

Loadstar: A Load Shedding Scheme for Classifying Data Streams

Yun Chi*, Philip S. Yu†, Haixun Wang†, Richard R. Muntz*

*Department of Computer Science, University of California, Los Angeles, CA 90095

†IBM Thomas J. Watson Research Center, Hawthorne, NY 10532

ychi@cs.ucla.edu, {psyu,haixun}@us.ibm.com, muntz@cs.ucla.edu

Abstract

We consider the problem of resource allocation in mining multiple data streams. Due to the large volume and the high speed of streaming data, mining algorithms must cope with the effects of system overload. How to realize maximum mining benefits under resource constraints becomes a challenging task. In this paper, we propose a load shedding scheme for classifying multiple data streams. We focus on the following problems: i) how to classify data that are dropped by the load shedding scheme? and ii) how to decide when to drop data from a stream? We introduce a quality of decision (QoD) metric to measure the level of uncertainty in classification when exact feature values of the data are not available because of load shedding. A Markov model is used to predict the distribution of feature values and we make classification decisions using the predicted values and the QoD metric. Thus, resources are allocated among multiple data streams to maximize the quality of classification decisions. Furthermore, our load shedding scheme is able to learn and adapt to changing data characteristics in the data streams. Experiments on both synthetic data and real-life data show that our load shedding scheme is effective in improving the overall accuracy of classification under resource constraints.

keywords: data mining, data streams, load shedding, classification, quality of decision, feature prediction, Markov model.

1 Introduction

Many new applications process multiple data streams simultaneously. For instance, in a sensor network, data flows from a large number of embedded sensors; and in the stock market, each security generates a stream of quotes and trades. However, applications that handle these unbounded, high speed incoming data are constrained by limited resources (e.g., CPU cycles, bandwidth, and memory).

Resource allocation for distributed stream applications is often formulated as an optimization problem [12, 8]. Much work focuses on allocating resources in a best-effort way so that performance degrades gracefully. For instance, if the data characteristics from a sensor exhibit a predictable trend, then the precision constraints might be satisfied by transmitting only a fraction of the sensor data to the remote server.

Other approaches assume that a set of Quality-of-Service (QoS) specifications are available [1, 3, 14]. A load shedding scheme decides when and where to discard data and how much data to discard according to the QoS specification. In other words, these approaches assume that the impact of load shedding on performance is known *a priori* through the QoS specifications.

Goal In this paper, we argue that for many data mining tasks a more intelligent load shedding scheme for streaming data is required. The goal of mining is to maximize certain benefits (e.g., to detect as many credit card transaction frauds as possible). It is more complex than achieving high precision of simple computations such as aggregation (e.g., AVG, SUM, and COUNT). Because in those cases, high precision can usually be secured as long as the percentage of data sampled from the stream reaches a certain threshold.

The benefits of mining does not depend on the sampling rate. For instance, assume the percentage of fraudulent credit card transactions is p and our fraud detector has 100% accuracy. Then, under random sampling, the expected number of frauds we can detect per investigated transaction is fixed at p . To increase the efficiency of fraud detection, we need to know how benefits of mining will be affected by the next incoming data before resources are committed to investigate it.

Thus, our goal is to design a load shedding scheme to maximize the benefits per resource used in mining.

Challenges Load shedding in *mining* data streams is a new topic and it raises many challenges. Although load shedding has been studied for *managing* data streams, many assumptions in these studies are not appropriate

*The work of these two authors was partly supported by NSF under Grant Nos. 0086116, 0085773, and 9817773.

for data mining tasks.

First, for many simple queries (e.g., aggregation) considered for managing data streams, it is often safe to assume that the quality of the query result depends only on the sample size. Some approaches assume simple (e.g., monotonic, or even concave or piecewise linear) QoS curves, which depict the relationship between the quality and the sample size, are available to the load shedding mechanism. In contrast, in mining data streams, sample size itself cannot guarantee good mining result, because the quality of mining often depends on specific feature values in a non-monotonic way. For example, in certain regions of the feature space, a classifier may have very high confidence in its classification decision, even if the feature value is only known approximately. But in other regions, the classifier may not be very sure about its classification decision because in these regions, a small variation in a feature value may change the decision. In this case, resources (i.e., computing the exact feature values) should be allocated to a stream if the decision is more sensitive to the feature value of the data in this stream. Thus, the challenge lies in determining how to make the resource allocation in a best effort way to minimize classification errors.

Second, data mining applications are often more sensitive to changes in data characteristics. For instance, a small move in the feature space may totally change the classification results, and more often than not, it is such changes that we care about the most. Thus, feature value prediction is important to load shedding design for mining data streams. Fortunately, many feature values (e.g, the reading of sensors that measure the temperature or the water level of a river, or the feature values extracted from consecutive satellite images) have strong time-correlation and we can build models to take advantage of such correlation. Thus, the challenge lies in building a feature predictor that is able to capture the time-correlation and adapt to the time-variance of the feature values.

Our Contributions To the best knowledge of the authors, this is the first work on load shedding for mining data streams. We make the following contributions. (1) We define two *quality of decision* (QoD) measures for classification based on the predicted distribution of the feature values in the next time unit. (2) We develop a prediction model for feature values using Markov models whose parameters can be updated in real time to reflect parameter changes. (3) We combine the first two to obtain a load shedding scheme, Loadstar¹, for classifying multiple data streams. Experiments on both synthetic

data and real-life data show that our load shedding scheme is effective in improving the accuracy of data stream classification in the presence of system overload.

Paper Organization The rest of the paper is organized as follows. We formally define the problem in Section 2. In Section 3, we introduce two QoD (quality of decision) measures. In Section 4, we present our Markov-chain model for predicting feature values. In Section 5, we describe Loadstar, the overall load shedding scheme. In Section 6, we summarize experiment results which demonstrate Loadstar’s effectiveness. In Section 7, we review related work and we conclude in Section 8 with future directions.

2 Problem Definition

The major system components are illustrated in Figure 1. Raw data flows in via multiple streams and are fed to the data preparation and analysis block through a communication channel. The data preparation and analysis block is responsible for data cleaning, feature extraction and composition, etc. The derived features enter the data classification block, which outputs mining results.

In this paper, we assume that data preparation and analysis is CPU intensive. In comparison, the CPU consumption for classification is negligible. This is true in many real applications especially those that handle multimedia data, for which feature extraction is usually a CPU intensive procedure. For example, if the raw data are text documents, the data preparation and analysis may involve removing stop words, counting the frequency of important words, projecting the vector of word frequencies to some pre-defined conceptual space, filtering the projected values in each dimension using thresholds, etc [2]; if the raw data are images from satellites, computing the features, such as luminance, shape descriptor, amplitude histogram, color histogram and spatial frequency spectra, will usually take a lot of CPU time [13]. As a result, when the system is overloaded, the data preparation and analysis block cannot process all of the data and load shedding is needed. (Another equivalent scenario is when the bandwidth of the communication channel is limited and therefore not all raw data can be transmitted to the data preparation and analysis block.)

