A Prototype-driven Framework for Change Detection in Data Stream Classification

Hamed Valizadegan
Computer Science and Engineering
Michigan State University
valizade@cse.msu.edu

Pang-Ning Tan
Computer Science and Engineering
Michigan State University
ptan@cse.msu.edu

Abstract- This paper presents a prototype-driven framework for classifying evolving data streams. Our framework uses cluster prototypes to summarize the data and to determine whether the current model is outdated. This strategy of rebuilding the model only when significant changes are detected helps to reduce the computational overhead and the amount of labeled examples needed. To improve its accuracy, we also propose a selective sampling strategy to acquire more labeled examples from regions where the model's predictions are unreliable. Our experimental results demonstrate the effectiveness of the proposed framework, both in terms of reducing the amount of model updates and maintaining high accuracy.

1. Introduction

Data stream classification [1,3,4,5,6] has attracted considerable interest among researchers because of its wide range of applicability, from network traffic analysis to the mining of video streams. However, characteristics of the data stream present several technical challenges that make conventional classification algorithms ineffective. First, it is infeasible to store and process the entire data stream in memory due to its high volume. Second, it is insufficient to build a static model to classify the entire data due to the presence of concept drifts.

Many works in data stream classification are therefore focused on developing models that adapt to changes in the data and are applicable in real time [3,4,6]. A key issue in the design of such an algorithm is the *model maintenance* problem. More specifically, the algorithm must consider the following two questions: (1) How often should the model be updated? (2) Which portion of the data stream should be used to create the new training set for model rebuilding?

There are several approaches employed by current data stream classification algorithms to address the first question:

1. **Continuous Update.** In this approach, the model is incrementally updated every time a new labeled example arrives. Recent studies have shown that such incremental learning algorithms [9, 15] do not perform quite as well as algorithms that periodically rebuild their models from scratch [4] on evolving data streams because of their tendency to retain information from outdated history.
2. **Periodic Update.** This approach partitions the data stream into disjoint time windows and rebuilds the model at the end of each time window [3, 4].
3. **Deferred Update**. This approach rebuilds the model only when there are significant changes in the distribution of data. It requires a change detection mechanism to determine whether the current model is obsolete.

The second question deals with the issue of creating the appropriate training set to ensure that the revised model is accurate in spite of concept drifts and memory limitations. In [4], Aggarwal et al. introduced the notion of a time horizon, referring to the earliest time period from which the labeled examples are selected for training. We consider this approach as selective sampling across time. To enhance the performance of the model, it would be useful to incorporate more labeled examples from regions that are hard to classify—a strategy adopted by boosting and active learning algorithms. We consider this approach as selective sampling across space. As far as we can tell, none of the existing data stream classification algorithms employ such a strategy for their training set creation.

In this paper, we introduce a new framework for data stream classification that uses deferred update with selective sampling across space as its model maintenance strategy. The data stream is partitioned into disjoint time windows and change detection is performed at the end of each time window. Model rebuilding is triggered only when significant changes are detected. The deferred update strategy helps to reduce the computational overhead of periodic updates and the amount of labeled examples needed. In the worse-case scenario, for rapidly evolving data streams, the number of times the model is revised is the same as that for the periodic update approach [2, 3, 8]. Furthermore, the sampling across space strategy helps to ensure that the model maintains a high accuracy despite performing less updates. The key features of our algorithm are as follows:

1. It utilizes both labeled and unlabeled data. The input space is partitioned into several clusters, represented by their corresponding prototypes. Our algorithm determines whether the current model is outdated by comparing the class distribution of labeled examples associated with each cluster against the class label predicted by the model.

2. The cluster prototypes also help to identify regions in the input space where the model's predictions are unreliable. If the model needs to be revised, a selective sampling procedure is triggered to acquire more labeled examples from these hard to classify regions.
3. The framework decouples change detection from the model rebuilding procedure. As a result, it is applicable to any combination of clustering and classification algorithms (unlike some of the previous approaches which are generally restricted to certain classifiers – e.g., tree-based classifiers or nearest-neighbor classifiers [4, 5]).
4. The framework also accommodates the discovery of new classes. When a new cluster is created, our selective sampling procedure will request for new labeled examples from the cluster. The newly acquired examples may suggest the presence of a new class.

The remainder of the paper is organized as follows. Section 2 describes the related work in data stream classification. Our proposed framework is introduced in Section 3. Section 4 presents our experimental results, while Section 5 concludes with a summary of the paper.

2. Related Work

Several incremental learning algorithms have been developed for data stream classification [9, 16]. Domingos and Hulten have proposed an online decision tree algorithm known as VFDT, which employs a statistical principle known as Hoeffding bound to provide guarantees on its asymptotic similarity to the corresponding batch tree [9]. This work was extended in [6] to handle concept drifts by growing alternative sub-trees to replace outdated sub-trees. Other extensions to the VFDT algorithm were proposed by Gama et al. [16] and Jin et al. [17]. In [15], Law et al. introduced an incremental classification algorithm based on multi-resolution data representation.

