LOCUST: An Online Analytical Processing Framework for High Dimensional Classification of Data Streams

Charu C. Aggarwal, Philip S. Yu

IBM T. J. Watson Research Center
19 Skyline Drive, Hawthorne, NY 10532, USA
{ charu, psyu }@us.ibm.com

Abstract— **In recent years, data streams have become ubiquitous because of advances in hardware and software technology. The ability to adapt conventional mining problems to data streams is a great challenge in a data stream environment. Many data streams are inherently high dimensional, which creates a special challenge for data mining algorithms. In this paper, we consider the problem of classification of high dimensional data streams. For the high dimensional case, even traditional classifiers do not work very well on fixed data sets. We discuss a number of insights for the intractability of the high dimensional case. We use these insights to propose a new classification method (LOCUST) which avoids many of these weaknesses. The key is to develop a subspace-based instance centered classification approach which can be implemented efficiently for a fast data stream. We propose a methodology to effectively process the data stream in an organized way, so that the intermediate data structures can be used to sample *locally discriminative* subspaces for the classification process. We show that LOCUST is able to work effectively in the high dimensional case, and is also flexible in terms of increased robustness with greater resource availability.**

I. Introduction

In recent years, advances in hardware technology have allowed us to automatically collect large amounts of data continuously. These continuously growing data sets are referred to as *data streams*. The data stream problem has been extensively researched in recent years because of the large number of relevant applications [2], [3]. An important data mining problem is that of classification [4], [6], [7], [14]. The classification problem has also been widely studied in the context of data streams [1], [5], [8], [11], [12], [15]. A survey of different data stream classification algorithms may be found in [2].

The classification problem faces a number of unique problems in the high dimensional case. This is because of the exponential number of attribute combinations which can be related to the class variable. For example, consider the set of *locality based classifiers*. These are defined as classifiers which use local proximity within one or more sets of dimensions for model construction. Some examples of such classifiers are decision trees, rule based systems and nearest neighbor classifiers. Each of these classifiers uses the local behavior of a test instance (either in the full dimensional space, or a well chosen local subspace) to determine its classification. In such cases, the large number of potential combinations of attributes creates a natural tradeoff between *model incompleteness* and *computational requirements*. For example, consider the case of a decision tree classifier. In this case, we encounter the tradeoff as follows:

(1) Model Incompleteness: Each path in the tree represents a local subspace for classification purposes. While classifying a test instance, an incorrect decision at a higher level of the tree could lead to a path which defines a poor choice of subspace. In general, it is rare that the path for a particular test instance provides the best possible subspace for classifying it. This is because the training process attempts to construct a globally optimal tree, and is myopic to the particular behavior of specific test instances. This problem increases with dimensionality, when the number of possible discriminative combinations of attributes increases.

This problem arises because the splits at the highest level of the tree are often not equally effective for all parts of the data. The optimal structure of a decision tree is highly dependent upon the particular data locality which is subject to classification. The number of possible decision trees varies exponentially with data dimensionality, and each tree may be better suited to a different locality of the data. Many specific characteristics of the test instance cannot be captured during the pre-processing phase on the training data.

(2) Computational Requirements: A natural solution to this problem is to build multiple decision trees, and construct forests for classification purposes. Often, more robust classifiers are obtained by using majority voting over many groups of decision trees [4], [14]. However, this method is at best an incomplete solution to the problem. With increasing dimensionality the (time and space) scalability required in the number of trees becomes unmanageable. Furthermore, if the data stream evolves, such a system may significantly degrade for classification purposes.

Similar problems are encountered with the use of rule based classifiers. The left-hand side of each rule typically relates a local subspace to a class variable. While an exponential number of rules can be constructed, only a small set of prioritized rules are retained for practical model representation. This rule set may not work equally well for all test instances, because different rules are more effective for different data localities. Therefore, any rule pruning or prioritization results in classification errors. In general, the most effective rule set is *specific to a particular instance*. This is practically infeasible for compute intensive applications. Finally, while nearest neighbor classifiers are instance specific, they do not scale well and are usually not designed to optimize the discovery of any subspace of the data.

In general, it is desirable to get the best of both worlds. While we would like to perform the classification in an instance specific manner, we would also like to leverage on well chosen subspaces for that instance. We note that instance specific learning is advocated in lazy learning [10], [13], though the approach has not been studied from the incompleteness perspective of high dimensional models. Furthermore, such an approach is not easily applicable to a data stream since this increases the computational needs of a real time classification process.

In our paper, we will propose a framework called LOCUST (LOCal stream sUbspace claSsificaTion), which creates intermediate statistics and data structures during stream processing. These intermediate statistics are leveraged for the final instance-specific classification phase. It may be noted that the construction of the intermediate data structure (training phase) and the test phase can both be performed using one-pass stream computations. Therefore, the framework allows the flexibility of simultaneous processing of training and test streams.

During the test phase, we use multiple instance-centered subspace samples in order to determine the final behavior of the test instance. Each instance-specific subspace sample is *locally discriminative* for that test instance. The statistics from multiple such samples are combined in order to construct a robust and effective classifier. The use of locally discriminative behavior for constructing subspace samples greatly reduces the number of combinations of dimensions which need to be explored. This is slightly different from traditional classification models which tend to construct the *entire preprocessed classifier model* a-priori. Therefore, the methodology has the flavor of an Online Analytical Processing (OLAP) technique in which the results from multiple classifiers are combined together.

Since the classification is based on combining the results from multiple *instance centered models*, it can be used in an adaptive way in which the number of learners combined depends upon the speed of the data stream and computational resources available. Therefore, there is a natural tradeoff between efficiency and accuracy. We will leverage this tradeoff in order to perform resource adaptive classification. When the input streams are bursty, the processing time available for test instance classification is also unevenly distributed. This would lead to excessive variations in classification quality. The adaptive subspace-sampling approach of this paper naturally allows for a technique for management of incoming test instances so that variations in processing times over different test instances are smoothed out. Such an approach is not only adaptive, but is also more effective than techniques such as micro-clustering [1] for the non-adaptive situations for which the latter is designed.