The input to the system consists of multiple streams of raw data. When the system is overloaded, data from some of the streams are dropped. For those streams whose data is dropped, their feature values can be predicted by the feature predictor block, based on historic feature values. Therefore, the classifier will handle both the real feature values generated by

¹A Load Shedding Scheme for Streaming Data Classifiers

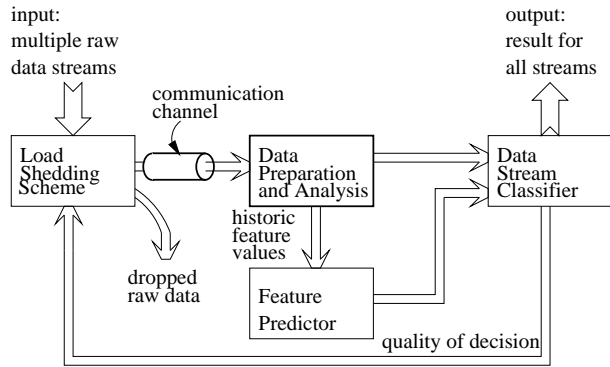


Figure 1: The System Setup

the data preparation and analysis block, and predicted feature values for those streams whose data has been dropped.

We assume that the classifier handles data streams consisting of a d -dimensional feature vector $\mathbf{x} \in \mathbf{X}^d$ (x_i can be either continuous or categorical) and produces a class label $c_i \in \{c_1, \dots, c_K\}$. The classifier performs classification for each incoming \mathbf{x} no matter whether \mathbf{x} is real or predicted feature values. The objective is to design a load shedding scheme that minimizes the overall error rate of the data mining task when the system is overloaded.

In this paper, we restrict the data mining task to be the classification problem, although the technique can be extended to other data mining tasks, such as clustering, on data streams.

3 Quality of Decision

Load shedding takes place when data from multiple streams exceeds the processing capacity. We are interested in load shedding schemes that ensure shed load has minimal impact on the benefits of mining.

In order to do this, we need a measure of benefit loss if data \mathbf{x} from a certain stream is discarded. However, we must be able to do that without seeing \mathbf{x} 's real feature values. In this section, we propose two QoD metrics, and in Section 4, we present a method to predict feature values such that we can make load-shedding decisions before seeing the real data.

3.1 The Quality of Classification One way to view a classifier is to consider it as a set of *discriminant functions* $f_i(\mathbf{x}), i = 1 \dots K$. The classifier assigns class label c_k to \mathbf{x} if $f_k(\mathbf{x}) \geq f_i(\mathbf{x}), \forall i$ ([6]). For traditional classification, only the ranks of the discriminant functions are important in decision making, i.e., we only care if we are right or wrong, and do not care how far off we are.

Consider an example where there are two classes

and the data is one dimensional (i.e., there is a single feature x). Figure 2(a) shows the two discriminant functions and Figure 2(b) their log ratio. We use logarithmic values because first, logarithm is a monotonically increasing function which preserves the original ranks of the discriminant functions; second, the ratio is invariant with the respect to the scale; third, as we will see shortly, it makes computations simpler.

For a given feature value x , if $f_2(x)$ is greater than $f_1(x)$, we assign class label c_2 to x ; we do not care how much $f_2(x)$ is greater than $f_1(x)$. For example, in two data streams, incoming elements with feature values $x = 2$ and $x = 1.5$ are both classified as c_2 . However, when the feature values are not exact, the two classification decisions will have different levels of certainty. For example, assume that $x = 2$ and $x = 1.5$ are *current* feature values and we believe x will not change dramatically in the *next* step. In such a case, if the classifier has to make a classification decision for the next step without updated feature values, it may still assign class label c_2 to both data streams; however, in this case, for the data stream with $x = 2$, the classifier is much more *certain* about its decision than for the data stream with $x = 1.5$. Intuitively, the *quality* of the classification decision for the first data stream is higher than that of the second data stream. If we have to shed load in the next step, we should shed load from the data stream whose current feature value is $x = 2$, because by allocating the available resource to the data stream with less quality of decision (i.e., the data stream with current feature value $x = 1.5$), we expect to gain more benefits in term of the improvement in the classification accuracy.

The question is, how to quantify this quality of decision?

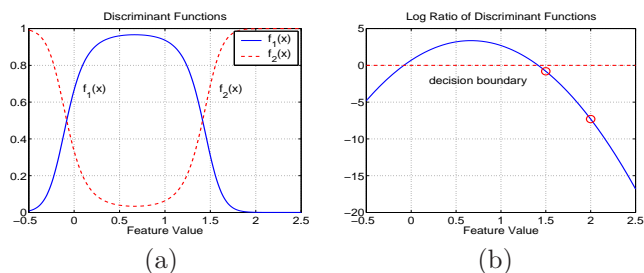


Figure 2: Certainty of a Classifying Decision

3.2 Quantifying the Quality of Decision Assume we have derived a probability density function for \mathbf{X} , the feature value in the next time unit:

$$(3.1) \quad \mathbf{X} \sim p(\mathbf{x})$$

It is worth mentioning that $p(\mathbf{x})$ is different from the estimated prior distribution $p(\mathbf{x}|\mathcal{D})$ that can be obtained from the training data \mathcal{D} . When we build the classifier based on \mathcal{D} , we consider each observation in \mathcal{D} as an independent sample from an unknown distribution. Here by $p(\mathbf{x})$, we mean that through some mechanism (e.g., by taking advantage of the temporal locality of the data), we have obtained an estimation of the feature value of the next time unit, and it is in the form of a density $p(\mathbf{x})$.

Quality Defined on Log Ratio We assume that the discriminant functions have positive values. Using Eq (3.1), the distribution of feature values in the next time unit, we can compute the expected logarithmic value of the discriminant functions:

$$(3.2) \quad E_{\mathbf{X}}(\log f_i(\mathbf{x})) = \int_{\mathbf{x}} (\log f_i(\mathbf{x}))p(\mathbf{x})d\mathbf{x}$$

We use δ_1 to represent the decision which chooses the class label that maximizes the expected value:

$$(3.3) \quad \delta_1 : k = \arg \max_i E_{\mathbf{X}}(\log f_i(\mathbf{x}))$$

Eq (3.3) only gives the classifying decision; to perform load shedding, we need to give a quantitative measure about the certainty of the decision. We introduce our first *quality of decision (QoD)* measure:

$$(3.4) \quad Q_1 = E_{\mathbf{X}} \log \left(\frac{f_k(\mathbf{x})}{f_{\tilde{k}}(\mathbf{x})} \right) \\ = E_{\mathbf{X}}(\log f_k(\mathbf{x})) - E_{\mathbf{X}}(\log f_{\tilde{k}}(\mathbf{x}))$$

where \tilde{k} is the second best decision according to Eq (3.2).

