time units. The minimum cluster size is 100 data elements. We set $\gamma = 70$ time units. The effect of Δ and γ settings on our proposed technique is studied in the next section.

The experimental results are shown in Table 1. If, at the time when a change is reported, the change point (i.e. the first element that belongs to the new distribution) is still in R, or it was contained in R at most Δ time units ago, then this detection is considered to be on time. Otherwise it is considered late. The results are reported in the form a/b where a is the number of changes detected on time, and b is the number of changes reported late.

In Table 1, "Γ" represents the results of the proposed change detection technique. "Wil" is the technique using Wilcoxon statistic. "KS" is the technique with Kolmogorov-Smirnov statistic over initial segments "KS". "KSI" is the technique using Kolmogorov-Smirnov statistic over intervals "KSI". "Φ" and "Ξ" represent the results of using ϕ_A and Ξ_A statistics proposed in [17] (where A is the set of initial segments), respectively. For detailed description on these techniques, the readers are referred to [17].

S_1 is a data stream whose initial distribution is uniform. The starting distribution of stream S_2 is a mixture of a Standard Normal distribution with some Uniform noises. Streams S_3, S_4, S_5 and S_6 contain Normal, Exponential, Binomial and Poisson distributions, respectively.

These results lead to the following observations:

- The distribution changes are usually detected on time using the proposed technique. For other techniques, the change detections are more likely to be delayed

[2]There are two versions of each experiment in [17], each using a different parameter called critical region defined in their paper. This parameter does not affect our techniques. According to [17], the results of the version using critical region $(20k, .05)$ are usually the better ones, thus, we only compare our results with this version.

Table 1: Number of distribution changes detected using different techniques

	Γ	Wil	KS	KSI	Φ	Ξ
S_1	42/17	0/5	31/30	60/34	92/20	86/19
S_2	40/9	0/2	0/15	4/32	16/33	13/36
S_3	46/13	10/27	17/30	16/47	16/38	17/43
S_4	73/6	12/38	11/38	7/22	7/29	11/46
S_5	63/0	36/42	24/38	17/22	12/32	23/33
S_6	61/2	36/35	23/30	14/25	14/21	23/22

for all testing streams except S_1.

This is because other techniques need to see the "big picture" of the stream data distribution in order to detect changes, while our technique can quickly locate the clusters (leaf nodes) where changes start without waiting until the new distribution is fully visible.

- Our technique performs much better than others for streams S_4, S_5 and S_6. Furthermore, for these streams, most of the distribution changes are detected on time.

  S_4 has exponential distribution and S_5 and S_6 have discrete distributions. In all three cases, the distribution changes are severe, i.e. the new distribution is considerably different than the old one. These results indicate that our technique performs best for data streams that may have severe distribution changes (such as detecting fraud, instrument failure, and virus attacks).

- For data streams S_1, S_2 and S_3, our technique may not perform better than other techniques on the total number of changes detected (even slightly worse in a few cases). This is because these three streams have relatively slow and smooth distribution changes. For these cases, we can improve the effectiveness of our technique by increasing γ. However, this increases the chance of false detection, as will be shown in next section.

To evaluate the false detection rate, we run all six techniques on five streams with 2,000,000 points each and no distribution change. The results of total number of false detections on all five streams are shown in Table 2. From these results, we can see that our technique has a lower false detection rate than most of other techniques.

We do not show experimental results for run time comparison of these techniques. As mentioned in Section 2, our change detection algorithm has a worst case complexity of $O(m)$, where m is the number of out-dated data when a new data arrives. Usually m is not a large value unless there is a severe burst in the stream. For data streams with steady arrival rate, the time complexity of our algorithm is $O(1)$. KS and KSI have $O(log(m_1 + m_2))$ time complexity [17], where m_1 and m_2 are the sizes of a pair of sliding windows. The complexity of computing Wilcoxon is $O(m_1 + m_2)$, and the computation time for ϕ_A and Ξ_A is $O([m_1 + m_2]^2)$. Notice that multiple pairs of sliding windows are used in [17] with different sizes, and some pairs may have very large window sizes in order to detect smaller changes that last over long time. Hence, the time complexity of our technique is better than these techniques.