This paper is organized as follows. In the next section, we will introduce the data structures and statistics used to perform resource adaptive classification in a massive data stream. We will show how the approach allows us to leverage the natural tradeoff between accuracy and efficiency. In section 3, we will illustrate the experimental results for the technique. Section 4 contains the conclusions and summary.

II. The LOCUST Framework

We assume that each data point is associated with a unique point identifier which is useful for tracking and indexing purposes. There are k classes in the data which are denoted by $\mathcal{C}_1 \ldots \mathcal{C}_k$. We will first discuss a simple two-pass method for creating the inverted histogram in the case of a massive data set $\mathcal{D}$. Then, we will discuss how to approximate this in one pass for the case of a data stream. Let us assume that we need to discretize the data into Φ equi-depth ranges. The first pass determines the limits of the equi-depth ranges. The second pass uses these limits to construct the equi-depth histogram. Let us assume that these ranges are denoted by $\mathcal{R}_1^j \ldots \mathcal{R}_\Phi^j$ for the jth dimension. The corresponding ranges are denoted by $[l_1^j, u_1^j] \ldots [l_\Phi^j, u_\Phi^j]$, where l_i^j and u_i^j are the lower and upper bounds for the ith range on dimension j. It is easy to see that for each $i \in \{1 \ldots \Phi - 1\}$, we have $u_i^j = l_{i+1}^j$. Let the set of data points in the ith range be denoted by V_i^j. The union of the points over the different ranges is as follows:

$$\mathcal{D} = \cup_{i=1}^{\Phi} V_i^j \tag{1}$$

Each set V_i^j is divided into sublists which correspond to the k different classes in the data. Let us assume that the k sublists of V_i^j are denoted by $W_{i1}^j \ldots W_{ik}^j$. The set of data points in $W_{i1}^j \ldots W_{ik}^j$ are indexed by the corresponding range and class value. For each inverted list W_{ir}^j, we maintain the following additional meta-information:
(1) The number of points in W_{ir}^j. **(2)** The index r of class $\mathcal{C}_r$ for list W_{ir}^j. **(3)** The upper and lower bounds for the range corresponding to W_{ir}^j. **(4)** The identifier list of the data points in W_{ir}^j.

The afore-mentioned construction of the inverted lists requires two passes over the data. For the case of a data stream, we are allowed only one pass. In order to achieve this goal, we process the data in blocks, each of which is smaller than the main memory constraints. In practice, we pick the block size to be a small fraction (say about 5%) of the available main memory. Each block is read into main memory, and the inverted list is created using only main memory operations. Therefore, it is possible to perform the required two passes on a block by using only main memory operations. For the tth block, we denote the corresponding data block by $\mathcal{D}(t)$. The Φ ranges for the tth block are denoted by $\mathcal{R}_1^j(t) \dots \mathcal{R}_\Phi^j(t)$. The corresponding lists are denoted by $V_1^j(t) \dots V_\Phi^j(t)$ respectively. We denote the sublists for the different classes by $W_{i1}^j(t) \dots W_{ik}^j(t)$. As in the previous case, we store the meta-information for each block along with the identifiers of the corresponding inverted lists. We note that the ranges for the different blocks may change over time because of data evolution. This is kept track of because of the meta-information stored in the afore-mentioned statistics.

Once these inverted lists have been constructed, they can be leveraged in order to sample subspaces which are local to that test instance. The subspace classifier constructs discriminative subspaces specific to the locality of the particular test instance. This is achieved with the use of the statistics which are stored in the inverted lists. The final classification of the instance is determined as a robust composite of the classification behavior of different sample subspaces. The training process only concentrates on the construction of this intermediate statistical structure and does not explicitly try to build a model to relate the attributes to the different classes. The summary information can then be leveraged at classification time in an *instance-specific* way. This provides a high level of flexibility. This makes it possible to avoid the inherent limitations of classifiers which try to build the *global* model during the training phase.

A. Instance-Centered Subspace Classification

The LOCUST approach works with the use of repeated subspace samples which are specific to a given test instance. We note that the intersection of two or more inverted lists determine the points in a corresponding subspace. For a given test instance, we perform the intersection of only those lists which are relevant to it. For each of the d dimensions, exactly one list (such that the range contains the corresponding attribute value) is relevant to a particular record. Let us assume that the index of the relevant range for the test instance T corresponding to dimension j is denoted by i_j. The corresponding range is denoted by $\mathcal{R}_{i_j}^j(t)$. The d inverted lists which are relevant to the test instance T at time t are denoted by $V_{i_1}^1(t) \dots V_{i_d}^d(t)$.

In most real data sets, some combinations of dimensions (or subspaces) may have deeper relationships with the class variable. Furthermore, different sets of subspaces may be more relevant to different test instances. This is because of local variations in subspace behavior over different test instances. Because of the use of our pre-processing methodology, we are able to do this in a way which reflects the *local behavior* of a test instance. This is particularly important in the high dimensional case since there are an exponential number of attribute combinations. Therefore, we need to quantify the (local) importance of an inverted list $V_{i_j}^j(t)$. For this purpose, we use the gini-index of that list local to the particular test instance T. The gini-index of each dimension measures the degree of skew across the different classes. Let the fractional presence of the different classes for the jth attribute be denoted by $f_1^j(t,T) \dots f_k^j(t,T)$. This fractional presence is defined as follows:

$$f_r^j(t,T) = |W_{i_j,r}^j(t)|/|V_{i_j}^j(t)| \tag{2}$$

A natural way to define the gini index $\mathcal{G}^j(t,T)$ specific to dimension j, data block $\mathcal{D}_t$, and time t is as follows:

$$\mathcal{G}^j(t,T) = \sum_{r=1}^{k} (f_r^j(t,T))^2 \tag{3}$$

The value of $\mathcal{G}^j(t,T)$ varies between $(1/k)$ and 1. For an equal distribution across classes, the value is $1/k$. For a data set skewed towards a single class, the value approaches 1. We note that this gini-index is *specific* to the local behavior of test instance T, since it uses only a particular set of inverted lists which are unique to that data locality. This is different from most classification algorithms in which the feature importance is determined globally during the training period.