Many algorithms also employ data compression schemes to efficiently store the data for subsequent processing. Aggarwal et al. [4] used micro-clusters as a condensed representation of the data. Ding et al. [13, 14] have used Peano Count Trees to keep a compressed version of their spatial data for data stream classification. An alternative way for handling concept drift is to build an ensemble of classification models [3, 18]. While such an approach may be able to handle concept drifts, maintaining and applying a large number of classifiers can be quite costly.

Another model maintenance issue is choosing the appropriate labeled examples to create the training set. This is also known as the data expiration problem [3]. Most works in data stream classification employ a FIFO queue which replaces older examples with newer ones. Alternative approaches for handling this problem have been investigated by Fan [2] and Yang et al. [8].

The deferred update strategy has been employed by several authors. One approach is to keep track of the misclassification rate in each time window [8, 10]. If the rate is higher than a specified threshold, then the model is revised. Fan et al. [5] proposed an alternative way for detecting changes by analyzing the data distribution at the leaf nodes of a decision tree. This approach however is designed specifically for decision trees and does not perform selective sampling across space..

3. Prototype-Driven Classification

A data stream S is a sequence of examples $<X_1, X_2, \ldots>$, where each example X_i is a d-dimensional feature vector with a timestamp t_i. Each X_i is also associated with a class label $X_i.class$. An example is said to be labeled if its class value is known; otherwise, it is unlabeled. We partition the d-dimensional input space into k disjoint regions or clusters, $\{P_1, P_2, \ldots, P_k\}$. A prototype C_j is chosen as the representative point for each partition P_j. For input space with continuous-valued attributes, the prototype is represented by the cluster centroids[1].

3.1 Proposed Framework

Figure 1 shows a summary of our proposed framework. The framework consists of the following two phases:

1. **Initialization Phase** to generate an initial classification model and a set of clusters from the data.
2. **Monitoring and Updating Phase** to assign newly arrived examples to their corresponding clusters and to determine whether the current model is outdated.

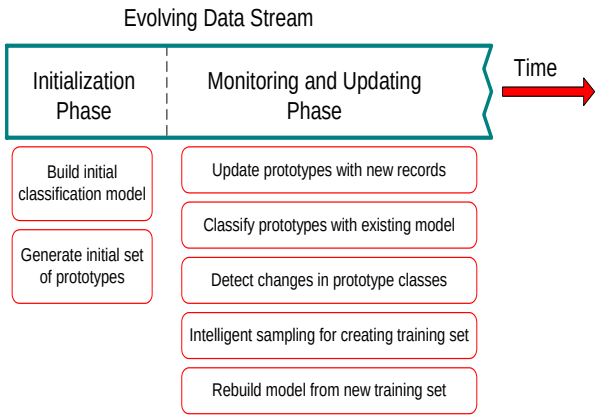


Figure 1: Framework of Proposed Approach

3.1.1 Initialization Phase

A classification model M is initially built from the labeled examples L. These examples are also used to partition the input space into k clusters. For each cluster P_j, we maintain a data structure (C_j, $class_j$, V_j, W_j), where C_j corresponds to the cluster prototype, $class_j$ denote its class label, V_j is a queue containing the labeled examples assigned to P_j, and W_j is a queue containing the unlabeled examples assigned to P_j. Due to memory limitations, the sizes of the queues are constrained as follows:

[1] This paper focuses only on centroid-based prototypes even though the framework can be generalized to data sets with nominal features.

$$\sum_{j=1}^{k} |V_j| + |W_j| \le \Delta \ , \qquad (1)$$

where Δ is the amount of memory available. In principle, our framework may accommodate any clustering algorithm including k-means, self-organizing map (SOM), and neural gas [7]. These algorithms seek to find a set of prototypes for encoding the data with minimum quantization error [11]. K-means uses a greedy approach to find the clusters. Despite its efficiency, it gets easily trapped in a local minimum and is susceptible to outliers. SOM arranges the centroids in a rectangular lattice and allows neighboring centroids to be updated for each example. This makes it less prone to outliers and local optima compared to k-means [19]. Since the lattice dimension is usually set to at most three, SOM may not be flexible enough for modeling complex manifolds [11]. Instead of using a predefined lattice, neural gas updates the centroids based on the rank of their distances to a given example [7]. The clustering method employed in this study is a variation of the recently developed online neural gas algorithm [11].

Table 1: Initialization Phase

Input: Labeled training set L, number of clusters k, and number of nearest neighbors m.
Output: Classifier M and set of clusters C

1. $M \leftarrow$ BuildClassifier(L); // Build initial model
2. $C = \{\}$;
3. **for** i =1 to k **do**
4. Initialize a new cluster P_i with centroid C_i.
5. C = C $\cup$ $\{C_i\}$;
6. **end**;
7. **do**
8. Randomly select an example X from L
9. Rank the prototypes in C based on their increasing distance to X
10. **for** k = 1 to m **do**
11. $C_j \leftarrow$ get the k^{th} highest ranked centroid
12. Update the prototype: $C_j = C_j + \eta_t e^{-(j-1)/\lambda_t}(X - C_j)$
13. **end**
14 **until convergence**
15 **for** each example $X \in L$
16 $P_j \leftarrow$ Find the closest cluster to X
17 Insert X into *the* queue V_j
18 **end**
19 Use M to classify all the clusters in C.
20 Discard all clusters with empty queues.

