# Novel Class Detection and Feature via a Tiered Ensemble Approach for Stream Mining

Brandon Parker, Ahmad M Mustafa, Latifur Khan
Department of Computer Science
University of Texas at Dallas
Richardson, TX
{brandon.parker, amm106220, lkhan}@utdallas.edu

***Abstract*— Static data mining assumptions with regard to features and labels often fail the streaming context. Features evolve, concepts drift, and novel classes are introduced. Therefore, any classification algorithm that intends to operate on streaming data must have mechanisms to mitigate the obsolescence of classifiers trained early in the stream. This is typically accomplished by either continually updating a monolithic model, or incrementally updating an ensemble. Traditional static data mining algorithms futile in a streaming context (and often in a distributed sensor network) due to their need to iterate over the entire data set locally. Our approach -- named *HSMiner* (Hierarchical Stream Miner) -- takes a hierarchical decomposition approach to the ensemble classifier concept. By breaking the classification problem into tiers, we can better prune the irrelevant features and counter individual classification error through weighted voting and boosting. In addition, the atomic decomposition of feature inputs enables straightforward mapping to distributing the ensemble among resources in the network. The implementation proves to be fast and very memory conservative, and we emulate a distributed environment via signal-linked threads. We examine the theoretical and empirical analysis of our approach, specifically examining trade-offs of three different novel class detection variations, and compare these results to a similar method using benchmark data sets.**

***Keywords* — novel class detection, distributed stream mining, concept drift, feature evolution, hierarchical ensembles.**

## I. INTRODUCTION

Mining streaming data presents several challenges beyond the usual issue confronting machine learning and data mining algorithms in an offline or data warehouse context. Stream mining algorithms cannot assume data is accessible once, and must therefore be designed to incrementally learn and label data as it flows through the system. This paradigm is also familiar to sensor networks, as such networks have typically constrained computational resources in the majority of the nodes, and thus must pass data and derived information through the network to collectively address the network goals.

The paradigm of incremental learning is also in line with the typical operational concept of stream mining and sensor network data analysis, where the users and operators are interested more in the current trending changes and near real-time identification of novel activities or events as opposed to post-analysis mining which tends to look at the holistic data set and glean overarching trends and associations. As such, streaming data presents three main classification challenges that are important to the application of such algorithms:

- Novel classes - new classes can appear mid-stream
- Concept drift - underlying class distributions can fluctuate over time
- Feature evolution - the underlying attributes can change over time, either through the introduction of new features, or the addition of new discrete values in an existing feature set.

One classic example that contains all the above fluctuations in the data is textual streams [1]. New terms can appear in text data (such as Twitter feeds) that are used as attributes to classify and label the text document. A monolithically trained algorithm can miss these dynamic changes in the data set and end up treating new classes as outlier data and new features as irrelevant.

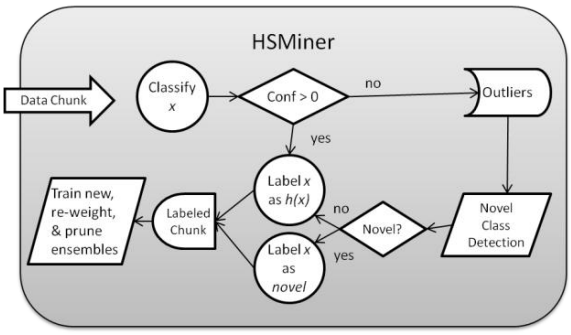


*Figure 1: HSMiner Workflow*

We therefore have designed an approach that builds a hierarchy of ensemble classifiers by breaking down the classification problem so that it can be incrementally updated and each independent section of the ensemble tree learns and updates the weights and inclusion of relevant features. We call the algorithm *HSMiner (Hierarchical Stream Miner).* This decomposition also lends itself quite well to a distributed approach, as each sub-learner operates independently with regard to training and prediction. Each parent node then

1082-3409/12 $26.00 © 2012 IEEE
DOI 10.1109/ICTAI.2012.168

updates a weight assignment to the contributing child nodes to mitigate error. Since the information exchanged between classifier nodes is minimal and the base learners are designed to be lightweight, the distributed learner can be deployed in the context of a sensor network, where each node feeds forward the current conditional predictions, and the upper tier nodes combine the report information to derive a more accurate picture and learn the accuracy of the contributing nodes. The basic workflow is depicted in Figure 1.