One disadvantage of the above definition of the gini-index is that it can lose its effectiveness, when the *global distribution* of the data is highly biased. For such cases, we use a normalized definition of the gini-index. Let $p_1 \dots p_k$ be the a-priori fractional presence of the different classes. In this case, we define the normalized fractional presence $fn_r^j(t,T)$ as follows:

$$fn_r^j(t,T) = \frac{|W_{i_j r}^j(t)|/(p_r \cdot |V_{i_j}^j(t)|)}{\sum_{l=1}^{k} |W_{i_j l}^j(t)|/(p_l \cdot |V_{i_j}^j(t)|)} \tag{4}$$

When the values of p_l are equal (to $1/k$), there is no difference between Equations 2 and 4. The normalized gini index is computed using the normalized fractional presence in the formula of Equation 3. In all our tests, we used the normalized computation.

```
Algorithm LOCUST(TestInst.: $T$, Database Seg.: $\mathcal{D}_t$,
        Numsamples: $n_s$, SigThreshold: $t$,
        Minpts: $n_{thresh}$);
begin
  for $i = 1$ to $n_s$ do
  begin
    $Q = \Phi$;
    For all classes $l \in \{1, \dots k\}$ $Count(l) = 0$;
    while $((\gamma(Q) \le t) \&\& (|U(Q)| > n_{thresh}))$
    do begin
      Sample the next dimension $j$ with bias
        defined by the corresponding gini-index;
      Let the inverted list corresponding to dimension $j$
      containing test instance $T$ be $V_{i_j}^j$;
      $U(Q) = U(Q) \cap V_{i_j}^j$; (inverted list intersection);
    end;
    Let $\mathcal{C}_r$ be the dominant class in $U(Q)$;
    $Count(r) = Count(r) + 1$;
  end
  report class with largest value of $Count(r)$;
end
```

Fig. 1. LOCUST: Local Subspace Classification

After determining the gini-index, we start subspace sampling process. For a test instance T, let $S(T) = \{i_1 \ldots i_d\}$ be the indices of the inverted lists corresponding to those attribute ranges. We first sample a subset Q from $S(T)$. This sampling is not unbiased, but uses the (local) gini-index in order to bias the instance-specific choice of the dimensions. We will discuss details of the biasing process slightly later. The set Q defines the local instance-specific subspace of dimensions which are picked in the current iteration. The intersection of the data points in this set of dimensions is denoted by $U(Q)$, and is computed as follows:

$$U(Q) = \cap_{i \in Q} V_{i_j}^j \tag{5}$$

We note that the intersection of different lists can generally be computed in an efficient way because of the inverted representation of the data. In general, it is expected that with increasing number of dimensions in Q, the set $U(Q)$ will reduce in size, and will also be more biased towards a particular class variable. In order to measure this bias, we compute the confidence threshold of the dominant class in $U(Q)$. Let f_d be the fraction of data points belonging to the dominant class in the set $U(Q)$. Let f_g be the same value for that class over the entire data set. Then, the confidence level $\gamma(Q)$ of $U(Q)$ is defined as follows:

$$\gamma(Q) = (f_d - f_g)/\sqrt{f_g \cdot (1 - f_g)/|U(Q)|} \tag{6}$$

The denominator in the above expression denotes the standard deviation of the dominant class distribution in $U(Q)$. This standard deviation is computed using the normal distribution assumption. Correspondingly, the value of $\gamma(Q)$ determines the number of standard deviations by which the distribution of the dominant class in $U(Q)$ is greater than the overall distribution as a matter of chance. A value of $\gamma(Q)$ larger than 3 provides a 99.99% level of significance to this probability under the normal distribution assumption.

We note that the set Q is not constructed at one time, but is constructed by sequential intersections of different inverted lists. We continue to add dimensions to Q until one of the two events happen:

- The value of $\gamma(Q)$ exceeds the desired level[1] of significance t.
- The number of records in $U(Q)$ fall below a user defined number n_{thresh}.

The set Q defines a local subspace for the test instance, and we use the dominant class in that subspace to update the class statistics for that test instance. The results from multiple such subspaces are combined in order to derive the final result. As discussed later, the number of sample subspaces n_s picked depends upon the current load on the system. In Figure 1, we have illustrated the LOCUST classification approach. In the current description, we have only illustrated the classification process using a single block denoted by $\mathcal{D}_t$. Later, we will discuss how the results from multiple blocks can be combined in order to determine the final classification result. The input to the algorithm contains the number of sample subspaces n_s, the minimum point threshold n_{thresh}, and the gini threshold t. For each sample, the dimensions are added one by one to the set Q. We note that the dimensions may either be picked in an unbiased way, or by using their relationship to the class variable in the locality of the test instance. The process of biased choice of dimensions will be discussed in detail slightly later. By adding a dimension j to the set Q, the value of $U(Q)$ is computed by intersecting it with the inverted list $V_{i_j}^j(t)$. After each intersection, we compute the value of $\gamma(Q)$. If the value of $\gamma(Q)$ is larger than t, then the current sampling is said to be successful, and we update the count of the dominant class by 1. This approach is repeated for at most n_s samples. At the end of the process, the counts for each class represent the distribution for the different classes for test instance T. The class with the highest count is reported as the relevant class at the end of the process. We note that the choice of the number of subspace samples n_s plays a critical role in both the accuracy and efficiency of the scheme. The choice of larger values of n_s results in a robust system in which the final class is defined by the average behavior of a large number of test-instance specific subspace samples. On the other hand, smaller values of n_s are not quite as accurate, but result in fast processing times. Therefore, we need a resource adaptive mechanism to decide the value of n_s depending upon the load on the system at a given point in time.