Table 1 summarizes the key steps of the initialization phase. First an initial classifier M is constructed from the labeled data. Our clustering algorithm begins by randomly choosing k initial centroids (Steps 3–6). The algorithm then proceeds to update the centroids incrementally in the following manner. Given a training example X, the algorithm ranks the prototypes according to their distance to X and chooses the m closest ones for updating (Steps 9–13) based on the following formula:

$$C_j = C_j + \eta e^{-(j-1)/\lambda}(X - C_j) \qquad (2)$$

where η is the learning rate, λ is a kernel width parameter, and j is the rank of the prototype (in terms of its distance to x). η controls the sensitivity of the centroid to newly arrived examples, while λ controls the influence of the neighbors. Each training example is then assigned to its closest cluster. The cluster prototypes are also classified using the model M (Step 19). Finally, we remove all clusters with empty queues.

3.1.2 Monitoring and Updating Phase

A summary of the monitoring and updating phase is shown in Table 2. The monitoring procedure is invoked to update the clusters each time a new example arrives, whereas the updating procedure is invoked at the end of each time window w to detect whether the current model is outdated. If significant changes are detected, the model will be revised.

Table 2: Monitoring and Updating Phase

Input: Data stream $S = L \cup U$, classifier M,
set of clusters C, threshold δ, window size W.
Output: Updated classifier M and set of clusters C

// **Monitoring Phase** (invoked each time a new example arrives)
1. **for** each example $X \in$ S **do**
2. find m nearest prototypes in C based on their to X
3. $C_j \leftarrow$ Get the nearest prototype
4. **if** (distance(X, C_j) > δ)
5. Create new cluster $P_{|C|+1}$ with centroid $C_{|C|+1} = X$;
6. **if** (X is *labeled*)
7. $P_{|C|+1}.class = X.class$;
8. Insert X into labeled queue $v_{|C|+1}$.
9. **else**
10. $P_{|C|+1}.class$ = classify(M, x)
11. Insert X into unlabeled queue $w_{|C|+1}$.
12. **end**
13. $C = C \cup \{P_{|C|+1}\}$;
14. **else**
15. Update centroid: $C_j = C_j + \eta e^{-0/\lambda}(X - C_j)$
16. **if** ($X \in U$)
17. Insert X into unlabeled queue $w^{(1)}$
18. **else if** ($X \in L$)
19. Insert X into labeled queue $v^{(1)}$
21. **end**
22. **for** k = 2 to m **do**
23. $C_j \leftarrow$ Get the k^{th} nearest centroid
24. **if** (distance(X, C_j) $\le \delta$)
25. Update centroid: $C_j = C_j + \eta e^{-(j-1)/\lambda}(X - C_j)$
26. **end**
27. **end**
28. **end**

// Updating Phase (invoked at the end of each time interval w)

29. **if** (t_i mod w == 0)
30. **if** (ChangeDetection(M, C))
31. *TrainSet* = IntelligentSampling({V},{W});
32. M = BuildClassifier(*TrainSet*);
33 Use M to classify all clusters in C.
34. **end**;
35. FlushQueue({v},{w});
36. **end**

When a new example X arrives, the algorithm selects its m closest centroids for updating. The update depends on the distance between X and the selected centroids. If the distance is less than δ, the location of the centroid is updated according to Equation 2. X is then inserted into the corresponding labeled (V)

or unlabeled (*W*) queue of its closest cluster. On the other hand, if the distance between *X* and the selected cluster centroid exceeds δ, a new cluster is formed with *X* as its prototype (Steps 4-13). *X* is then inserted into the corresponding labeled or unlabeled queue of the newly formed cluster. Furthermore, if *X* is a labeled example, the cluster is assigned the class label of *X*; otherwise the label of the new cluster is determined by applying the current model *M*.

During the updating phase, which is invoked periodically at the end of each time window w, the current model is applied to each cluster prototype. The ChangeDetection function, which will be described later in Section 3.2, is used to determine whether the model needs to be revised. If change is detected, the algorithm employs the selective sampling procedure described in Section 3.2.3 to create a training set from which the classifier can be re-trained. Finally, the labeled and unlabeled queues for each cluster are periodically flushed to eliminate outdated examples. A cluster is discarded if both of its queues are empty.

3.1.3 Selective Sampling Across Space

Our proposed algorithm uses a selective sampling across space strategy to choose the labeled examples needed for creating the training set. More specifically, our procedure focuses on: (1) selecting more labeled examples from regions that are hard to classify and (2) acquiring additional labels from the unlabeled examples in clusters whose predictions are most uncertain. By manipulating the distribution of training examples, the classifier is biased towards learning regions that are hard to classify.

Our sampling procedure is implemented with the aid of the labeled and unlabeled queues. The idea here is to incorporate more examples from impure clusters into the training set. Given a cluster P_i, its impurity is computed as follows:

$$\varepsilon_i = -\sum_j g_{ij} \log g_{ij} \qquad (3)$$

where g_{ij} be the fraction of labeled data from class j. Each cluster P_i is then assigned a weight factor f_i depending on the cluster impurity:

$$f_i = \frac{\varepsilon_i - 1/k * \log(1/k)}{Z} \qquad (4)$$

where k is the number of clusters and

$$Z = \sum_{i=1}^{k} (\varepsilon_i - 1/k * \log(1/k)) \qquad (5)$$

is the normalization factor. Note that, if all the clusters are pure, then the sampling is performed uniformly across all clusters.