The first decomposition assigns a top tier ensemble learner to each class or label of interest. These ensembles are composed of identical classifiers that are trained separately by individual data chunks extracted from the stream. The chunk-based approach for incremental learning is a fairly common practice [2][3][4][5], and works well to process both large data sets and streaming data.

At the bottom of the hierarchy, there are two distinct types of base learners: Naive Bayes for discrete data, and threshold learners for continuous data. While we could map continuous data into a discrete form to use with Naive Bayes exclusively, it requires deeper distribution analysis in order to perform such mapping without introducing undue error. It is easier to treat continuous data separately. The continuous data learner finds a series of boundaries, or thresholds, that linearly separate the binary classes with minimal error. This is a traditional and well document approach for weak learners using AdaBoost [2][6][7]. However, such thresholds require a numeric ordering that is not feasible with string or enumerated data. We therefore also cannot easily map such discrete data to the threshold learner approach, and find that retaining two heterogeneous base learner types is ideal.

Each sub-learner casts a vote in a continuous range from (-1,+1) to indicate whether the data instance in question is (+1) or is not (-1) predicted to belong to the classifier's label. The continuous nature of such a vote also identifies outlier data instances - a benefit of AdaBoosting discussed in [8]. Marking the outliers at the top level of the hierarchy then allows the discovery of novel classes within the set of outliers, somewhat similar to [9].

Unlike other approaches that must either normalize attribute values and attempt to find multi-dimensional classification rules [9], are bound to a single base learner type [5], or must choose exclusively between continuous or discrete approaches per data set [10], our approach is designed to use attributes as they are provided, choosing the appropriate base classifier that requires neither normalization nor parameter tuning. Other approaches may also attempt to keep a unified view of all useful features for classification [11], but through our tiered and separated learners, each classification attempt can be made independently on the features that are present. If a classifier shares no common subset of attributes with a data instance to be classified, then the classifier simply votes neutrally (i.e. with zero confidence).

Using this hierarchical additive weighted voting ensemble method to boost the accuracy of the collection of weak learners, we build an efficient and accurate model. Our solution consists of successive learners updating in continuously evolving data boosted in opposition of error.

Additional discussion on alternative approaches and related work are found in Section 2. In Section 3, we give further detail to our implementation. Section 4 examines a theoretical basis for our approach including performance and error, while empirical results are discussed in Section 5. We conclude in Section 6 with a summary and outline future research extensions.

## II. Approach

Algorithms for mining streaming data must meet certain design criteria, as indicated in [1],[17],[18], and [19], including:

- only has one chance to use a data instance as there is no persistence data storage,
- must limit the memory utilization to a near-constant and computational complexity to linear values to have a stable and usable system,
- should be able to predict a label at any time, yet handle the evolving data characteristics as the stream progresses.

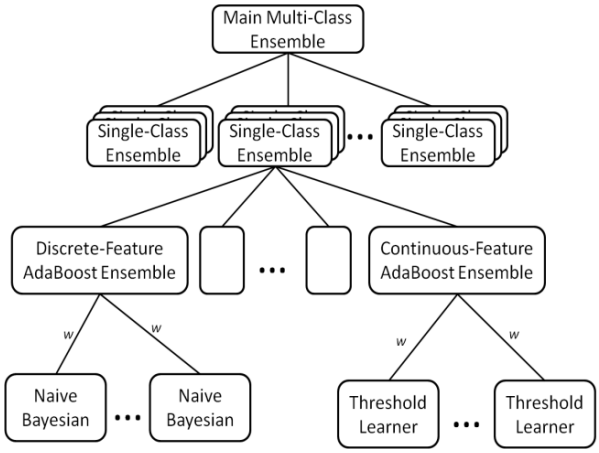


***Figure 2:*** *HSMiner tiered architecture*

Our approach, as previously depicted in Figure 1, creates an ensemble hierarchy (Figure 2) and trains a series of chunk-based single-class ensembles at the top tier in order to address these stream mining requirements.

The approach at the top level of the hierarchy is quite straight forward. In our setup, we collect and queue one chunk of data instances (typically 1000 instances), using the ensemble to label each instance. In the context of a sensor network, the network nodes would be independently receiving

data instances and passing their votes up the tree, which would then be queued as a chunk for the remainder of the chunk iteration.