The overall approach of LOCUST has an interesting relationship to the method of random forests which has been discussed in the literature [4]. In the random forest approach, the classification is performed by using sampling over a bunch of pre-determined trees. The subspace classifier provides a more flexible framework than the random forest approach since the subspaces are sampled *with knowledge* of the test instance. The level of bias during sampling can be easily controlled in this case and can be tailored to the *specific locality* of the test instance. In addition, the online analytical processing framework provides a technique which makes it applicable to data streams in a resource adaptive way.

One observation about the simplified description of Figure 1 is that it uses only the distribution of the latest block $\mathcal{D}_t$ for the classification process. In practice, it is desirable to use the previous blocks, when the latest data does not reflect the behavior of the current class of the test instance. In a future subsection, we will show how to combine the classification behavior of multiple blocks in a decay based approach.

B. Incorporating Attribute Bias

Each sample subspace is determined as an intersection of a set of inverted lists (or dimensions). For a given sample subspace the subspace is determined progressively by (a second level of) sampling of the corresponding inverted list, and increasing the dimensionality of the current subspace

[1]The desired level of significance t was chosen to be 3. This corresponds to the 99.99% level of significance under the normal distribution assumption.

by 1. The dimensions are sampled until the current sample subspace has the desired classification characteristics. In earlier sections, we have discussed this approach for subspace construction without discussing the issue of bias in the choice of dimensions. In general, we would like to pick dimensions which are more closely correlated to the classification behavior. A larger bias leads to faster processing, but possibly more inaccurate results for the same number of sample subspaces. In general, a variety of heuristics can be used in order to define the effect of bias on the choice of inverted lists:

(1) Unbiased: In this case, all inverted lists (or dimensions) are sampled with equal probability to construct a given local subspace. Therefore, each of the subspaces sampled is truly random. From earlier observations on the effectiveness of using random decision trees [4], it follows that if a larger number of subspaces n_s can be sampled, this works very effectively.
(2) Damped Gini: We used a damped function of the gini index as a bias factor during the sampling process. Therefore, if the gini indices of the d dimensions be denoted by $g_1 \dots g_d$, and $f(\cdot)$ be the damping function, then the probability ps_i of sampling the dimension i is given by:

$$ps_i = f(g_i) / \sum_{i=1}^{d} f(g_i) \tag{7}$$

A typical damping function used is the square-root or the logarithm. We chose the use of the square-root function.
(3) Proportional: In this case, we sample with bias probability proportional to the gini index.

These methods provide different tradeoffs between accuracy and efficiency. When unbiased choice of dimensions is used, the accuracy is high at the end of the process, but a larger number of subspace samples n_s are required to reach the final accuracy. When the choice of dimensions is biased with the use of the gini-index, then a small number of iterations can provide a high level of accuracy, but the final accuracy even with a large number of samples is not as high. Thus, if our approach is used for static data sets with a large pool of available resources, one can choose to use the unbiased method. On the other hand, a greater bias may be needed for faster applications such as data streams. In our experimental results, we found that the use of damped gini-index provides a nice trade-off between accuracy and efficiency. In practice, we used a combination of the three functions in order to achieve the greatest level of flexibility.

C. Resource Adaptivity

We note that the efficiency of the LOCUST method is governed by the number of subspace samples n_s used for the sampling process. A larger number of subspaces provides greater accuracy, but also increases the running time. Therefore, the value of n_s needs to be determined from the current speed of the stream. If the stream is bursty, then the value of n_s also needs to be continuously adjusted to account for the corresponding changes. For this purpose, we keep a historical processing rate of the test instances for each sampling iteration of the algorithm (i.e. for $n_s = 1$). Correspondingly, the value of n_s is continuously adjusted. Let t' be the average time required for each sample during the classification process. The value of t' is computed by using the average processing time in the history of the stream. Then, the expected time for classification of a test instance with the use of n_s samples is given by $n_s \cdot t'$.

At each moment of time, a queue of test instances is maintained in order to perform the classification. We would like to perform the classification process at a rate so that the expected time of processing the queue is equal to a target queue waiting time. Let t_w be the target queue waiting time, and q_c be the current queue length. Thus, the average time for processing the current queue is equal to $q_c \cdot n_s \cdot t'$. Since we would like the current queue to be fully processed by the time that the target time t_w has elapsed, we would like the number of subspace samples n_s to be chosen so that the following relationship is satisfied:

$$q_c \cdot n_s \cdot t' = t_w \tag{8}$$

The value of q_c is always at least 1, since the current test instance being processed is included in the queue. From the above relationship, the value of the number of subspace samples n_s is computed as follows:

$$n_s = t_w / (q_c \cdot t') \tag{9}$$

The value of the number of subspace samples n_s is continuously re-calculated each time stamp with the new value of q_c. The value of t' may also vary slightly with time, in order to account for random variations in processor efficiency. In general, a waiting time of t_w is the steady state required by the application. However, when the stream is bursty, it may lead to waiting times which are larger or smaller than the target values. For example, when a large number of points are suddenly received, the current queue size q_c increases. Correspondingly, the number of subspace samples n_s reduces, and the processing rate speeds up. However, the waiting times may increase. On the other hand, when the stream is slow, the queue lengths are relatively small, and a larger amount of time can be spent on each test instance. In practice, if loadshedding is not used, than the average output processing rate is equal to the input rate. If the input rate is larger than the output rate for a prolonged period of time, then the queue length may exceed the available buffer length, and it may be desirable to reduce the load from the system.

We will also see in the experimental section that the choice of different values of t_w leads to different kinds of tradeoffs on the system. In general, a larger value of t_w leads to a more robust classifier, since the system has the flexibility to gradually adjust the value of n_s. Therefore, for larger values of t_w the processing rate is amortized over the different test instances. As a result, the processing rate does not vary much over time. In the experimental section, we will illustrate the effects of such a choice.