Let *N* be the training set size. Based on Equation (4), the the training set is created using Nf_j examples from cluster P_j. If the number of examples in the labeled queue for P_j is less than Nf_j, we consider two approaches for acquiring additional labeled examples. The first approach performs sampling with replacement on the examples stored in the labeled queue. This approach is known as *Deferred-Bootstrap*. The second approach explicitly requests for new labels from the examples stored in the unlabeled queue. This allows the possibility of discovering new classes especially when a new cluster does not have any labeled examples. We call this the *Deferred-Active* approach.

3.2 Cluster Based Change Detection

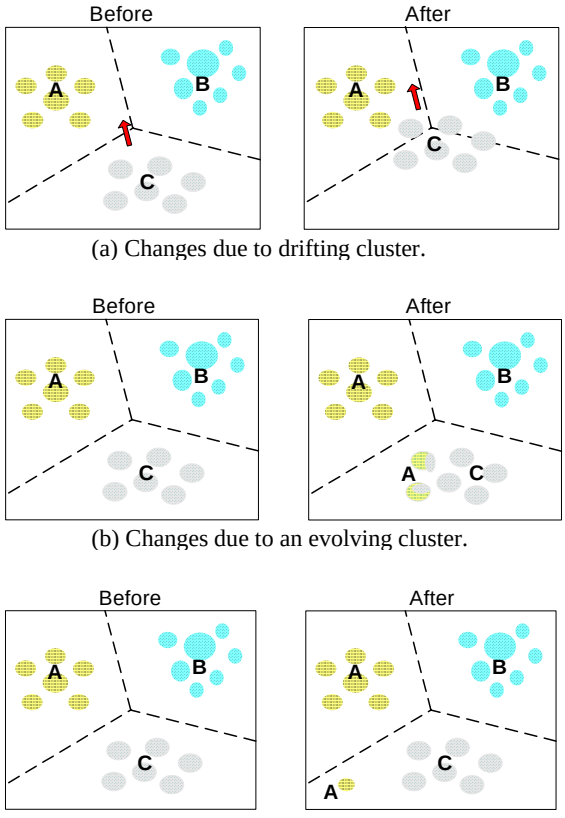
This section explains how the change detection algorithm works. Note that each cluster is associated with a class label. The class labels are determined either during initialization (Step 20 in Table 1) or during the monitoring and updating phase in previous time windows (Steps 7, 10, and 30 in Table 2).

Our algorithm applies two tests to determine whether the current model is outdated. First, it compares the class label of each prototype to the label predicted by the current model. If there is a difference, the model will be rebuilt. Second, it performs a chi-square test to determine whether the class distribution within each cluster has changed significantly. To compute this, let OC_{ij} be the number of examples from class j in cluster i during the previous time window and NC_{ij} be the number of examples in the current time window. The chi-square statistic for cluster i is:

$$X_i^2 = \sum_j \frac{(OC_{ij} - \frac{NC_{ij}}{\sum_j NC_{ij}} \times \sum_j OC_{ij})^2}{OC_{ij}} \qquad (6)$$

If the chi-square for any cluster is higher than a pre-defined threshold, the model will be rebuilt.

To understand the intuition behind our change detection approach, Figure 2 illustrates three scenarios that will trigger



(a) Changes due to drifting cluster.

(b) Changes due to an evolving cluster.

(c) Changes due to emerging cluster.

Figure 2: Three scenarios that will trigger model rebuilding in the change detection procedure.

model rebuilding. In each figure, we show the data distributions in two successive time windows (before and after) as well as the decision boundary produced by the current model. We assume that the examples belong to three classes: A, B, or C.

In Figure 2(a), the clusters associated with class C have drifted upwards. When the current model is applied, it predicts some of the clusters from class C as either A or B (depending on their locations with respect to the decision boundary). Because of the discrepancy between the predicted class and the assigned class of the clusters, the model rebuilding procedure is invoked. In Figure 2(b), the class distributions for two of the clusters have changed (from pure class C to a mixture of classes A and C). Such distribution change can be detected using the chi-square test. In Figure 2(c), a new cluster (for class A) is formed because its location is far away from other existing clusters (Steps 4-13 in Table 2). When the model *M* is applied, the difference between the predicted class and the assigned class of the cluster will trigger the model rebuilding procedure.

4. Experimental Evaluation

We have implemented our algorithm in MATLAB and conducted our experiments on a PC with Intel Pentium 4, CPU 3.2 GHz with 1.00 GB of RAM.

4.1 Data set

There are two data sets used in our experiments. The first corresponds to a synthetic data set generated using a mixture of three Gaussian distributions with the following parameters:

$$\mu_1 = (1.5,0), \quad \Sigma_1 = \begin{bmatrix} 0.375 & 0 \\ 0 & 0.375 \end{bmatrix}$$

$$\mu_2 = \left(3, \left(\frac{t-50}{25}\right)\right), \quad \Sigma_2 = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$$

$$\mu_3 = (4.5,0), \quad \Sigma_3 = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.375 \end{bmatrix}$$

The synthetic data stream is generated for 100 time windows. In each time window, there are 2500 examples generated for each mixture component. The entire data stream contains 2500 $\times$ 3 $\times$ 100 = 750,000 examples.