Table 2: Number of false alarms

	Γ	Wil	KS	KSI	Φ	Ξ
False alarms	26	40	40	49	18	36

5.2 Varying Distance Threshold and Sliding Window Interval

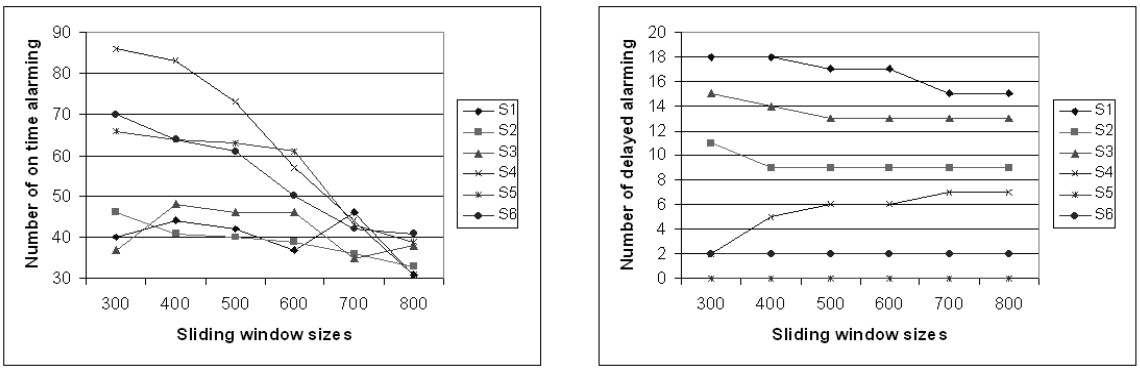
In order to analyze the effect of different Δ and the distance threshold γ settings on change detection, we conduct a set of experiments on the six data streams $S_1 - S_6$ using various settings:

1. Keep $\gamma = 70$ time units unchanged, adjust Δ from 300 time units to 800 units by increasing Δ 100 time units each time. Figures 5(a) and 5(b) show the results of the number of changes detected on time and delayed, respectively.

2. Fix Δ to 500 time units, and vary the value of γ from 30 to 150 time units in increment of 20 units. Figures 6(a) and 6(b) show the results of the number of changes detected on time and delayed, respectively.

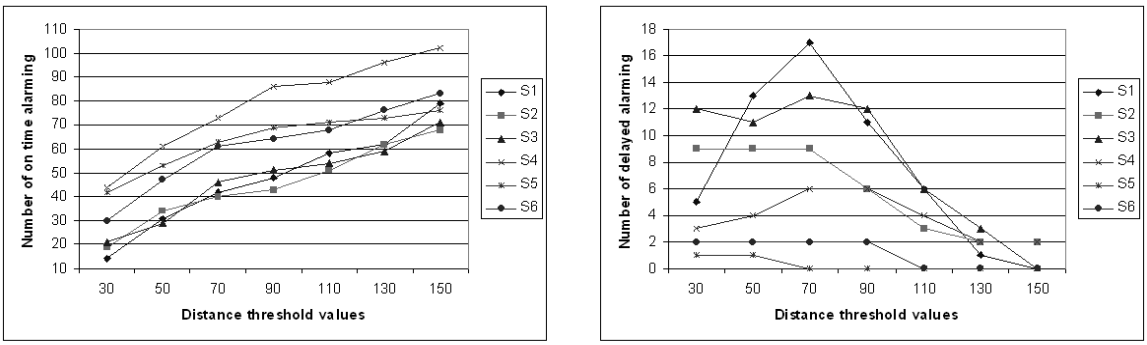
From these results, we have the following observations:

- Increasing Δ may result in a reduction in the number of on time change detections. This is because, as Δ increases, more "old" data are involved in calculating the distance. Hence, it is harder to have the distance less than threshold γ. However, because the definition of "on time" in Section



(a) (b)

Figure 5: Performance of our change detection technique with Δ varied



(a) (b)

Figure 6: Performance of our change detection technique with γ varied

5.1 depends on Δ, a larger Δ implies a larger R, and thus, the number of changes detected on-time may increase.

- Larger γ values may increase the number of changes detected on time. However, notice that some of the distribution changes detected may be false.
- It is unclear how the number of delayed change detections will vary when adjusting Δ or γ.

We also run experiments studying how Δ and γ can affect the false alarm rate, using the same five data streams used in Section 5.1. The results are shown in Tables 3 and 4. From these results, we see that decreasing Δ or increasing γ may lead to a high false alarm rate.