Once the contributing classification votes are complete, novel class detection occurs by attempting to find cohesive groups of outliers. Finally, each per-class learner is updated through training on the new chunk (including any identified novel classes as a new classifier). This process is depicted in Algorithm 1:

**Algorithm 1: ProcessChunk(X)**

**Input:** Data Set X (representing one data chunk)
**Vars:** FullEns (top-level ensemble)
Outliers (set of known outliers
1: currentErr := testFullEnsemble(X)
2: NC := extractNovelClasses(X,Outliers)
3: FullEns := FullEns U TrainEnsembleHierarchy(X)

*A. Building Classifiers*

As new labels or classes are discovered, a new per-class learner is established. On the training phase of each chunk iteration, each per-class learner independently trains a new version of the learner for the new data chunk. The per-class learners then each rank the accuracy of their ensembles and only retain the L best members (where L is configurable, but typically between 3 and 6). Algorithm 2 depicts that basic training process in a consolidated summary form.

**Algorithm 2: TrainEnsembleHierarchy(X)**

**Input:** Data Set X (representing one data chunk)
**Output:** Top level MultiClass Ensemble
1: foreach L ϵ Labels found in X
2: S := ø
3: foreach F ϵ Features found in X pertaining to L
4: A := ø
5: if (F isa {string | enum})
6: then H := *new NaiveBayesAdaBoost(X)*
7: else H := *new ThresholdAdaBoost(X)*
8: if (H.trainingError < 0.5) then A := A U H
9: endfor
10: S := S U A
11: endfor

When a new chunk appears and the instances therein are labeled (per the classification phase), the ensemble is updated by sharing the data chunk to the known multi-chunk/single-label learners. Any new class discovered (and given a temporary label) is added to the top-level ensemble.

Each multi-chunk/single-label learner trains a new single-label member on the new chunk, and then re-ranks the per-chunk trained members, eliminating the one with the worst accuracy.

Each single-label learner creates a separate single-feature learner for every pertinent feature found in the data chunk. Any feature that has no positive examples of the label in question is skipped. Likewise, any single-label/single-feature learner that has an error rate greater than half is eliminated. This quickly prunes the set of features that contribute to the classification of a label. The individual single-label/single-feature learners are chosen based on the feature type: Naive Bayes for discrete feature types, and a boosted threshold ensemble for continuous types. Both of these learners are efficient to train.

Note that we dynamically choose the nature of the base learner with regard to the feature type (Algorithm 2 line 5). Discrete data (enumerations, strings, etc) use a simple Bayesian classifier which will determine Maximum a Posteriori (MAP) probability between the probability: $P(c|x)$ versus $P(\sim c|x)$. For continuous (or unbounded integer) valued features, we use a simple threshold approach. Since a threshold learner is a simple linear separator and we cannot assume linear separability in a given feature space, we build a single class AdaBoost ensemble of threshold learners.

*B. Classification*

The classification phase is straightforward and computationally fast. When an instance is labeled by the system, it is passed to down the hierarchy and each base learner casts a weighted vote between (-1.0, 1.0) indicating their prediction as to whether the instance is (+1.0), is not (-1.0), or undecidedly (0.0) a member of their class. These votes propagate up the hierarchy and are aggregated into a final voting tally:

$$h_c(x) = \sum_{i \in MC} \sum_{f \in Feature} h_{i,c,f}(x_f)$$

The label with the highest vote is said to be the predicted class of the data instance:

$$h(x) = \underset{c \in Class}{\operatorname{argmax}}\, h_c(x)$$

However, since the votes also represent a form of confidence of the ensemble hierarchy, if the highest vote is less than some threshold (i.e. 0.0), then the instance is marked as a potential outlier since no single label classifier was confident the instance belonged to their class. These outliers are then examined by the novel class detection algorithm, and may be re-labeled as *novel*.