Figure 3 shows snapshots of the data taken at different time windows. Notice that as time progresses the cluster due to the second component drifts upward while the rest of the clusters remain stationary. Also notice that the data is well separable at the beginning and at the end of the data stream. In the middle of data stream there is substantial overlap among the three classes. We therefore expect the accuracy of the model to be very good at the two ends of the data stream and poor in the middle of it.

The second dataset used for our experiment corresponds to the KDDCup 1999 Intrusion Detection data. We randomly choose a million records from the full KDDCup data to perform our experiment.

4.2 Experimental Results

We compare the performance of the following data stream classification methods in our experiments:

1. **Periodic Update (PERIODIC).** This approach rebuilds the model at the end of each time window.
2. **CVFDT,** which is a state of the art decision tree classifier for evolving data streams developed by Hulten et al. [6].
3. **Deferred Update (DEFERRED).** In this approach, the model is tested at the end of each time window (using the ChangeDetection function in Table 2) to determine whether it is outdated. If so, the algorithm uses labeled examples from the most recent time window to rebuild the model.
4. **Deferred Update with Selective Sampling Across Space (DEFERRED-Bootstrap and DEFERRED-Active).** These approaches correspond to the deferred update strategies described in Section 3.2.3.

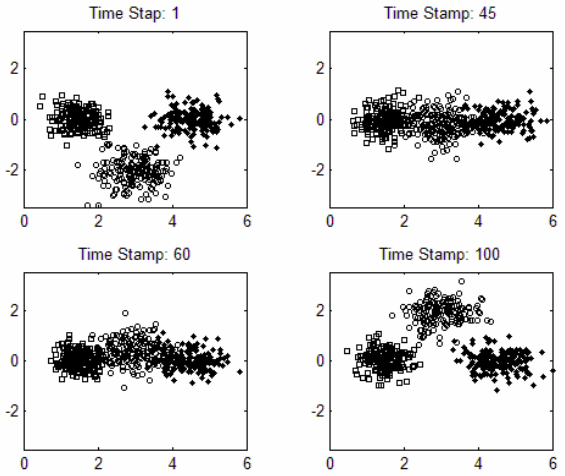


Figure 3: The synthetic dataset

4.2.1 Performance Comparison for Synthetic Data

For the synthetic data set, there are 7500 examples generated in each time window, out of which only 20% of them are labeled. The training set during the initialization phase contains 7500 labeled examples. Because of memory limitations, we assume that we may store only up to 4000 examples. The maximum number of clusters is set to 50 and the learning rate is .02. We use decision tree as the base classifier of our framework.

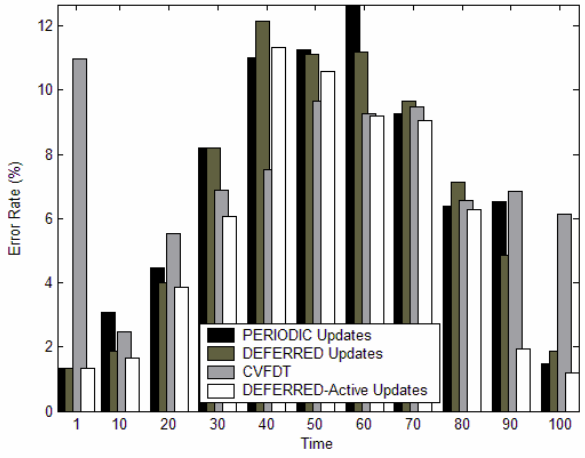


Figure 4: Error rate comparison for synthetic dataset

Figure 4 shows the error rate comparison among the four competing approaches while Figure 5 shows their update times. The horizontal axis in both figures refers to the time window. Figure 4 suggests that the DEFERRED-Active approach tends to produce the lowest error rate compared to other approaches. In Figure 5, the points in each row indicate the update times of the models. Notice that DEFERRED-Active requires less number of updates (24) compared to the PERIODIC (100) and DEFERRED (29) approaches. These results demonstrate that the deferred update approach with selective sampling strategy generally produces better models than those produced by algorithms that periodically rebuild their models. Furthermore, because of their less number of updates, the amount of labeled examples needed for each time window can be reduced.

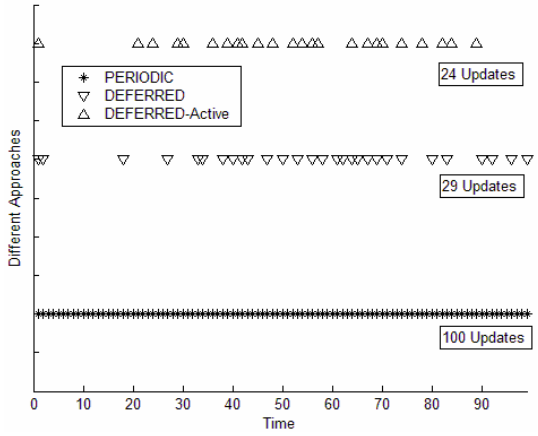


Figure 5: Updating time for synthetic dataset

Figure 6 shows the number of labeled examples chosen from each cluster to create the training set. Each rectangle denotes the location of a cluster prototype. The size of the rectangle represents the number of examples chosen—the larger it is, the more examples chosen from the cluster to create the training set. Observe that our selective sampling procedure tends to choose more examples from regions near the decision boundary. This explains the improved accuracy obtained using the DEFERRED-Active method. Finally, note that the runtime using DEFERRED-Active is about 150 seconds, which means a record processing rate of 5000 examples per second.