From the definition, we have $Q_1 \geq 0$. Intuitively, the higher the Q_1 , the more we are confident in our best-effort decision, that is, we compare our decision with the second-best choice, and if the expected performance of our decision is much better than that of the second-best choice, we believe that our decision has high quality.

Quality Defined on Overall Risk We introduce another quality of decision measure based on Bayesian decision theory. We use the *posterior* distribution of the classes given the feature vectors as the discriminant functions.

At point \mathbf{x} in feature space \mathbf{X} , if we decide the class is c_i , then the conditional risk of our decision is

$$R(c_i|\mathbf{x}) = \sum_{j=1}^K \sigma(c_i|c_j)P(c_j|\mathbf{x})$$

where $\sigma(c_i|c_j)$ is the loss function, i.e., the penalty incurred when the real class is c_j and our decision is c_i . For example, for zero-one loss, we have:

$$\sigma(c_i|c_j) = \begin{cases} 0 & i = j \\ 1 & i \neq j \end{cases}$$

in which case we can simplify the conditional risk as

$$R(c_i|\mathbf{x}) = 1 - P(c_i|\mathbf{x})$$

Because we have the distribution of the feature value \mathbf{x} at the next time unit, we can compute the expected risk for a decision for next time unit as

$$E_{\mathbf{X}} [R(c_i|\mathbf{x})] = \int_{\mathbf{x}} R(c_i|\mathbf{x})p(\mathbf{x})d\mathbf{x}$$

We use δ_2 to represent the best-effort decision rule which minimizes this expected risk:

$$(3.5) \quad \delta_2 : k = \arg \min_i E_{\mathbf{X}} [R(c_i|\mathbf{x})]$$

Assume for $\mathbf{x} \in \mathbf{X}$, the optimal decision is c^* . (More rigorously, c^* should be written as $c^*(\mathbf{x})$.) Because the real value of \mathbf{x} is unknown, c^* is infeasible to realize in our load shedding environment. The risk associated with c^* is

$$E_{\mathbf{X}} [R(c^*|\mathbf{x})] = \int_{\mathbf{x}} R(c^*|\mathbf{x})p(\mathbf{x})d\mathbf{x}$$

This risk is the Bayesian lower bound based on distribution $p(\mathbf{x})$. We then define the QoD based on the difference between the expected risk and the lower bound:

$$(3.6) \quad Q_2 = 1 - (E_{\mathbf{X}} [R(c_k|\mathbf{x})] - E_{\mathbf{X}} [R(c^*|\mathbf{x})]) \\ = 1 - \int_{\mathbf{x}} [P(c^*|\mathbf{x}) - P(c_k|\mathbf{x})] p(\mathbf{x})d\mathbf{x}$$

From the definition, we have $0 \leq Q_2 \leq 1$. Also, $Q_2 = 1$ if and only if c_k is the optimal decision for every $\mathbf{x} \in \mathbf{X}$ where $p(\mathbf{x}) > 0$. Intuitively, the larger the Q_2 , the higher quality the decision.

A Comparison of the Two QoDs The quality of classification depends on two factors. The first factor is the feature distribution $p(\mathbf{x})$. Both Q_1 and Q_2 have taken $p(\mathbf{x})$ into consideration. For example, as shown in Figure 2, we are quite confident that if $x = 1$ then the class is c_1 , and if $x = 2$ then the class is c_2 . If $p(\mathbf{x})$ is given by $P(x = 1) = P(x = 2) = 0.5$, then both Q_1 and Q_2 will give low values. Thus, resources allocated to the stream (which helps to reveal the real feature values) will improve the quality of decision.

The second factor is the discriminant functions. In this case, although both Q_1 and Q_2 reflect the quality of decision, Q_2 is a better metric, because it indicates the benefit of allocating resources to the data stream. For example, consider an extreme case where $f_1(x) = f_2(x) = 0.5$ for all x . Then, Q_1 is 0, which (correctly) indicates that the classification result is very unreliable. Q_2 is 1, which (also correctly) indicates that allocating more resources to the data stream will not improve the accuracy of the classification.

Base on the above discussion, we expect Q_2 to perform better than Q_1 . This is verified by the experiments that will be given in a later section.

Naive Bayesian Classifier The QoDs defined above are mathematically appealing but computationally challenging, especially when the dimension of the feature space d is large. We can, however, simplify the QoD defined on log ratio (δ_1 and Q_1) by assuming that each feature is conditionally independent given the class labels. With this assumption, a very simple classifier, the naive Bayesian classifier, can be applied. In spite of its naivety, it has been shown in many studies that the performance of naive Bayesian classifiers are competitive with other sophisticated classifiers (such as decision trees, nearest-neighbor methods, etc.) for a large range of data sets [5, 9]. Because of the ‘‘Bayesian’’ assumption, we restrict the discriminant function to be the posterior distribution of each class. (Without the ‘‘Bayesian’’ restriction, for δ_1 and Q_1 , the discriminant functions could be any positive functions on the feature space.)

With the assumption of a naive Bayesian classifier, we have

$$\begin{aligned} E_{\mathbf{X}}(\log f_i(\mathbf{x})) &= E_{\mathbf{X}}(\log P(c_i|\mathbf{x})) \\ &= E_{\mathbf{X}}\left(\log \frac{P(\mathbf{x}|c_i)P(c_i)}{\sum_j P(\mathbf{x}|c_j)P(c_j)}\right) \end{aligned}$$

The classifying decision δ_1 and the QoD Q_1 only depend on the relative value. So we ignore the denominator and derive the following.

$$\begin{aligned} E_{\mathbf{X}}[\log(P(\mathbf{x}|c_i)P(c_i))] &= E_{\mathbf{X}} \log P(\mathbf{x}|c_i) + E_{\mathbf{X}} \log P(c_i) \\ &= E_{\mathbf{X}} \sum_j \log P(x_j|c_i) + \log P(c_i) \\ &= \sum_j E_{\mathbf{X}} \log P(x_j|c_i) + \log P(c_i) \\ &= \sum_j E_{X_j} \log P(x_j|c_i) + \log P(c_i) \end{aligned}$$

Thus, we only need the distribution of each feature $X_j \sim p(x_j)$ instead of the joint density function $\mathbf{X} \sim p(\mathbf{x})$ to compute δ_1 and Q_1 .

4 Prediction in the Feature Space

In Section 3, we assume that we know the distribution of the feature values when their real values are not available. The computation of the QoD and the choice of load shedding are based on the distribution. In this section, we study how to obtain the feature distribution.