Without loss of generality, imagine a data chunk contains data instances with four features $\{x_1,x_2,x_3\}$ and a label class y ϵ {A,B,C}. Assume also, for the sake of simplicity, that the *HSMiner* algorithm has already discovered the first two classes A and B. As such, the hierarchical ensemble contains two top branches for classes A and B, and each of those branches contain four sub-branches for each of features, which

we can label $x_{1A}$, $x_{2A}$, $x_{3A,}$ $x_{1B}$, $x_{2B}$, and $x_{3B}$ for this discussion. When a new data instance is passed in for prediction, each base learner cast a vote between (-1.0, +1.0). For instances, we could find that:

$hx_{1A}(X) = 0.2$ $\quad$ $hx_{1B}(X) = 0.3$
$hx_{2A}(X) = 0.4$ $\quad$ $hx_{2B}(X) = 0.6$
$hx_{3A}(X) = 0.1$ $\quad$ $hx_{3B}(X) = -0.7$

With all weights equal, the top level of the ensemble would select class A from the comparison:

$$(0.2 + 0.4 + 0.1) > (0.3 + 0.6 - 0.7)$$

However, when the learned accuracy rates are applied, the results may change. For instance, if the following weights were also learned:

$Wx_{1A}(X) = 0.1$ $\quad$ $Wx_{1B}(X) = 0.4$
$Wx_{2A}(X) = 0.4$ $\quad$ $Wx_{2B}(X) = 0.9$
$Wx_{3A}(X) = 0.1$ $\quad$ $Wx_{3B}(X) = 0.1$

The ensemble would then choose class B after comparing:

$$(0.2*.01+0.4*0.4+0.1*0.1) > (0.3*0.4 + 0.6*0.9 - 0.7*.01)$$

There are instances, however, which result in comparisons of extremely small or negative results on both sides of the comparisons. If, for instance, the final confidence vote for both classes A and B were negative, the instance could be an outlier, or it could be a member of a yet undiscovered third class C.

### C. Novel Class Detection

We examine two novel class detection approaches. The first approach is rather naive, but used as a baseline for comparison. It simply assumes that any data instance with a predicted label confidence below a certain threshold is novel without regard to outlier cohesion or other factors.

The second and more interesting approach is a sieve-like approach. A sieve, conceptually, iteratively refines a noisy mix of objects separating the undesirable objects from the objects of value. For this algorithm, we create a pseudo data chunk where each instance in the real data chunk is given a label of either "novel" if it had been marked with an outlier flag, or labeled "known" otherwise. The algorithm then trains a new single-label classifier using this new "novel" label, as show in Figure 3.

The pseudo chunk is then tested against this new trained novel-class decider, and any instance that has a higher confidence of being labeled *novel* by this new classifier than by the known labels (i.e. the data instances original predicted label) has the *outlier* mark removed and is re-marked as *novel*. The algorithm then repeats this process until any of the following conditions are met:

- no data instances are marked *outlier*
- the training error of the *novel* learner is greater than half
- no data instances are re-marked as *novel*

Once this novel class detection iterative process is complete, all instances that still are marked as *outliers* retain their original predicted label (which at least has a decent chance of being right), and all data instances marked as *novel* are grouped as a new class and given a temporary label which the user can change in due time.

## III. Theoretical Foundation

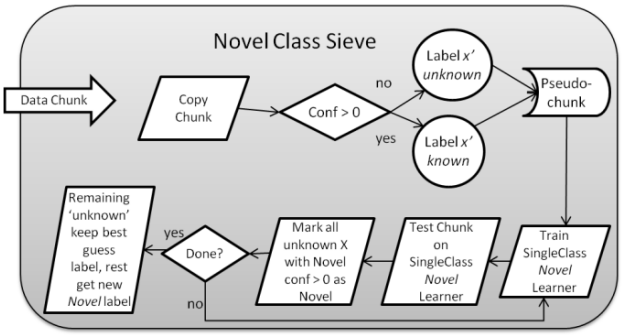


*Figure 3: Sieve Novel Class process*

Core factors to decompose the classification problem into a hierarchical approach are computational performance and distributability (for performance or across a sensor network). However, such extreme separation can severely reduce accuracy if left unchecked since many features can be inter-dependant, correlated, or irrelevant to a class.

### A. Computational Performance

Due to the decomposition nature of our approach, training and classification both scale linearly with the data, and both phases are highly parallelizable. Since each base learner is independently trained per feature and per class, the computational complexity of both the Naive Bayesian and threshold learners are *O(n)* where *n* represents the number of data instances in a chunk. Iterations of AdaBoost are configurable, but constant in our implementation. Therefore the overall computational performance can be characterized as *O(nfc)* where *f* is the number of features and *c* is the number of known labels or classes in the current data chunk. Proportionally it is often the case that:

$$n >> f >> c$$

Since *n* is the dominant factor in this case, and since we can divide the computation to parallel processes across the labels (c) - and even across the per-feature learners (f) if we have highly parallel processing resources like a GPU, the overall complexity estimation remains at O(n). This bodes well for mining stream data, since it means the algorithm is not likely to fall behind in processing the data.