Table 3: Number of false alarms changing Δ

Δ	300	400	500	600	700	800
False alarms	30	26	26	21	12	10

Table 4: Number of false alarms when γ varies

γ	30	50	70	90	110	130	150
False alarms	11	15	26	28	32	43	49

5.3 Efficiency Comparison of Realigned Decision Tree and Original Tree

To verify the effectiveness of the proposed decision tree re-aligning technique, we apply both the change detection algorithm (Algorithm 1) and the decision tree re-aligning algorithm (Algorithm 2) on the six data streams $S_1 - S_6$ using the same parameter and threshold settings as in Section 5.1. The efficiency of the original decision trees D_i $(i = 1, ..., 6)$ and the re-aligned trees D_i' is measured using the weights W_i and W_i', as described in Section 3. For each data stream S_i, every time a change is detected, we re-align the decision tree and record the ratio

$r_j = W_i'/W_i$, where j indicates the j^{th} change being detected. Notice that although the original tree D_i does not change during processing, each time we calculate W_i', W_i also needs to be updated, because the weights attached to some leaf nodes may have changed. The average ratio $Avg(W_i'/W_i) = \frac{r_1+r_2+...+r_q}{q}$ (where q is the total number of changes detected) is used to estimate the overall efficiency improvement of our decision tree re-aligning algorithm on each data stream.

The results are shown in Table 5. We can see that the efficiency of each decision tree is greatly improved after re-aligning.

Table 5: Efficiency improvement of our decision tree re-aligning algorithm

S_i	S_1	S_2	S_3	S_4	S_5	S_6
$Avg(W_i'/W_i)$	4.67	3.65	7.01	8.69	7.10	4.72

5.4 Performance comparison of Optimal and Sub-optimal Tree Re-aligning Strategies

In Algorithm 2 for decision tree re-aligning, we adopted an approximate algorithm that generates a sub-optimal tree. This is for the purpose of increasing the efficiency of decision tree re-aligning process for streams with high arrival rates. However, as mentioned in Section 3, if the speed of the data stream is not extremely high and the re-aligning process is not triggered frequently, we can apply the dynamic programming approach to generate an optimal tree. To compare the performance of both sub-optimal and optimal re-aligning approaches, we implemented the dynamic programming approach for tree re-aligning, and applied it on streams $S_1 - S_6$. The performance and time comparison results are shown in Tables 6 and 7.

Table 6: Efficiency improvement of the re-aligned tree using dynamic programming

S_i	S_1	S_2	S_3	S_4	S_5	S_6
$Avg(\frac{W_i'}{W_i})$	5.29	4.13	7.66	11.00	8.83	5.56

Table 7: Time comparison of dynamic programming (DP) and sub-optimal (Sub) approach

Data stream S_i	S_1	S_2	S_3	S_4	S_5	S_6
DP (time unit)	313	269	370	575	661	396
Sub (time unit)	14	21	32	27	30	13

By comparing tables 5 and 6, we can see that the average performance increment of the optimal decision tree over the sub-optimal tree is 17%. On the other hand, the sub-optimal re-aligning approach is 19.92 times faster than the dynamic programming on average.

5.5 Pruning Heuristics Evaluation

To evaluate the power of our pruning heuristics, we generated a data stream that has only one numerical attribute. Data arrive at a rate of one element per time unit. The value of the data element grows over time. Hence, the decision tree will grow increasingly deeper as leaf nodes keep splitting, and there will be a large number of historical clusters. The pruning procedure is triggered when the height of the decision tree is greater than a threshold (set to 12 in this experiment). Δ is set to 500 time units. For heuristic H2, we set the distance threshold to 1500 time units. For heuristic H3, when the weight of one leaf node is less than 1/16, this leaf node will be pruned. Table 8 demonstrates the results of our pruning procedure using H2 and H3, respectively.

Table 8: Pruning results using H2 and H3

Heuristic	# of nodes before pruning	# of nodes after pruning	tree height after pruning
H2	35	15	6
H3	35	21	8

From these results, we can see that the tree size is greatly reduced after pruning. Note that H2 (eager pruning) usually prunes more nodes than H3 (lazy pruning). Which heuristics to choose is an application based decision.

5.6 Running on Real Streams

All experiments we conducted so far use synthetic data streams. To test the performance of our proposed techniques in practice, we design a set of experiments using a real data stream.

The data set we use is called Tao [16]. It is a copyrighted data set from the Tropical Atmosphere Ocean (TAO) project. Detailed information about this project and the Tao data can be found in [24]. Tao data records the sea surface temperature (SST) from past years.

Tao data contains 12218 streams each one of length 962. Since the streams are too short to

fit our experiments, we concatenated them in the ascending order of the time these streams are recorded. This is a reasonable modification because each stream represents the sea surface temperature for a certain period, thus the concatenation represents a record for a longer period.