D. Incorporating Time Decay

In many data stream scenarios, recent data may provide the greatest level of insight about the classification of the data stream. There can also be cases in which it may be necessary to use historical data. This is especially true when there is a considerable level of evolution of the underlying stream. Therefore, associated with each data block $\mathcal{D}_t$, we associate a time decay function $2^{\lambda \cdot t}$, where the half-life of the block is $1/\lambda$. The value of λ is determined based on application specific requirements. The decay is accounted for by varying the probability that a block is used in the classification process. Specifically, the probability that the tth block is included during the classification process is equal to $2^{\lambda \cdot (t-t_c)}$. Here we assume that a total of t_c blocks have arrived so far.

Then, we would like to sample the blocks in such a way that the probability of using a block in the classification process is proportional to $2^{\lambda \cdot (t-t_c)}$. In practice, this is achieved by deterministically including the current block, and dropping each of the currently included historical blocks independently with probability $2^{-\lambda}$. It is easy to see that this method of sampling leads to the probability of a block being included equal to $2^{\lambda \cdot (t-t_c)}$. This is because the tth block is removed with probability $2^{-\lambda}$ for a total of $(t_c - t)$ iterations.

Let q be the total number of blocks that the current main memory limitations can hold. The number of blocks $n(t_c)$ required to be held in main memory using the above methodology is given by:

$$n(t_c) = 1 + 2^{-\lambda} + 2^{-2\cdot\lambda} + \dots 2^{-(t_c-1)\cdot\lambda} \quad (10)$$

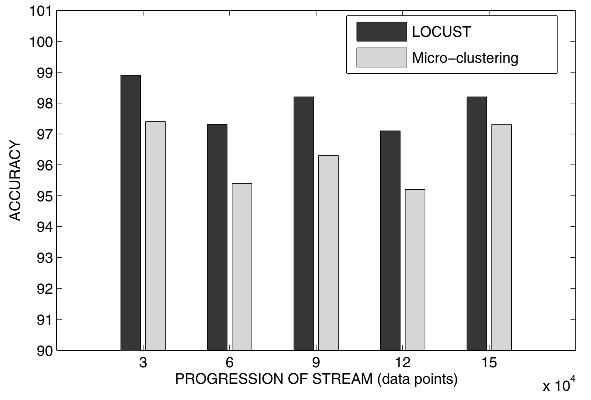
$$= \frac{1 - 2^{-t_c \cdot \lambda}}{1 - 2^{-\lambda}} \quad (11)$$

We note that this is bounded above by the constant $1/(1 - 2^{-\lambda})$. When this constant is less than q, the entire process can be used effectively in main memory. Therefore, in such a case, we can use this scheme with no modifications. When the available memory is restricted, the value of q may be less than the above expression. In such cases, there may be iterations in which there may be a potential overflow beyond the q blocks, when a block is inserted, and no block is correspondingly removed. In such a case, we remove the most stale block from the data. By using this modification, the decay characteristics are still maintained over the blocks in the recent history, though the length of the retained history is curtailed.

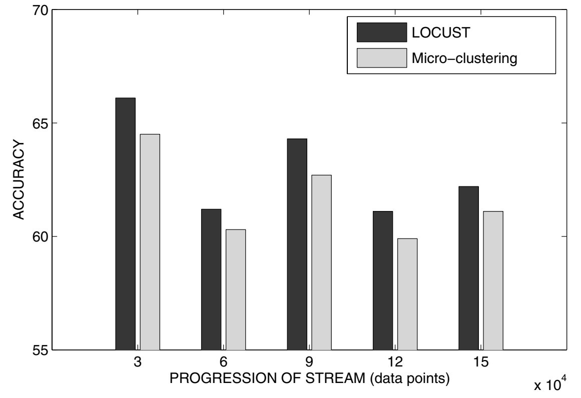
Let $r \leq q$ be the current number of blocks being used for classification purposes. Them if n_s be the total number of samples, then we use n_s/r samples from each block. Since the probability of inclusion of a block is proportional to its recency, this ensures that the probability of using a data point for the sampling based approach is also proportional to its recency.

III. Experimental Results

In this section, we will present the experimental results illustrating the effectiveness and efficiency of the resource adaptive LOCUST method. Our experimental results will compare the classifier with the micro-clustering technique in terms of effectiveness and efficiency. We will also study the effects of varying stream rates on the effectiveness and efficiency of the method.



(a) Network Intrusion data set



(b) Forest Cover data set

Fig. 2. Comp. with Micro-Cluster Classifier

Unless otherwise mentioned, we used a time decay factor $\lambda = 0.2$, $\Phi = 10$, and a blocksize of 20000 points. The results were tested on the Network Intrusion and Forest Cover data sets from the UCI Machine Learning Repository. The data sets were converted into streams by using the same order of the records as the input data.

A. Comparison with Micro-cluster Classifier

While the LOCUST system is designed to work for the case of resource adaptive scenario, it is also useful to test the system against the micro-clustering technique [1] when both streams have the same (testing) speed. This is because the subspace classification system has a tradeoff between accuracy and efficiency, whereas the micro-clustering system does not have such a trade-off. In order to fairly compare the two methods, one must use a setting in which both the systems had similar speed. In this case, we computed the total time required to run the testing phase using the micro-clustering method. Then, we picked the number of samples n_s such that the LOCUST system required a running time which was as

TABLE I

TRAINING RATE COMPARISON

Data Set	MicroClustering (Pts/sec.)	LOCUST (Pts/sec)
Network Intrusion	1105	4421
Forest Cover	2134	7355

close as possible to the running time of the micro-clustering system, but was always slightly smaller than that of micro-clustering. Thus, the value of n_s was fixed throughout the execution unlike the resource allocation methodology that is characteristic of this method. This is because the only aim of these tests is evaluate the effectiveness of the classifier with respect to the micro-clustering technique when the *same amount of resources* are used. This was done by running the stream subspace classification method several times, and picking a value of n_s which provides a computation time similar to the micro-clustering method. We note that while a stream mining algorithm does not allow multiple runs, the purpose of this methodology is only to pick a *fair value* of n_s for comparison purposes, rather than an optimum value. The value of n_s was calculated separately for each data set.