If the feature values of the current time is independent of those in the next time unit, the best we can do is to use the prior distribution of the feature values². This is commonly assumed in data stream management systems, where data-value histograms are often created to assist in query answering.

However, in many real life applications, feature values often have short-term temporal correlation. For example, temperatures of a region and water levels of a river usually do not change dramatically over a short period of time. Feature values extracted from consecutive satellite images also have strong temporal correlation. On the other hand, data characteristics of a stream usually change with time. Thus, our task is to capture short-term time correlation in a time-varying environment.

In this section, we propose a finite-memory Markov model and introduce an algorithm to incrementally update the Markov model so that it reflects the characteristics of the most recent data.

4.1 The Markov Model Markov models have been used extensively in many fields to model stochastic processes [11]. In this study, we use discrete-time Markov-chains with a finite number of states. A discrete-time Markov-chain is defined over a set of M states s_1, \dots, s_M , and an $M \times M$ state transition probability matrix P , where P_{ij} is the probability of transition from state s_i to s_j . We use one Markov-chain for each feature in each data stream. The Markov-chains are used to model both categorical and numerical features. For continuous values, we discretize them into finite number of bins.

Consider any feature x and its corresponding Markov-chain. Assume the feature value at time t_0 is known to us, and we have $x = s_i$, $1 \leq i \leq M$. Thus, the distribution of the feature value at t_0 is $p_0(x) = e_i$, where e_i is a $1 \times M$ unit row vector with 1 at position i and 0's at other positions. The distribution of the feature value in the next time unit t_1 is $p_1(x) = p_0(x)P = e_iP$, where P is the state transition probability matrix. In the next time unit t_2 , the distribution of the feature value becomes $p_2(x) = p_1(x)P = e_iP^2$.

If we shed load at time t_1 , $p_1(x)$ will give us a distribution of the value of x at t_1 . At time t_i ,

²In such a case, the quality of decision for each classifier will not change with time.

the distribution is $p_i(x)$. When i becomes large, the distribution will converge to $p(x) = \pi$, where π is the steady-state solution of the Markov-chain, i.e., π is the solution to

$$\begin{cases} \pi = \pi P, \\ \sum_j \pi_j = 1. \end{cases}$$

It is clear that π is the prior distribution (among the historic data based on which we have built the Markov-chain) of the feature values. In other words, the probability of a certain feature value in the next time unit is approximately the fraction of its occurrence in the historic data. This makes sense, because as the gap between the current time and the time when we last investigated the feature values becomes larger, the temporal correlation will disappear.

In this study, we assume that for a given data stream, the Markov-chains for the features are independent. In other words, we assume that given the feature values of the data stream at current time, the distribution of each feature of next time unit is independent of the distributions of other features. This assumption makes it easier for us to solve the problem (e.g., to compute δ_2 and Q_2) numerically by using, e.g., Monte Carlo methods. Without this assumption, we have to use some special sampling technique (e.g., for the Gibbs sampler, we need the marginal distribution for each feature given all other features [10]), with the independence assumption, sampling is easier, i.e., we can draw samples for each feature following its own distribution, independent of other features. To study the cases in which the feature distributions are dependent is among our future work.

4.2 Finite-Memory Markov-Chains An important issue in data stream is time-variation, i.e., the data characteristics may drift with time. To handle this phenomena, we adopt a finite-memory Markov-chain model and incrementally update its parameters so that they reflect the characteristics of the most recent data. The main idea is to maintain the Markov-chains using a sliding window of the most recent W transitions and update the parameters of the Markov-chains when new observations are available.

First, we consider a simple case in which there is no load shedding. At time t , the most recent $W + 1$ states are $s(t - W), \dots, s(t - 1), s(t)$, and these $W + 1$ states contain W transitions, i.e., from $s(t')$ to $s(t' + 1)$ for $t - W \leq t' < t$. Assume $s(t - W), \dots, s(t - 1), s(t)$ are generated by a Markov-chain P , it can be shown that the maximum-likelihood estimation (MLE) for P_{ij} is

$$(4.7) \quad \hat{P}_{ij} = \frac{n_{ij}}{\sum_k n_{ik}}$$

where n_{ij} is the number of observed transitions from state s_i to s_j among the W transitions. To obtain the MLE, we only need to maintain a matrix \bar{P} of $M \times M$ counters and update the entries using the most recent observations. For example, assume that $s(t - W) = s_p$, $s(t - W + 1) = s_q$, $s(t) = s_i$, and at time $t + 1$, a new observation becomes available and $s(t + 1) = s_j$. To update \bar{P} , we increase \bar{P}_{ij} by 1 as we insert s_j into the sliding window, and decrease \bar{P}_{pq} by 1 as we remove s_p from the sliding window. To get the MLE, we multiply each row of \bar{P} by a normalizing factor to make the row sum to 1.

However, when load shedding takes place, we may not have consecutive observations. When load shedding is frequent, the observations could be very sparse. To obtain the maximum-likelihood estimation of the parameters based on observations with missing values, we can use, for example, the EM algorithm to compute the unobserved feature values. However, iterative algorithms such as EM are time-consuming, which makes them unacceptable for data stream applications. In addition, such algorithms very often only converge to local maximums.

To solve this problem, we use an approximate approach to update the parameters of the Markov-chains: for each data stream, we maintain a flag to indicate if it has been observed in the previous time unit (we say that a data stream is *observed* or it gets an *observation* at time t if we do not shed load from the data stream at time t); if at time t , a data stream is observed, and it was not observed at time $t - 1$, then we will observe the data stream in two consecutive time units (i.e., t and $t + 1$), whenever possible. In such a case, we say that the data stream has a *consecutive observation request (COR)* at time t . If all CORs are satisfied, the observations from a data stream will be in pairs of consecutive states, with possible gaps among the pairs. Therefore, instead of maintaining $W + 1$ most recent states, we maintain in the sliding window the most recent W transitions, where each transition consists of a pair of states (s_{from}, s_{to}). The method to compute and update \bar{P} is similar to the one introduced above, and we still use Eq (4.7) to estimate the P matrix for a Markov-chain, knowing that it is just an approximation.

Furthermore, because the memory of the Markov-chains is finite, it is possible that some rows of \bar{P} are zero vectors. To handle this case and to represent certain prior knowledge about the models, in our implementation we added some pseudo-counts to \bar{P} , that is, instead of all zeros, some counters in the \bar{P} matrix (e.g., those on diagonal) are initialized with some small positive integers.

5 The Load Shedding Scheme

Our load shedding scheme, Loadstar, is based on the two components introduced in the previous two sections, i.e., the quality of decision and the predicted distribution in the feature space. Pseudo-code for the Loadstar algorithm is given in Figure 3. The inputs to the algorithm are i) N' data streams that contain data at time t ($N' \leq N$), and ii) the capacity C of the system, in terms of the number of data streams that the system can handle, at time t . When $N' > C$, load shedding is needed. The outputs of the algorithm are the decision of the classifier for each data stream at time t .