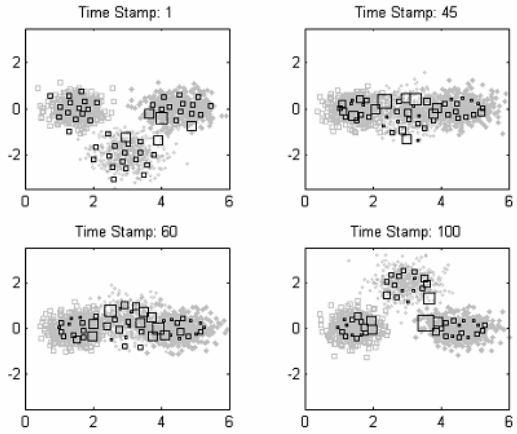


Figure 6: Selective sampling from the synthetic data

4.2.2 Performance Comparison for KDDCup Data

For the KDDCup data, we assume the data stream is partitioned into 100 time windows, with 10,000 examples collected in each window. We used the examples in the first time window for initialization. For the remaining time windows, we assume that only 10% of the examples are labeled. For model rebuilding, we assume the training set contains 3000 labeled examples. We use a decision tree classifier to build our classification models.

Figure 7 shows the error rate comparison for the three approaches. Both DEFERRED-Active and DEFERRED-Bootstrap approaches produce models with lower error rates than the other approaches. Furthermore, the DEFERRED-Active tends to produce lower error rates compared to DEFERRED-Bootstrap. This is because DEFERRED-BOOTSTRAP reuses some of the labeled examples whereas DEFERRED-Active acquires additional labeled examples by requesting the labels for some of the unlabeled examples.

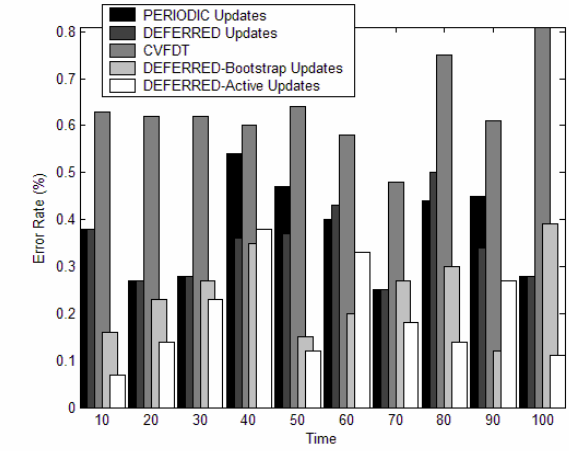


Figure 7: Error rate comparison for intrusion detection dataset

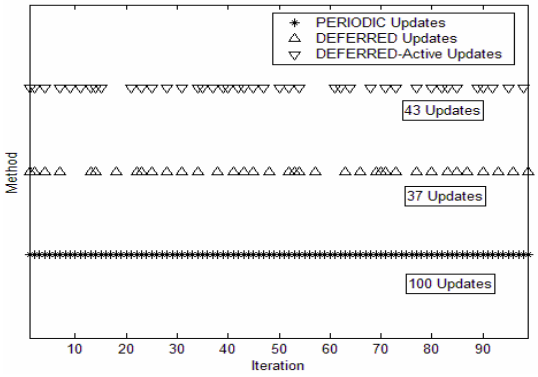


Figure 8: Update time for intrusion detection dataset

Figure 8 shows the model update times for each method. Notice that the deferred update methods perform less number of updates than the PERIODIC approach. The average processing rate for DEFERRED-ACTIVE is 1200 examples per seconds.

4.2.3 Runtime Analysis

Another important criterion for evaluating any data stream mining algorithm is its time complexity. A good algorithm needs to be able to process the examples at the rate of the

incoming data stream. Our proposed algorithm performs a one time batch clustering step during the initialization phase. Then for each newly arrived example, it needs to update the centroid of its *m* nearest clusters. This requires O(mk) computations, where k is the number of clusters. The time complexity for the monitoring phase is linear in the number of clusters and number of neighbors. Figures 9, 10, 11 show the number of examples processed per second when the number of clusters, number of neighbors, and number of attributes increase. As the value for each parameter increases, the number of processed examples decreases but still quite reasonable for data stream processing.