In Figures 2(a), and (b), we have illustrated the effectiveness of LOCUST with respect to the micro-clustering method. In each case, we used the biased classifier in which we used the square-root of the gini index for the purposes of incorporating bias. The reason for this choice will become clear shortly. On the X-axis, we plot the progression of the stream in terms of the number of data points. On the Y-axis, we have plotted the accuracy of the classification process. It is clear that the accuracy of LOCUST is higher than that of the micro-clustering technique in each case. This effectively means that LOCUST was more effective when similar resources were used for the two methods. The reason for the higher accuracy of LOCUST was that it used repeated subspace sampling in order to improve the accuracy of the classification process. We further note that while LOCUST provides the ability to adapt its efficiency to varying stream speeds, this was not the case for the micro-cluster classifier. Thus, our online analytical processing approach is superior to the micro-cluster classifier not only in terms of flexibility, but also in terms of accuracy when both classifiers were used under similar constraints.

We also tested the training rates of the two algorithms. The results are illustrated in Table I. We note that the training rate of LOCUST is significantly higher than that of the micro-clustering method. This is because the micro-clustering method needs to continuously update and store the micro-clusters in a pyramidal scheme. On the other hand, LOCUST uses efficient main memory operations because of its time decay scheme in which only a subset of the blocks need to be maintained. This subset can be used to efficiently classify the data stream. Thus, the results show that LOCUST is more efficient than the micro-clustering method even in the restricted case where a resource allocation strategy is not required.

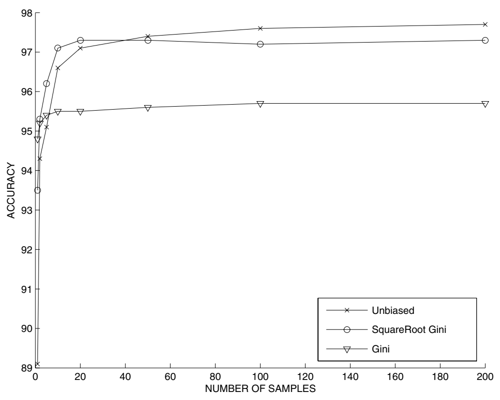
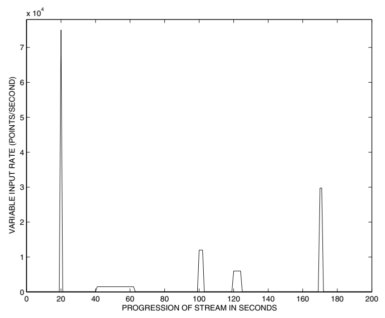


Fig. 3. Effect of Number of Samples

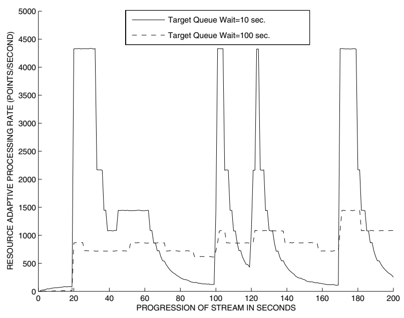
B. Effect of Number of Samples

We tested the effect of the number of samples on the effectiveness of the technique. For this purpose, we tested the effectiveness on a fixed subset of test instances for each of the techniques, while increasing the value of the number of samples n_s. In Figure 3 we have illustrated the effectiveness result for the Network Intrusion data set. In each case, it is evident that the accuracy of the technique increases with the number of samples. However, it is evident that most of the increase is for small values of n_s which are less than 10. For larger values of n_s, the accuracy levels off, and there is no further advantage in increasing the value of n_s. This means that for reasonably small values of n_s, we can maintain modest classification accuracy.

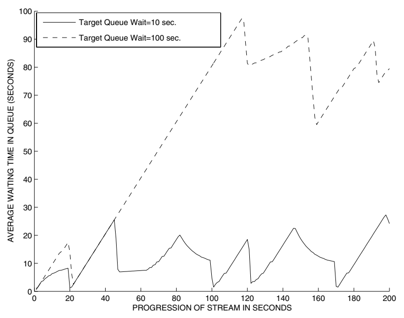
Another observation is that the *relative behavior* of the different methods varies with the number of samples. While the use of highly biased samples (pure gini index for bias) results in quick convergence, the final accuracy of the method is not as high. On the other hand, the use of completely unbiased sampling may result in very accurate results but a corresponding reduction in efficiency. When only a small number of samples can be used, unbiased sampling may not be the method of choice. The highest accuracy of the unbiased sampling method is in agreement with earlier results on techniques such as random forests (or combinations of weak learners) which show that a minimum level of bias results in the highest accuracy (as long as sufficient number of samples are used). Our overall conclusion from this is that the use of the square-root of the gini index provides a nice trade-off between accuracy and efficiency. Therefore, we assumed that other than in extreme cases, we would almost always use the square-root of the gini index for classification. Therefore, for a given test instance, we chose to use either of the three techniques depending upon the value of n_s. For values of n_s less than 3, we used the proportional method for bias. On the other hand, for values of n_s larger than 50, we used the completely unbiased method.