Figure 3 actually contains two versions of our load shedding scheme: the basic Loadstar algorithm (without lines 8–11), in which the parameters of Markov-chains are fixed, and the extended version (with lines 8–11), which we call Loadstar*, in which the parameters of Markov-chains are updated in real time. For the basic version Loadstar, we assume that the parameters of Markov-chains do not drift with time so they are learned from training data; for the extended version Loadstar*, we assume that the parameters of Markov-chains drift with time and they are updated using the most recent observations.

Algorithm **Loadstar**(N', C)

inputs: data from N' streams,
and system capacity C ;
outputs: decisions $(\delta_1, \dots, \delta_N)$;
static variables: feature distributions $p(\mathbf{x})$'s,
Markov-chains MC's,
COR flags (f_1, \dots, f_N) ;

- 1: update $p(\mathbf{x})$ for each feature \mathbf{x} using its MC;
- 2: compute decisions $(\delta_1, \dots, \delta_N)$
and QoDs (q_1, \dots, q_N) using $p(\mathbf{x})$'s;
- 3: select C streams from N' data streams
based on (q_1, \dots, q_N) and (f_1, \dots, f_N) ;
- 4: **for each** selected stream i **do**
- 5: observe the feature value for stream i ;
- 6: revise δ_i for stream i ;
- 7: revise $p_i(\mathbf{x})$ for stream i ;
- 8:* **if** stream i has had load in the
 previous time unit **then**
- 9:* update MC's for stream i ;
- 10:* $f_i \leftarrow false$;
- 11:* **else** $f_i \leftarrow true$;
- 12: **return** $(\delta_1, \dots, \delta_N)$;

Figure 3: The Loadstar and Loadstar* Algorithms

Some internal variables are maintained as static by the algorithm. Among them, $p(\mathbf{x})$'s are the distributions of the features in the current time unit; MC's represent the Markov-chains learned from training data for Loadstar or the Markov-chains in the current time unit for

Loadstar*; (f_1, \dots, f_N) are a vector of COR flags for the data streams in Loadstar*, and in Loadstar, they are all set to *false*.

At time $t - 1$, the feature distributions at time t are predicted by updating the $p(\mathbf{x})$'s using the Markov-chains (line 1). Each stream first assumes that it will not be observed at time t ; it computes the decisions using Eq (3.3) or Eq (3.5) and the qualities of decision using Eq (3.4) or Eq (3.6), both based on the predicted feature distributions (line 2). Then when N' and C are available at time t , if $N' > C$, load shedding is applied. C streams are selected to be observed based on the COR flags and the QoDs: if among the N' data streams, the number of streams with *true* COR flags is less than C , then their requests are fulfilled first and the remaining resources are assigned to other streams based on their QoDs; otherwise, the C streams will be only selected from the data streams whose COR flags are *true*, based on their QoDs (line 3). When deciding which streams to be observed based on QoDs, we use a weighted randomized algorithms where the chance for a stream to be observed is inversely proportional to its QoD value. We choose to use a randomized scheme in order to avoid starvation of a data stream. For the data streams that are observed, because they obtain the real feature values, their feature distributions are changed to unit vectors, and their classification decisions are updated using the new feature distributions (lines 5–7). For Loadstar*, after the data streams to be observed are selected, their COR flags are updated, and if necessarily, their MC's are updated (lines 8–11). Finally, the classification decisions are returned (line 12).

Time Complexity We study the overhead introduced by the load shedding algorithm shown in Figure 3. For each feature of each data stream, updating $p(\mathbf{x})$ (line 1) takes $O(M^2)$ time, where M is the number of states in the Markov-chain. In the Loadstar* version, updating the Markov-chains (lines 8–11) takes $O(1)$ time, because it only involves updating a counter and a flag. The most time-consuming step is to compute the decision and the QoD for each data stream (line 2), because it requires integrations over the whole (possibly multi-dimensional) feature space. This situation, however, can be alleviated by making some assumptions. For example, we have shown that the conditional independence assumption on the feature values makes Q_1 easy to compute; furthermore, we have argued that the conditional independence assumption on the feature distributions makes the numerical integration easier. In the section of experimental studies, we will show that a numerical integration using only a few samples can give us a very accurate approximation and save us from computing the exact integration over the whole feature space.

6 Experimental Results

In this section, we use both synthetic and real-life data sets to study the performance of the Loadstar algorithm. We compare Loadstar with a naive algorithm, in which loads are shed from each data stream equally likely. Both algorithms are implemented in C++. In the experiment setup, for easy of study, instead of varying loads, we fix the load (to be 100 data streams for both the synthetic and the real-life data sets) and change the number of data streams that the system can handle at each time unit. In other words, we study the system under different levels of overloads. In addition, because of the random nature of the algorithms, for all the experiments we run 10 times with different random seeds and report the average values.

6.1 The Synthetic Data Set By using synthetic data, we sought to answer the following experimental questions about our load shedding algorithm:

- (1) Does Loadstar improve the performance over the naive algorithm? If so, how is the improvement achieved?
- (2) Do the Markov-chains capture the models of feature space accurately? Do they adapt to drifts of data characteristics?

We generate data for 100 data streams, and for each data stream, we set the number of features d to be 3. Among the three features, x_1 and x_2 are numerical and x_3 is categorical. The two numerical features are generated using the following random walk model:

$$(6.8) \quad x_t = x_{t-1} + \varepsilon, \text{ where } \varepsilon \sim N(0, \sigma^2)$$

where $N(\mu, \sigma^2)$ is a Normal distribution with mean μ and variance σ^2 . In addition, we add boundaries at 0 and 1 in the random walk model, i.e., at a given time unit t , if $x_t > 1$ or $x_t < 0$, we switch the sign of the corresponding ε and make x_t between 0 and 1. We partition the 100 streams into two families: for the first family, which consists of 10 data streams, the σ in Eq (6.8) is set to be 0.1; for the second family, which consists of 90 data streams, $\sigma = 0.01$. For obvious reasons, we call the first family the *volatile* streams and the second family the *non-volatile* streams. As can be seen soon, such a setup reveals the mechanics that Loadstar uses to obtain good performance. For the categorical feature x_3 , which consists of 4 distinct values s_1, \dots, s_4 , all data streams have the same characteristics: the feature values are generated as time series using a Markov-chain whose P matrix has the following form: the element on diagonal is 0.91 and all other elements have value 0.03.