Such observation is echoed in [8] which compares boosting to a greedy approach for linear programming, in comparison to the much more complex and demanding quadratic programming required for Support Vector Machines (SVM).

*B. Error Analysis*

Reducing computational complexity with heuristics is only valuable if we can contain the added error induced by the approximation or assumptions used by the heuristic. In our approach, the single-class, single-feature classifiers at the bottom of the hierarchy are the core of the classification capability of the composite ensemble. Because we prune out any base learner and ensemble that is not better than a random classifier (i.e. better than 50% accurate for binary classifiers), we start with the premise that each base weak contributor is at least better than random on the training data. From there we can leverage the error bounds guarantees from AdaBoost for the single-feature, single-class ensemble. As discussed in [8] and [15], AdaBoosting minimizes the error by iteratively adding refined learners, continually emphasizing hard to classify instances. The resulting ensemble is then able to discriminate classes across nonlinear separation boundaries. In [15], the generalized error upper bound for our feature-based AdaBoost assuming the *margin* for a given data instance *(x,y)* as *yf(x)* is:

$$E_f = P(yf(x) \le \theta) + O\left(\frac{\sqrt{\left(\frac{d\log^2(m/d)}{\theta^2} + \log(1/\delta)\right)}}{\sqrt{m}}\right)$$

for all $\theta > 0$ and probability $(1 - \delta)$. While this shows a definite benefit when a feature is a good indicator of a label, we cannot assume that the data distribution in the training set will be consistent with later data sets of subsequent data chunks. Therefore we must examine how this error propagates up the hierarchy and how we can mitigate larger errors for less accurate feature-based classifiers. Classical error propagation would indicate that the error at the single-class ensemble level is characterized by:

$$E_c = \sqrt{\sum_{f \epsilon\ Features} {E_f}^2}$$

As we build our single-class ensemble, however, we assign a weight to each single-feature ensemble based on the function, recognizable from traditional AdaBoost [2][7]:

$$w_f = \frac{1}{2}\log\left(\frac{1 - error}{error}\right)$$

The overall error of AdaBoost, as noted in [2], can also be viewed in terms of the exponential loss function, so we can exploit the weight function to counter-act the error yielding the per-class error function:

$$E_c = \sqrt{\sum_{f \epsilon\ Features} \left(w_f E_f\right)^2}$$

Ensemble error is reduced further by minimizing the impact of error-prone features and maximizing the contribution of the more robust features per class.

At the top tier, a maximum weighted vote of the underlying single-class ensembles determines the predicted label(s). As such, the final propagated error is simply:

$$E_{mc} = \sqrt{\sum_{c \epsilon\ Class} {E_c}^2}$$

A sufficiently dense set of similar outliers at the top of the hierarchy may be an indication of a novel class. If we can properly identify these novel classes in the stream, we further reduce the error resulting from the divergence between the training set and current test (chunk) data set. As discussed in [15], we can interpret the raw continuous output of an AdaBoost ensemble as a confidence. In other words, given hypothesis h(x), the binary classification is obtained through *sign(h(x))* but the confidence can be viewed as the magnitude of *|h(x)|*. We take this approach and forward the continuous-valued hypothesis up to the main ensemble. The final hypothesized label is therefore the label with the highest aggregate (and weighted) confidence. If, however, the highest confidence is below some threshold (for instance, if all confidences are near zero, or all hypothesizes are negative), then the instance is considered an outlier - a property of AdaBoost recognized in [8]. Identifying these outliers further reduces our error bounds by the probability that the data instance is a member of an identifiable novel class:

$$P(\overline{Err}|Outlier) = P(NovelClass)P(x \epsilon Cluster_{NovelClass})$$

One instance of a novel class has a higher probability of detection if a sufficient quantity of similar instances exists in the data chunk. Therefore, in the presence of novel classes, either the total error is small due to the lack of novel class saturation, or is minimized due to the removal of such outliers from the error since the instances are able to be grouped into a new class (and given official/meaningful labels later by the user). Thus the final aggregate is approximated as:

$$E_T = \sqrt{\sum_{mc \epsilon\ MultiClass\ Ens} (w_{mc}E_{mc})^2} - P(\overline{Err}|Outlier)$$

Existing data mining algorithms tend to limit assignment of weight to given feature contribution. Our approach dynamically adapts to the changes in the stream re-weighting not only the feature contributions as new learners are created and removed from the ensemble, but by also re-weighting the intermediate ensemble weights the algorithm further mitigates the drift in error through the duration of the data stream.