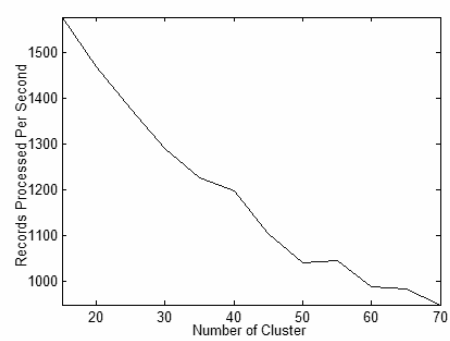


Figure 8: Processing rate for different number of clusters

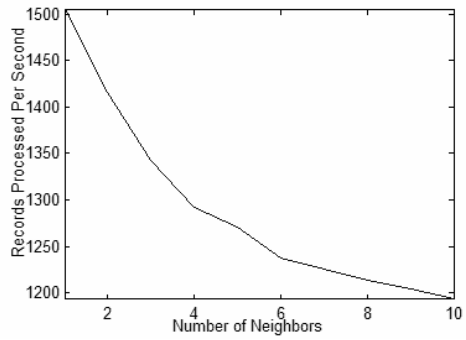


Figure 9: Processing rate for different number of neighbors

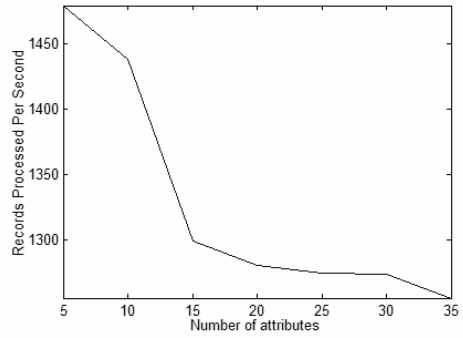


Figure 10: Processing rate for different dimensionality

4.2.4 Sensitivity Analysis

There are several parameters that need to be tuned in our algorithm. These parameters correspond to the learning rate (Equation 2), the number of neighbors, the number of clusters and window size. In this section, we illustrate the effect of using different parameter values on the performance of our algorithm. Some of these parameters may affect the runtime of the algorithm while others may affect model accuracy. The default values of these parameters are: window size = 10000, learning rate = 0.1, number of neighbors = 5, and number of clusters = 30. We empirically show that our algorithm is not too sensitive to the choice of parameter values. We use the KDDCup data for our experiments.

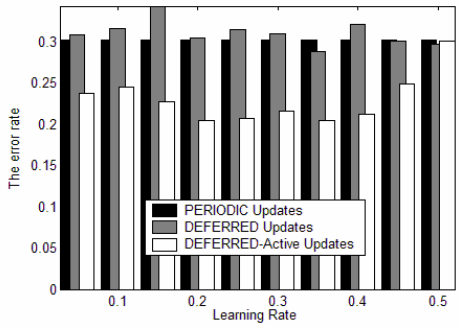


Figure 12: Error rate at different learning rates

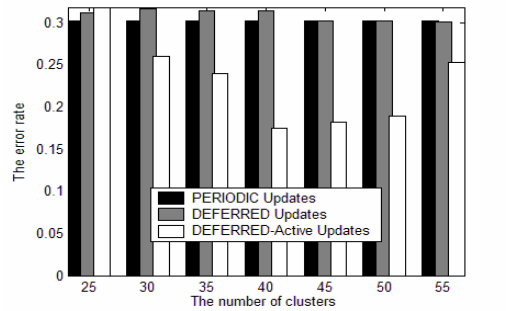


Figure 13: Error rate for different number of clusters

The learning rate parameter controls the rate at which the algorithm forgets older examples when computing the cluster centroids. A large learning rate causes the algorithm to be more sensitive towards newer samples. In principle, the learning rate should be chosen based on the rate of concept drift. For rapidly evolving data streams, a large learning rate would be preferable while for slower drifts, a small learning rate would be sufficient. Figure 12 shows the effect of varying the learning rates on the performance of our algorithm. Notice that although the default learning rate does not yield the optimal result, it is still better than other algorithms. More importantly, the performance of the model is quite stable when the learning rate is varied.

The number of clusters is another parameter that affects the performance of the algorithm. When the number of clusters is too small, the cluster sizes will be very large and as a result, the change detection function may not be that effective. On the other hand, when the number of clusters is large, there are many clusters with few samples, which may reduce the effectiveness of our selective sampling procedure. Figure 13 shows the result of our algorithm when the number of clusters is varied. While the performance of our algorithm varies with the number of clusters, it still outperforms other algorithms in most cases.

Figure 14 shows the error rate when the number of neighbors is varied. Observe that the number of neighbors does

not have a significant impact on the error rate of the model. Nevertheless, the number of neighbors does affect the average processing time of the algorithm. As the number of neighbors increases, the number of records processed per second will decrease (Figure 10). Our experience shows that a choice of 1-5 neighbors seems reasonable for this dataset.