The arrival speed of Tao is set to one tuple per time unit. The decision tree is built using temperature value as classifier. Minimum cluster size is 100 data elements. Δ and γ is set to 500 and 50 time units, respectively.

Experimental results show that a total of 2746 distribution changes were detected using the proposed change detection technique, with 2173 on time and 573 delayed. The average efficiency improvement after applying our decision tree re-aligning technique is 8.68. This result shows that our approach is effective on real data sets.

6 Related Work

Detecting changes in data streams and adjusting stream mining models accordingly is a challenging issue and has only been recognized as an important one in the past few years.

Aggarwal addressed the data stream change detection problem by providing a framework that uses a concept called *velocity density estimation* [1, 2]. The idea is to estimate the density of the data periodically using a kernel function. Then by estimating the rate at which the changes in the data density occur, the user will be able to analyze the changes in the data over different time horizons.

Kifer et al. present another approach for change detection using two sliding windows over a data stream [17]. They test distributions P_1 and P_2 for the sample data sets S_1 and S_2 in each window, respectively. By comparing P_1 and P_2, it is possible to tell if S_1 and S_2 are generated by the same distribution, i.e. $P_1 = P_2$. If $P_1 \neq P_2$, it indicates that a distribution change has occurred. A new family of distance measures between distributions, and a meta-algorithm for change detection are proposed based on this idea.

These two approaches look at the nature of the data in the stream. They are general, and can be applied to any stream mining model. However, these approaches have high computational cost, and thus may not be suitable for real-time clustering. Moreover, they do not provide guidance for adjusting the stream mining technique to reflect the changes. Our techniques not only detect the distribution changes, but also provide the ability to adjust the mining models to reflect these changes. Furthermore, as we have discussed in Section 5.1, a change detected using these general techniques is more likely to be delayed, since these techniques need to wait until the "big picture" of the stream is clear, while our technique usually can report changes on time.

Aggarwal et al. introduce a change detection approach for the stream mining models using K-means [3]. The idea is to periodically store summary statistics in snapshots. By analyzing these statistics, one can have a quick understanding of current clusters. However, if data is evolving, the model has to be revised off-line by an analyst. Besides, K-means suffers from the well-known problems such as fixed number of clusters, high computational cost, etc. Our approach is based on a more flexible stream mining model (decision trees), and the model can be revised online automatically when a change is detected.

For stream mining models using decision trees, a common solution for detecting and handling time-changing data is to recalculate the gain/loss periodically for a newly arrived data set using the existing classifier. If this gain/loss exceeds a threshold value, it is accepted that the distribution of the data has changed and that the old model no longer fits the new data [7, 9]. When a distribution change occurs, one solution is to rebuild the tree. However, rebuilding has a large overhead and for a data stream with high arrival rate, it may not be feasible for high speed data streams.

Hulten et al. propose the CVFTD algorithm [13] based on their well-known decision tree building algorithm VFDT [6]. CVFTD reflects the distribution changes in real-time by replacing leaf nodes that seem to be out-of-date with an alternate subtree. Gama et al. point out [10] that CVFTD algorithm cannot deal with numerical attributes and propose an algorithm to extend the CVFTD system. Jin and Agrawal present another approach for con-

structing a decision tree that can handle numerical attributes [14]. In their approach, the decision tree is also constructed by repeatedly replacing leaf nodes with subtrees. However, these approaches may lead to an inefficient tree for time-varying data streams. As shown in our experiments, the re-aligned decision tree using our technique can be 4 or even 11 times more efficient than the tree produced by these approaches.

7 Conclusion

In this paper, we propose a new technique for detecting distribution changes over continuous data streams. We use timestamps of each arriving data element as guidance and maintain a synopsis for each leaf node (cluster) in a decision tree. For each cluster, we use its synopsis to tell if this cluster is receiving new data or it contains only out-dated data. If we notice that most of the recent data elements are falling in a small number of clusters, a distribution change is detected.

To improve the efficiency of the decision tree, we propose a tree re-aligning algorithm. This algorithm generates a decision tree that provides the same clustering results as the original tree. Since the frequently visited leaf nodes are moved closer to root, this re-aligned tree is more efficient over the current distribution.

We further propose two heuristics for pruning a decision tree. Our heuristics are based on the common knowledge that clusters with only out-dated data are less important, and thus can be safely removed. This point of view is different than the traditional pruning methods, and serves time-changing data streams better.