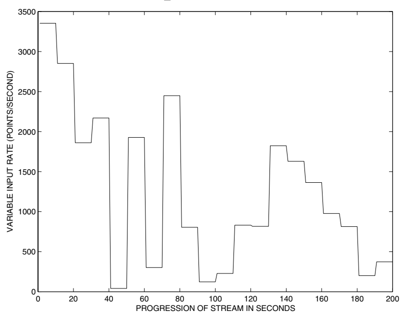
(a) Input Rate (A)



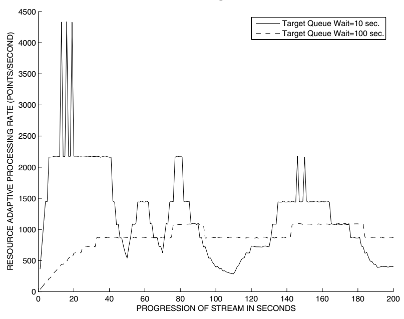
(b) Processing Rate (A)



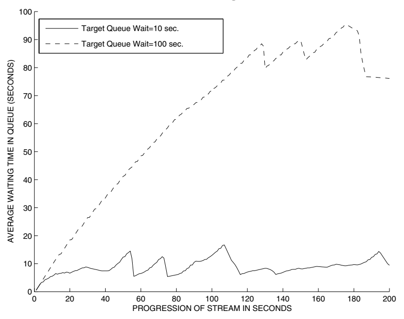
(c) Queue Waiting Times (A)



(d) Input Rate (B)

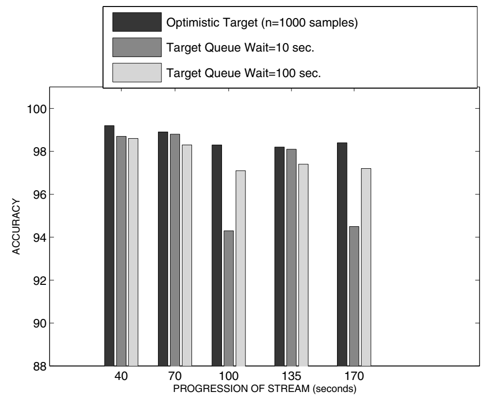


(e) Processing Rate (B)

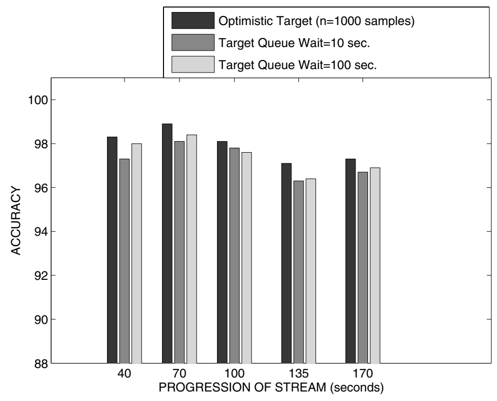


(f) Queue Waiting Times (B)

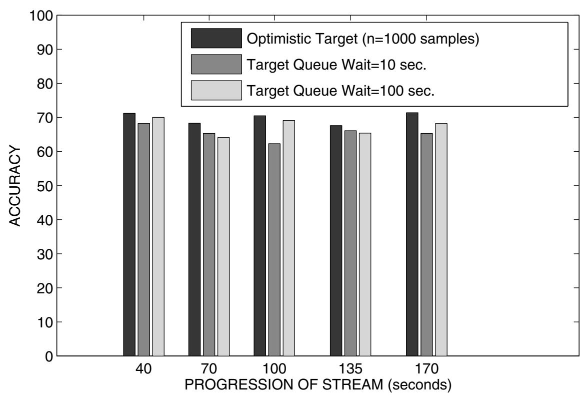
Fig. 4. Resource Adaptive Input and Output Characteristics for Workloads A and B



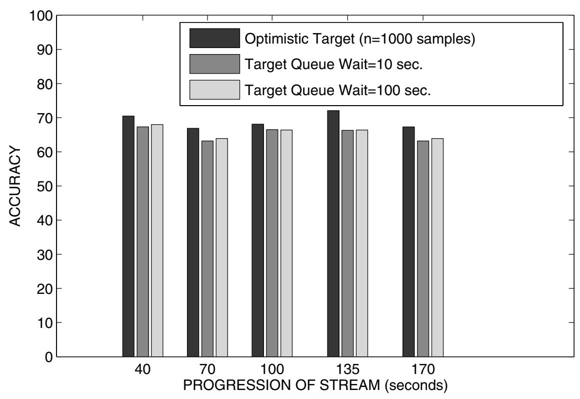
(a) Effectiveness for Network Intrusion (A)



(b) Effectiveness for Network Intrusion (B)



(c) Effectiveness for Forest Cover (A)



(d) Effectiveness for Forest Cover (B)

Fig. 5. Resource Adaptive Effectiveness for Workloads A and B

C. Effect of Varying Stream Rates on Processing Rates of Resource Adaptive Method

We next tested the effect of varying stream rates on the efficiency of the resource adaptive version of the LOCUST method. In many real applications, the peak input rate is much higher than the processing rate. This leads to a need to shed load, unless the resource adaptive method is able to hold points in a queue for later. Our technique uses this approach and saves points in the queue for later. When the input rate is significantly higher than the processing rate, then a burst of data points over several seconds may also lead to corresponding waiting times in the queue of several seconds. As discussed in the section on the resource adaptive method, we provide such target queue waiting times in order to adjust for this flexibility. These target queue times should typically be of the order of several seconds in order to provide the ability to adjust for sudden multi-second bursts in the input data. In order to test robustness, we used a pair of workloads in which the input rate was rapidly varied. In Figures 4 (a), (b), and (c), we have illustrated the results for one workload (workload A) on the network intrusion data set, whereas Figures 4(d), (e), and (f) illustrate the results for the second workload (workload B) on the same data set. As evident from Figure 4(a), workload A has low input rates most of the time, but it has sudden increases in the processing rate at various times. The peak processing rates at these spikes are much higher than the maximum processing rate of 4000 data points per second. This is often the case with many applications in which the processor may remain at low load most of the time, but may witness sudden increases which are beyond the processing capacity. Workload B is somewhat different, and has random variations in the step function representing the input rate. We used two different target queue waiting times of 10 seconds and 100 seconds respectively.

In Figures 4(b), and 4(e) we have illustrated the processing rate for different target queue times for workloads A and B respectively. One interesting observation is that while the processing rates are highly correlated with the input rates for both data sets, the use of larger queue waiting times leads to a smoother variation in the processing rate. This creates a system which has more time to respond to variations in the workload, and therefore provides more robust processing variations over a variety of workloads. We also note that the use of larger queue waiting times leads to a more even distribution of the workload over time. This is true for both the workloads. It is desirable to have a more even distribution of the computational resources across different test instances, since it leads to a robust qualitative performance over different workloads. The trade-off is that a data point has to spend a larger amount of time in the queue before being processed.