To generate the model for the classification problem, we use two class labels, + and -, and we assume the features to be independent given the class label. For the two numerical features, their likelihood functions are given as $p(x_1|+) \sim N(0.2, 0.1^2)$, $p(x_1|-) \sim N(0.8, 0.1^2)$, $p(x_2|+) \sim N(0.8, 0.1^2)$, and $p(x_2|-) \sim N(0.2, 0.1^2)$. For the categorical feature x_3 , its likelihood functions are given as $p(s_1|+) = p(s_3|+) = 0.4$, $p(s_2|+) = p(s_4|+) = 0.1$, $p(s_1|-) = p(s_3|-) = 0.1$, and $p(s_2|-) = p(s_4|-) = 0.4$. Because of the symmetry of the model, we assume that the prior distribution for the two classes to be equally likely. Finally, the real class label for each feature triplet is assigned to be the class that has higher joint posterior distribution value.

We generate data for 11,000 time units, where data for each time unit consists of 100 observations for the 100 data streams. Data in the first 6,000 time units are used as training data to build a naive Bayesian classifier. For the naive Bayesian classifier, we use 10 bins with equal width to discretize the two features with numerical values. Although our algorithm allows each data stream to have its own classifier, for simplicity, in the experiments we use a single naive Bayesian classifier for all data streams. Data in the last 5,000 time units are used as test data. We set the window size W for Markov-chain learning in Loadstar* to be 100.

Performance Comparison In this study, we compare Loadstar (and its extension, Loadstar*) with the naive algorithm in terms of error rates under different levels of overload. For this, we fix the load to be 100 data streams, and increase the number of data streams to have loads shed at each time unit from 0 to 80. Figure 4(a) and Figure 4(b) show the error rates of the classifier under different levels of overload, using δ_1 , Q_1 and δ_2 , Q_2 , respectively.

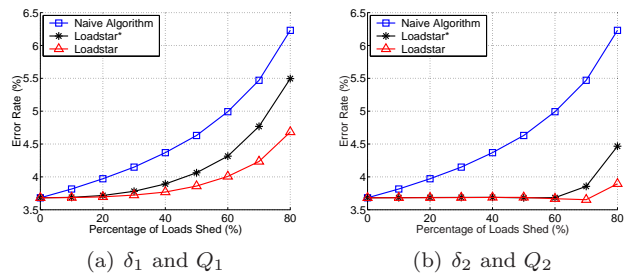


Figure 4: Performance Comparison

From the figures we can see that in both cases Loadstar has lower error rates than the naive algorithm under different levels of overload. Loadstar that uses δ_2 and Q_2 has better performance than that uses δ_1 and Q_1 . In particular, for the former one, when the percentage of loads shed is under 70%, the error rate remains the same as that of the case with no load

shedding. Because of this, in the remaining discussion, we focus on δ_2 and Q_2 . Also can be seen from the figures, the error rates of Loadstar* are higher than those of Loadstar. This result is not unexpected, because for learning Markov-chains, Loadstar* requires consecutive observations. That is, with 80% loads shed, for Loadstar, on average each data stream is observed every 5 time units; for Loadstar*, each stream is observed consecutively every 10 time units. As we know, because of the temporal locality, consecutive observations every 10 time units does not provide as much information as two separate observations with distance of 5 time units.

To shed light on the reasons for Loadstar’s good performance, in Figure 5(a) we plot the percentage of observations that are assigned to the volatile data streams under different levels of load shedding. As can be seen from the figure, the naive algorithm always assigns 10% observations to the volatile streams because there are 10 out of 100 data streams that are volatile. In contrast, for Loadstar, as the number of available observations becomes smaller, a higher fraction of them are assigned to the volatile streams. For example, when there are only 20 observations available, on average, at each time unit the naive algorithm assigns 2 of them to the volatile streams, but Loadstar assigns more than 5 observations to the volatile streams.

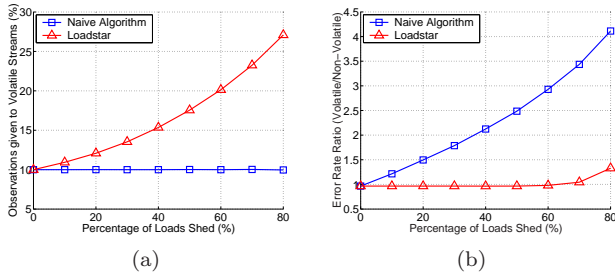


Figure 5: (a) Percentage of Observations Assigned to Volatile Data Streams, and (b) Error Rate Ratio between Volatile and Non-Volatile Data Streams

In addition, we compute the error rates for the volatile and non-volatile families separately. Figure 5(b) shows the error rate ratio between the volatile family and the non-volatile family, under different levels of load shedding. As can be seen, for the naive algorithm, because it sheds loads from all data streams equally likely without considering their data characteristics, as the percentage of load shedding increases, the error rate of the volatile family suffers more and more comparing to that of the non-volatile family; in contrast, for Loadstar, because the quality of decision automatically includes the characteristics of data into consideration, the error rate ratio between the two families remains around 1 until the percentage of load shedding increases

to 60%, and does not go beyond 1.5 even when the percentage of load shedding increases to 80%.

In summary, when different data streams have different characteristics, Loadstar is more fair in that it gives more available resources to the data streams that are less certain, and as a result, it balances the error rates among the data streams with different characteristics and achieves better overall performance.

Markov-Chain Learning In this experiment, we study the Markov-chain learning part of our load shedding scheme. We generate the data streams such that x_3 has time-varying characteristics, using the following two Markov-chains:

$$P_A = \begin{bmatrix} .91 & .03 & .03 & .03 \\ .03 & .91 & .03 & .03 \\ .03 & .03 & .91 & .03 \\ .03 & .03 & .03 & .91 \end{bmatrix}, P_B = \begin{bmatrix} .25 & .25 & .25 & .25 \\ .25 & .25 & .25 & .25 \\ .25 & .25 & .25 & .25 \\ .25 & .25 & .25 & .25 \end{bmatrix}$$

For the test data, for the first 1,000 time unit, we generate x_3 using P_A (P_A is also used to generate the training data); then at time unit 1,000, we switch to P_B ; finally, at time unit 3,000, we switch back to P_A .

To quantify the performance of Markov-chain learning, we use the Kullback-Leibler divergence as the measure of error. Notice that each row P_i of the P matrix is a distribution; in our algorithm, we have a estimation matrix \hat{P} and each row \hat{P}_i of \hat{P} is also a distribution. To see if the two distributions are near to each other, we compute their Kullback-Leibler divergence $d(P_i, \hat{P}_i) = \sum_j P_{ij} \log(\frac{P_{ij}}{\hat{P}_{ij}})$. And finally, we sum the Kullback-Leibler divergences over all the rows and all the data streams. Figure 6 shows the results over time units 500 to 5,000 for Loadstar and Loadstar*. For Loadstar*, we report the results for two cases: the case in which there is no load shedding and the case in which there is 50% load shedding.