Experiments verify the feasibility of our approach. According to the results, our change detection technique can report most of the distribution changes in real time. The decision tree re-aligning technique can improve the efficiency by at least a factor of 4. It is also shown that the proposed techniques can be applied to real data streams with good performance.

We are currently investigating a number of issues, including finding more efficient decision tree re-aligning algorithms (although they may not generate an optimal decision tree), designing more useful synopsis, analyzing how the threshold values should be set for different applications, and developing good sampling technique for stream with extremely high arrival rate.

About the Author

Yingying Tao is a Ph.D student in Cheriton School of Computer Science at the University of Waterloo. She received her M.Sc degree at University of Michigan. Her research interests include data stream management, data mining, and query optimization.

M. Tamer Özsu is a Professor and Director of Cheriton School of Computer Science at the University of Waterloo. His research interests are in database systems, focusing on distributed data management, multimedia data management and XML.

References

[1] C. Aggarwal. A framework for diagnosing changes in evolving data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 575–586, 2003.

[2] C. Aggarwal. On change diagnosis in evolving data streams. *IEEE Trans. Knowledge and Data Eng.*, 17(5):587–600, 2005.

[3] C. Aggarwal, J. Han, J. Wang, and P. Yu. A framework for clustering evolving data streams. In *Proc. 29th Int. Conf. on Very Large Data Bases*, pages 81–92, 2003.

[4] B. Babock, S. Babu, M. Datar, R. Motiwani, and J. Widom. Models and issues in data stream systems. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 1–16, 2002.

[5] M. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *1st Symposium on Network Systems Design and Implementation*, pages 309–322, 2004.

[6] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proc. 6th ACM*

SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, pages 71–80, 2000.

[7] W. Fan, Y. Huang, and P. Yu. Decision tree evolution using limited number of labeled data items from drifting data streams. In *Proc. 2004 IEEE Int. Conf. on Data Mining*, pages 379–382, 2004.

[8] M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining data streams: A review. *ACM SIGMOD Record*, 34(2):18–26, 2005.

[9] J. Gama, P. Medas, and P. Rodrigues. Learning decision trees from dynamic data streams. In *Proc. 2005 ACM Symp. on Applied Computing*, pages 573–577, 2005.

[10] J. Gama, R. Rocha, and P. Medas. Accurate decision tree for mining high-speed data streams. In *Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 523–528, 2003.

[11] S. Guha, A. Meyerson, N. Mishra, and R. Motwani. Clustering data streams: Theory and practice. *IEEE Trans. Knowledge and Data Eng.*, 15(3):515–528, 2003.

[12] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58:18–30, 1963.

[13] G. Hulten, L. Spencer, and P. Domingos. Mining time-chaning data streams. In *Proc. 7th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 97–106, 2001.

[14] R. Jin and G. Aggrawal. Efficient decision tree constructions on streaming data. In *Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 571–576, 2003.

[15] L. Kaufman and P. Rousseeuw. *Finding groups in data: An introduction to cluster analysis*. Addison-Wesley, 1990.

[16] E. Keogh and T. Folias. The ucr time series data mining archive. *http://www.cs.ucr.edu/eamonn/TSDMA/index.html*.

[17] D. Kifer, S. Ben-David, and J. Gehrke. Detecting change in data streams. In *Proc. 30th Int. Conf. on Very Large Data Bases*, pages 180–191, 2004.

[18] D. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.

[19] D. Knuth. *The art of computer programming 3: Sorting and searching*. Addison-Wesley, 1973.

[20] L. Larmore. A subquadratic algorithm for constructing approximately optimal binary search trees. *Journal of Algorithms*, 8:579–591, 1987.

[21] C. Levcopoulos, A. Lingas, and J. Sack. Heuristics for optimum binary search trees and minimum weight triangulation problems. *Theoretical Computer Science*, 66(2):181–203, 1989.

[22] J. Li, D. Maier, K. Tufte, V. Papadimos, and P.Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 311–322, 2005.

[23] Abadi et. al. The design of the borealis stream processing engine. In *Proc. 2nd Biennial Conf. on Innovative Data Systems Research*, pages 277–289, 2005.

[24] Pacific Marine Environmental Laboratory. Tropical automosphere ocean project. *http://www.pmel.noaa.gov/tao/*.

[25] E. Reingold and W. Hansen. *Data structures*. Little Brown and Company, 1983.

[26] H. Wang, W. Fan, P. S. Yu, and J. Han. Mining concept-drifting data streams using ensemble classifiers. In *Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 226–235, 2003.