In Figures 4(c) and 4(f), we have illustrated the waiting time for each test instance for workloads A and B. It is clear that the waiting times gradually increase at the beginning of the process, and increase to the target value over time. The system them maintains this steady state value with minor fluctuations depending upon the workload. While the target queue waiting times of the order of 10 to 100 seconds may seem large, this should be understood in the context of the relationship of the input rate to the peak processing rate ($n_s = 1$). Since the input rate is often higher than the peak processing rate (with $n_s = 1$) over several seconds in both the two workloads, the theoretical minimum for the target queue waiting times should have a similar magnitude. In order to provide more robust classification, larger values of n_s are needed. Therefore, the use of such target waiting times is quite modest in order to ensure smooth adjustment of processing speeds in cases where there are rapid variations in input rate.

D. Effect of Varying Stream Rates on Accuracy of Resource Adaptive Method

Finally, we tested the effect of varying stream rates on the effectiveness of LOCUST. While the results of Figures 3 illustrate the effect of the number of samples on accuracy, it is also instructive to examine the effect of varying stream rates on the effectiveness of the resource adaptive method. As an optimistic baseline, we calculated the accuracy when $n_s = 1000$ samples were used in each case. Since the number of samples used in the actual resource adaptive algorithm were much smaller, the accuracy with $n_s = 1000$ provides an *optimistic target accuracy*. The results for the Network Intrusion data set for workloads A and B are illustrated in Figures 5(a) and 5(b). We have picked a number of points in the stream which illustrate the classification accuracy at different levels of input load. In each case, we used the classification accuracy of the set of 5000 input points received just before the point marked on the X-axis. We also note that while these points are received at that point, they may be processed at different times depending upon the target queue delay. The final classification accuracy is only slightly lower than the classification accuracy with the use of the optimistic scheme. The results for the forest cover set are illustrated in Figures 5(c) and 5(d), and are similar.

While the classification accuracy of either scheme (with different target queue waiting times) may be higher, the scheme with the higher target waiting time is more robust. While the number of samples n_s may be higher or lower for either scheme (depending upon the processing rates as illustrated in Figures 4(b) and (e)), the use of larger target waiting times leads to a more uniform value of the number of samples n_s. In most cases, the accuracy is quite close to the optimal accuracy, but there are two cases for workload A in which the accuracy of the scheme with lower target times is modestly lower than the optimistic accuracy. These correspond to the situations in which the input rate is significantly higher than the peak processing rate. As a result, significantly lower values of n_s are picked for short periods of time, which reduces the accuracy. This is not true for the second scheme (with $t_w = 100$) in which larger queue waiting times help to adjust the processing rates, and spread the load out over time.

IV. Conclusions

In this paper, we present a new technique for resource adaptive classification of high dimensional data streams. In order to achieve this goal, we use an online analytical processing framework in which we pre-store certain statistics about the stream in an underlying data structure. This data structure is then used for effective instance specific subspace sampling for classification. This method is especially useful for high dimensional data because of its ability to sample locally important subspaces for classification. The use of an intermediate structure also allows for a resource adaptive approach in the processing rate can be controlled depending upon the input rate of the data stream. For the case of bursty workloads, it is possible to smooth out the processing rate effectively. This makes it possible to leverage on periods of slow input rate in order to obtain higher overall accuracy. Such an approach models many real situations in which the input rate of the stream may vary considerably over time. Our results show that the technique is more accurate than the micro-clustering method even for the non-adaptive case for which the latter is designed.

References

[1] C. C. Aggarwal, J. Han, J. Wang, and P. Yu, "On Demand Classification of Data Streams," in *KDD Conference Proceedings*, 2004.

[2] C. C. Aggarwal, "Data Streams: Models and Algorithms," *Springer*, 2007.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and Issues in Data Stream Systems," in *ACM PODS Conference Proceedings*, 2002.

[4] L. Breiman, "Random Forests," in *Machine Learning Journal*, vol. 45, no. 1, pp. 5–32, 2001.

[5] P. Domingos, and G. Hulten, "Mining High-Speed Data Streams," in *ACM KDD Conference Proceedings*, 2000.

[6] R. Duda, and P. Hart, *Pattern Classification and Scene Analysis*, Wiley, New York, 1973.

[7] J. H. Friedman, *A recursive partitioning decision rule for non-parametric classifiers*, *IEEE Trans. on Computers*, vol. C-26, pp. 404-408, 1977.

[8] G. Hulten, L. Spencer, and P. Domingos, "Mining Time Changing Data Streams," in *ACM KDD Conference Proceedings*, 2001.

[9] F. Farnstrom, J. Lewis, and C. Elkan, "Scalability for Clustering Algorithms Revisited", in *ACM SIGKDD Explorations*, Vol. 2, no. 1, 2000, pp. 51–57.

[10] J. Friedman, R.Kohavi, and Y. Yun, "Lazy Decision Trees," in *AAAI Conference*, 1996.

[11] J. Gama, R. Rocha, and P. Medas, "Accurate Decision Trees for Mining High-Speed Data Streams," in *ACM KDD Conference Proceedings*, 2003.

[12] R. Jin, and G. Agrawal, "Efficient Decision Tree Construction on Streaming Data," in *ACM KDD Conference Proceedings*, 2003.

[13] O. Maron, and A. W. Moore, "The Racing Algorithm: Model Selection for Lazy Learners," *Artificial Intelligence Review*, 1997.

[14] S. K. Murthy, "Automatic construction of decision trees from data: a multidisciplinary survey," *DMKD Journal*, number 2, pp. 345–389, 1998.

[15] H. Wang, W. Fan, P. Yu, and J. Han, "Mining Concept-Drifting Data Streams Using Ensemble Classifiers," in *ACM KDD Conference Proceedings*, 2003.

[16] http://www.cs.uci.edu/˜mlearn