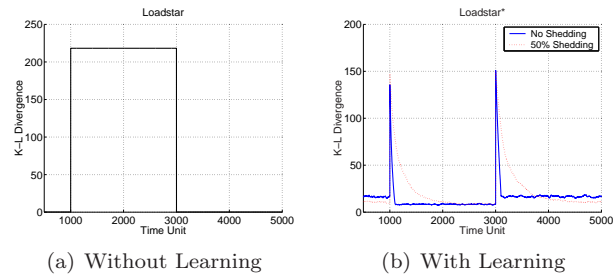


Figure 6: Learning the Markov-Chains

As can be seen from Figure 6(a), because Loadstar learns the parameters of Markov-chains from the training data and because P_A is used to generate the training data, before time 1,000, the error is very small; the error increases sharply when the parameters are changed at

time 1,000, and remains high until at time 3,000, when the original parameters are restored. In contrast, as can be seen from Figure 6(b), Loadstar* can learn the new parameters of Markov-chains in real time: when the parameter change happens, the error of Loadstar* also increases sharply; however, when there is no load shedding, as we expected, after 100 time units (which is the sliding window size W), the new parameters are learned and the error drops back; this learning takes longer time for the case of 50% load shedding.

It is interesting to observe from Figure 6(b) that when the Markov-chain has parameter P_A , Loadstar* has more accurate estimation for the parameters when there is 50% load shedding than when there is no load shedding. To explain this, we have to see the difference between the two cases: in the case of no load shedding, data from the most recent 100 time units are used to learn the parameter; in the case of 50% load shedding, on average, samples from the most recent 200 time units are used. When the distributions are skewed (e.g., P_A), the temporal locality prevents us from learning the parameter very accurately using only 100 time units; when there is 50% load shedding, samples are drawn from longer history (on average 200 time units) and therefore the parameters can be learned more accurately. To verify this, we look at Figure 6(b) between time units 2,000 and 3,000. During this period, P_B is used and from the parameters we can see that when P_B is used, there is no temporal locality at all. Therefore, as expected, during this period both cases learned the parameters equally accurately.

Monte Carlo From Eq (3.5) and Eq (3.6) we can see that to compute δ_2 and Q_2 , we need to do an integration (or weighted sum) over all the feature space. We now show that a sampling method can help us reduce the computation. We use a Monte Carlo method that instead of integrating over the whole feature space, just samples some points from the feature space, and compute unweighted average of δ_2 and Q_2 over these points. In our implementation, because of the conditional independence assumption on the feature distributions, to draw a sample point (x_1, x_2, x_3) , we can draw x_1 following $p_1(x)$, x_2 following $p_2(x)$, x_3 following $p_3(x)$ (all with replacement) and then put them together. Figure 7(a) and Figure 7(b) show the results for the Monte Carlo method with 5 sample points and 10 sample points, respectively. As can be seen from the figures, with only 5 sample points, the Monte Carlo method has already clearly outperformed the naive method, and with 10 sample points, the performance of the Monte Carlo method becomes very close to that of the original Loadstar algorithm in which integration is taken over the whole feature space. This experiment demonstrates

that our load shedding scheme is particularly suitable for data stream applications, in which quick response time is crucial.

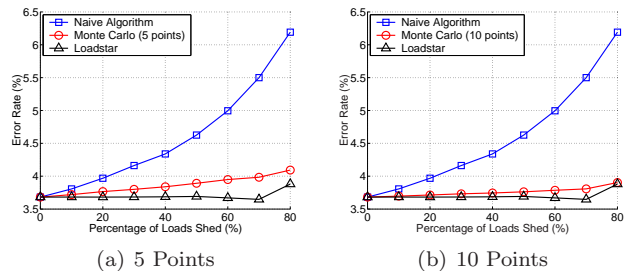


Figure 7: The Performance for Monte Carlo

6.2 The NASDAQ Data Set For real-life data, we use a data set of stock streaming price data. We recorded 2 weeks of price data for 100 stocks (NASDAQ-100) as well as the NASDAQ index. For each stock, the close price at the end of each minute is recorded. The streaming price for each stock is considered as a data stream. Therefore, there are 100 data streams and for each data stream, there are observations for 3,900 time units (10 business days \times 6 $\frac{1}{2}$ hours per day \times 60 minutes per hour) with a time unit of 1 minute. The price for each stock is normalized with respect to the stock's open price on the first day. In other words, after normalization, the price of a stock at time unit t will be $\frac{v_t}{v_1}$, where v_i is the stock's real price at time unit i .

We define the classification problem as the following. At a given time t , a stock is called *outperform* if its normalized price is higher than that of the NASDAQ index, *underperform* otherwise. The classification problem is defined as at each time unit t , predicting the class (*outperform* or *underperform*) of each stock at time $t+1$.

Here is the way how we build our classifier. We assume that the NASDAQ index follows the random walk model given in Eq (6.8). (Stock price is one of the best-known examples of time series that behave like random walks [4].) We assume the noises at different time units are independent. Because we do not have the noise variance σ^2 , at any given time t we use the sample variance $\hat{\sigma}^2$ of the NASDAQ index in the hour before t as an estimation for σ^2 . If we have y_t , the NASDAQ index value at time t , then according to our model, the NASDAQ index value at time $t+1$ follows a Normal distribution:

$$y_{t+1} \sim N(y_t, \hat{\sigma}^2)$$

For our Bayesian classifier, we choose the posterior probability, as shown in Figure 8, as our discriminant function (here we assume an equal prior distribution for *outperform* and *underperform*). For example, assume

$y_t = 1.2$ and we know the value of a stock at time $t+1$ to be $\tilde{x}_{t+1} = 1.3$, then if we decide the class to be *outperform*, the probability for the decision to be correct is the area under the curve of the distribution of y_{t+1} for which y_{t+1} is less than 1.3; if we decide the class to be *underperform*, the probability for this decision to be correct is the area under the curve where y_{t+1} is greater than 1.3. Obviously, conditioning on the value of a stock \tilde{x}_{t+1} at time $t+1$, the decision will be *outperform* if $\tilde{x}_{t+1} > y_t$, and *underperform* otherwise (here we use \tilde{x}_{t+1} because we do not know the real value x_{t+1} , i.e., we are making decisions about time $t+1$ at time t).

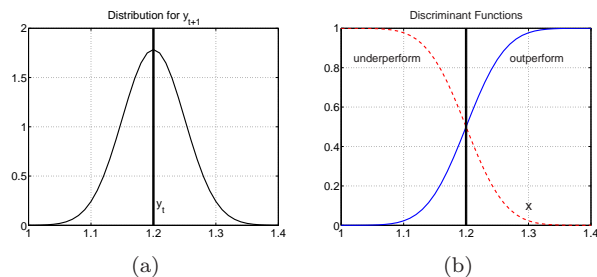


Figure 8: Bayesian Classifier for the Stock Data Set

For our load shedding scheme, we choose δ_2 and Q_2 as they are defined before. For the Markov-chains, because the feature values (i.e., normalized stock price at time t) are continuous, we discretize them into 20 bins with equal width where each bin corresponds to 1 percentile. In this experiment, because the prices for all stocks behave similarly, for simplicity we use a single Markov-chain for all data streams, where the parameters of the Markov-chain are learned using the first hour of data. Again, as a base case, we defined a naive load shedding algorithm, which chooses data streams to have observations shed equally likely.