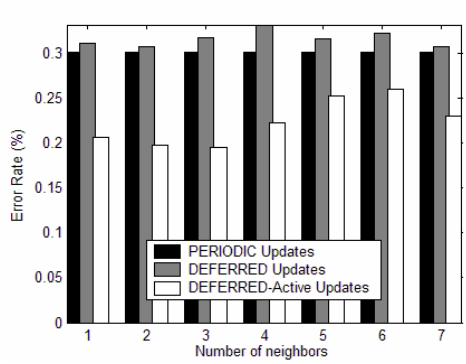


Figure 14: Error rate with different number of neighbors

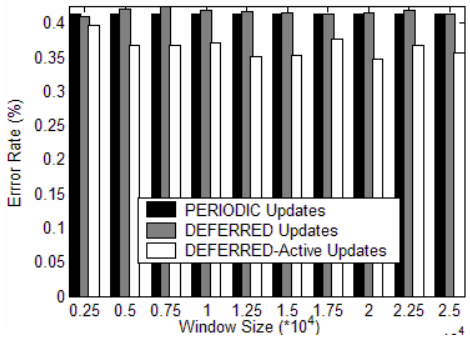


Figure 15: Error rate comparison when the window size changes while the available memory is kept constant.

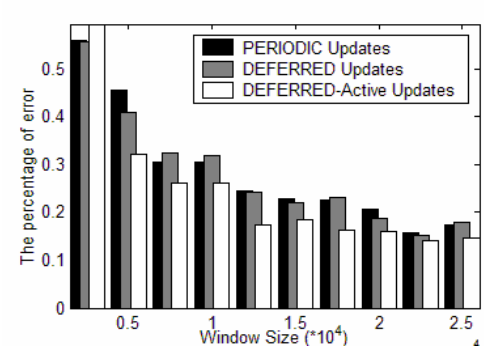


Figure 16: Error rate comparison when the available memory changes proportional to window size.

Window size is another parameter of this algorithm as well as other window-based data stream algorithms. As the window size increases, the error rate tends to increase if the amount of memory is constant. This is because any new model built at the end of the current time window must wait until the completion of the next time window before it is applied. We perform two experiments to evaluate the effect of window size. First, we keep the available memory constant and increase the size of the time window. Figure 15 shows the result of this experiment. Observe that the number of updates decreases as window size increases. In the second experiment, we increase the window size and the available memory. Figure 16 shows the result of the experiment. As can be seen, when we use a larger window size and a larger training set size, the accuracy increases.

5. Conclusions

In this paper, we propose a novel prototype-based framework for data stream classification. Our approach has several advantages. First, it reduces the number of model rebuilding steps without sacrificing accuracy. Second, the framework allows us to decouple change detection from model updating. Thus it can be used as a wrapper approach for applying any classification method. Third, the prototypes allow us to identify regions in the input space that are hard to classify. This allows us to develop a selective sampling procedure to select training examples that can improve the performance of the classifier. Our experimental results confirmed these assertions.

References:

[1] G. Widmer, M. Kubat, Learning in the presence of concept drift and hidden context, Machine Learning, 23(1), 69-101 (1996).

[2] Wei Fan, Systematic Data Selection to Mine Concept-Drifting Data Streams. In Proc of KDD (2004).

[3] H. Wang, W. Fan, P.S. Yu and J. Han, Mining Concept-Drifting Data Streams Using Ensemble Classifiers, Proc of KDD (2003).

[4] C. Aggarwal, J. Han, P.S. Yu, On Demand Classification of Data Streams, In Proc of KDD (2004).

[5] W. Fan, Y. Huang, H. Wang, P.S. Yu, Active Mining of Data Streams, In Proc of SIAM Int'l Conf on Data Mining (2004).

[6] G. Hulton, L. Spencer and P. Domingos, Mining Time-changing Data Streams, In Proc of KDD (2001).

[7] T.M. Martintetz, S.G. Berkarich, K.J. Schultem. Neural Gas Network for Vector Quantization and its Application to Time Series Prediction. IEEE Trans on N. Networks, 4:558-569 (1993).

[8] Y. Yang, X. Wu and X. Zhu, Combining Proactive and Reactive Predictions for Data Streams, In Proc of KDD (2005).

[9] P. Domingos and G. Hulten, Mining High-Speed Data Streams, In Proc of KDD (2000).

[10] F.Chu and C.Zaniolo, Fast and Light Boosting for Adaptive Mining of Data Streams. In Proc of PAKDD (2004).

[11] D. Deng, N. Kasabov, Online pattern analysis by evolving self organizing maps, J. of Neurocomputing, 51, pp. 87-103 (2003).

[12] L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, R. Motwani. Streaming-data algorithms for high-quality clustering. In Proc of ICDE (2002)

[13] Q. Ding, Q. Ding, W. Perrizo, Decision Tree Classification of Spatial Data Streams Using Peano Count Trees. In SAC (2002)

[14] M. Khan, Q. Ding, W. Perrizo, K-nearest Neighbor Classification on Spatial Data Stream Using P-trees, In Proc of PAKDD (2002)

[15] Y-N Law, C. Zaniolo, An Adaptive Nearest Neighbor Classification Algorithm for Data Streams, Proc of PKDD (2005)

[16] J. Gama, R. Rocha and P. Medas, Accurate Decision Trees for Mining High-Speed Data Streams, In Proc of KDD (2003).

[17] R. Jin and G. Agrawal, Efficient Decision Tree Construction on Streaming Data. In Proc of KDD (2003).

[18] W.N. Street, Y. Kim, A streaming Ensemble Algorithm (SEA) for Large-Scale Classification. In Proc of KDD (2001)

[19] F. Bacao, V. Lobo, M. Painho, Self-organizing map as Substitutes for K-means Clustering, In Proc of ICSS.