## IV. Empirical Results

In order to test our approach, we implemented the described method and tested against several established data sets. **Table 1** depicts the characteristics of the two data sets.

**Table 1: Test data set characteristics**

| | Instances | Classes | Continuous Features | Discrete Features |
|---|---|---|---|---|
| KDD | 490,000 | 21 | 39 | 3 |
| ForestCover | 581,000 | 7 | 10 | 44 |
| NASA | 125,955 | 21 | 0 | 1,000 |
| Twitter | 229,513 | 100 | 0 | 19,397 |

For the first data set, we use the *ten percent, corrected data* from the KDDCup '99. It depicts meta-data of network traffic with a labels for normal traffic and 21 different cyber attack methods. We treat three of the 42 attributes as discrete, since they are represented by a string set (protocol_type, service, flag), and the rest as numeric/continuous since they represent times, rates, counters, and byte sizes.

The second data set, *Forest Cover*, was obtained from the authors of DXMiner, who obtained it from the UCI repository as explained in [12]. It contains 10 numeric (treated as continuous) attributes, and 44 discrete (absence or presence Boolean indicators) attributes. We use all attributes as they arrive in the stream by parsing the data file without filtering and without normalization.

The third data set, *NASA*, was obtained by extracting terms from the same ASRS source data found in [12]. However, unlike [12], we extracted 21 classes and 1000 features to obtain a data stream representative of a truly dynamic and evolving feature space. Similarly, we tested a Twitter feed data set that consists of collected Tweets where the hash-tag denoted word indicates the topic, or class label. Like the NASA data set, the feature are the existence of words filtered to remove articles and other stop words.

*HSMiner* is mostly parameter free with regard to the learners, although there are a few parameters that can be tuned for computational performance that can also affect the ensemble accuracy. As discussed in [13] and [2], chunk size and ensemble size are two such parameters. For our experiments, we used a chunk size of 500 and 1000 instances and our upper level ensemble retained the best 3 of the per-chunk trained ensembles.

Our solution was implemented in C++ using multi-threading. Table 2 shows the computational performance characteristics with regard to the data sets. The experiments were run on a i7-2600k machine running Windows Vista. In addition, the process image quickly approached but never exceeded 50MB of RAM, demonstrating the memory efficiency of our design.

**Table 2: Execution time in milliseconds per chunk**

| | HSMiner | DXMiner |
|---|---|---|
| KDD | **215** | 819 |
| ForestCover | **220** | 559 |
| NASA | **35,527** | (program crashed) |
| Twitter | **175** | 2,497 |

Table 3 depicts the accuracy metrics of the algorithm by showing the average error incurred by the algorithm on the data sets. For each chunk, we find the error of using the current chunk's instance to test the current ensemble hierarchy. This test occurs before the chunk's instance are used to modify any component of the ensemble. The percent error across these chunks are then averaged to produce the results indicated here.

**Table 3: Average error results**

| | HSMiner | DXMiner |
|---|---|---|
| KDD | **2.8** | 13.1 |
| ForestCover | 7.9 | **5.2** |
| NASA | **59.1** | (program crashed) |
| Twitter | **41.8** | 68.9 |

We ran the same data set with our implementation and on the DXMiner implementation to obtain the above results. Figure 3 shows the average error per chunk obtained by the *HSMiner* method for the KDD data set. Note that we did not reduce the feature space as was done in [12] since our method can use the feature in their native format (continuous or discrete without normalization). DXMiner could only handle a reduced feature set discarding the discrete data, and had to normalize all features to ranges between (0.0,1.0) while *HSMiner* required no such pre-processing. In addition, the ForestCover data set does not truly have a dynamic feature space, and DXMiner retains a union of all features for its learners as it progresses. Therefore the lower error of DXMiner on the Forest Cover data set is not surprising due to the fixed feature set in the data. Figure 4 shows the average error per chunk as *HSMiner* method processed the ForestCover data set. As an additional comparison point, the Hoeffding Tree