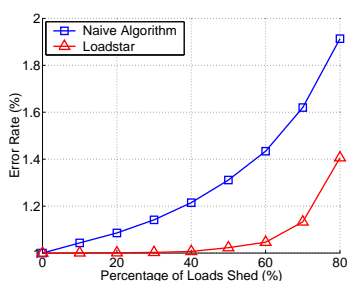


Figure 9: The Performance of Loads for the Stock Data

The experimental results are shown in Figure 9. As can be seen from the figure, because the stock prices do not change very dramatically in a time interval of 1 minute, the error rate for this classification problem is not very high. However, as load shedding becomes more severe, the error rate for the naive algorithm grows

continuously. In contrast, when the load shedding level is between 0% and 40%, there is no obvious change in error rates for our load shedding algorithm. In the whole load shedding range, Loadstar always outperforms the naive algorithm.

7 Related Work

In [8], Jain et al. proposed using Kalman filters to adaptively manage resources in data stream management systems, where the main goal is to minimize bandwidth usage under a given precision requirement. In [12], Olston et al. proposed an adaptive-filter scheme for continuous queries over distributed data sources, where the main concern is also the tradeoff between the precision of the answers to queries and the communication cost. These studies are similar to ours in that they use mathematical models to model the data sources and adaptively allocate resources accordingly. The main difference of these methods with ours is they assume that the data sources have processors to do complicated computation (to filter data based on thresholds or to solve linear equations). This assumption may be acceptable for simple numerical values; however, for complex data types, such as multimedia data, whose feature values must be derived using specialized software or hardware, such an assumption become invalid. In addition, these studies all assume numerical data; in our algorithm the features can have either numerical values or categorical values.

In [3], Babcock et al. studied the load shedding problem in systems that process continuous monitoring queries over data streams. The main idea of the study is that when overload happens, inserting load shedders in various locations of the query plan, such that the maximum relative error among all queries is minimized (with high probability). However, this study was restricted to sliding window aggregate queries and did not consider queries that involve the join operation among multiple streams. In [14], Tatbul et al. described a scheme for load shedding in the Aurora Data Stream Management System. In the study, the load shedding is based on the QoS specifications on latency, values, and loss-tolerance. This work is similar to ours in that it adjusts load shedding according to the status of each sub system of the whole system. However, it assumed static QoS curves (e.g., concave or piece linear curves) are available to guide load shedding. In contrast, how to defined the quality measure for data mining tasks is a major part of our work.

Another topic that is closely related to our work is concept drifts in mining data streams. In [15], Wang et al. proposed an algorithm for mining concept-drifting data streams using weighted ensemble classifiers. In

[7], Fan et al. proposed an active mining method that detects potential changes in data streams. These studies are similar to ours in that statistics are defined to measure the characteristics of current data streams. However, they assume that the correct class labels are readily available for newly arrived testing data and therefore, as the concepts in data streams change, it is possible to revise the decision models correspondingly in real-time. In our study, we do not assume the availability of the correct class labels for test data, and therefore our classifier is fixed beforehand based on training data. Instead, we assume that at different time, the feature values are moving around different regions of the feature space. In other words, it is the region of the concept we are currently in, not the concept itself, that changes with time.

8 Conclusion and Future Directions

In this paper, we studied the resource allocation problem in mining data streams and in particular, we developed a load shedding algorithm, Loadstar, for classifying data streams. The Loadstar algorithm consists of two main components: i) the quality of decision (QoD) measures that are defined based on the classifier, the feature space, and the predicted feature distribution of the next time unit, and ii) the feature predictor which is based on finite-memory Markov-chains, whose parameters can be updated in real time. Extensive experimental results on both synthetic and real-life data sets showed that Loadstar has better performance than a naive algorithm in term of classification accuracy, where its superior performance is achieved by automatically focusing on data streams that are more uncertain while shedding data streams whose class labels in the next time unit are more certain. In addition, experiments showed that the Loadstar algorithm can efficiently learn parameters of its Markov-chains and computation in Loadstar can be reduced by using Monte Carlo methods.

For future work, we plan to extend our study in the following directions. First, in this paper we assume that the streams are independent; however, in many real-life applications, one mining task may need multiple data streams and each data stream can be involved in multiple data mining tasks. To take these relationships into consideration in our algorithm is one of our future directions. Second, in this paper we assume the data mining task (the classification) is the last stage of the system. In the future, we plan to consider systems in which data mining is just an intermediate computation, e.g., as a filter to decide which data streams to be sent for more detailed analysis. Third, in this paper we consider a simple case that at each given time, we either

apply load shedding to a data stream or not; in the future, we plan to extend our load shedding algorithm to control the communication rates of the data streams, e.g., given many video streams, the frame rate of each stream is proportional to its importance.

References

- [1] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.
- [2] C. C. Aggarwal and P. S. Yu. On effective conceptual indexing and similarity search in text data. In *Proc. of the 2001 IEEE Intl. Conf. on Data Mining*, 2001.
- [3] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *20th International Conference on Data Engineering*, 2004.
- [4] C. Chatfield. *The Analysis of Time Series: An Introduction*. Chapman & Hall/CRC, 2004.
- [5] P. Domingos and M. Pazzani. On the optimality of the simple Bayesian classifier under zero-one loss. *Mach. Learn.*, 29(2-3):103–130, 1997.
- [6] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. John Wiley & Sons, Inc., 2001.
- [7] W. Fan, Y-A Huang, H. Wang, and P. S. Yu. Active mining of data streams. In *Proceedings of the Fourth SIAM International Conference on Data Mining*, 2004.
- [8] A. Jain, E. Y. Chang, and Y-F Wang. Adaptive stream resource management using kalman filters. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, 2004.
- [9] P. Langley, W. Iba, and K. Thompson. Analysis of Bayesian classifiers. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 1992.
- [10] J. S. Liu. *Monte Carlo Strategies in Scientific Computing*. Springer-Verlag, 2001.
- [11] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
- [12] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003.
- [13] W. K. Pratt. *Digital Image Processing*. John Wiley & Sons, 1991.
- [14] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proc. of the 29th Intl. Conf. on Very Large Databases (VLDB'03)*, 2003.
- [15] H. Wang, W. Fan, P. S. Yu, and J. Han. Mining concept-drifting data streams using ensemble classifiers. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2003.