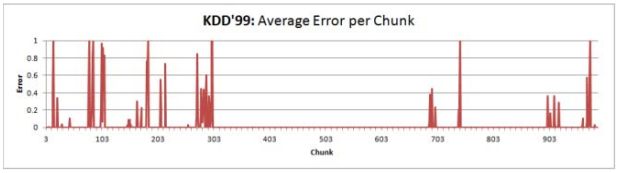


*Figure 3 - HSMiner average error per chunk on KDD data*

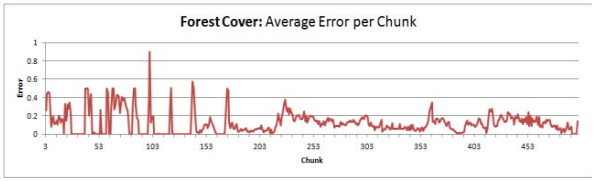


*Figure 4 - HSMiner average error per chunk on ForestCover data*

ensemble result from [16] reported a lowest error of 8.54% on the Forest Cover data set, but at an exceptionally long runtime of 241,144.76 seconds.

The NASA and Twitter data sets are much better representation of a truly evolving and dynamic data stream. They were also much more problematic as the NASA set caused DXMiner to crash. While the overall accuracy of *HSMiner* was only roughly 41% on NASA, considering the noise in the data and the random-guess probability of 1 out of 21 classes (roughly 4.7%), the results are still encouraging. *HSMiner* performed even better on the highly dynamic Twitter data set, as seen in Table 3. This data set reveals 100 total labels and is *much* more dynamic in the feature space than any of the other data sets. Even so, *HSMiner* obtained an accuracy of 58.2% - significantly better than DXMiner's 31.1% accuracy and over ten times faster. The NASA stream shows a trending decrease in error, likely because of the fewer classes and recurrent features, whereas the Twitter data set shows spikes of error as the ensemble learns new trends in the stream.

These results show that our approach is indeed competitive in both accuracy and computational speed. The computational speed advantage of *HSMiner* over DXMiner is due in part to the targeted class and feature learners in the hierarchy. *HSMiner* also does not have to pre-process the data in any form - a capability absent from other methods tested. With future work, we expect to be able to refine the accuracy and efficiency even further

## V. Related Work

Data mining streaming has been an area of research for several decades. With the recent explosion in interconnectivity of data sources and constant data accumulation from the expanding use and proliferation of Internet connected devices, it is no surprise that gleaning useful information from data streams has become a research focus. Since our approach is hierarchical, it has several similarities to various sub-areas of stream mining research. Very few, however, adequately address as many of the stream mining issues as our approach.

The closest work to our solution is [12], where the authors address all three issues, but use simple ensemble voting, handle only continuous-valued features, and retain a homogenized conglomeration of all features in all ensemble model evaluations. Likewise, Masud et al. in [9] [11] [13] (which are precursors to [12]) outlines an approach to use an ensemble of hyper-spheres to capture the decision boundary for classes as the stream is processed. These only use continuous-valued features, and normalize these features so that they all range from (0.0, 1.0) in the data set as a pre-processing task. This approach, although shown to be accurate on the tested data, has several drawbacks. First, true streaming data cannot be pre-processed in a manner to normalize all features a priori. Using hyper-spheres as a base learner limits the feature set to only continuous data and applies a constant cross-feature radius.

In contrast, our hierarchical concept begins down at the feature level. The individual feature based learners have analogous feature-specific centroids and decision radii, allowing the use of stream features in their native formal without filtering or normalization. Our approach thus provides a more robust, more accurate, and more interpretable method to mitigate feature evolution and concept drift in a data stream without ignoring or normalizing attributes, while also retaining the capability to handle novel classes.

Masud et al. in [12] directly address the issue of novel class detection. In essence, they take data instances identified as outliers by the classification ensemble and run k-Means to find a unified cohesion and separation metric for each outlier with regard to the outlier's nearness to existing classes versus the outlier set itself. If a sufficient set of outliers have a common cohesion to each other and a sufficient separation from existing classes, a novel class has been discovered.

Katakis et al. in [1] used Naive Bayes to mitigate dynamic features incrementally for textual data streams. They run statistics on the feature space to prune the features to those that are best suited for the current classification problem. Naive Bayes implicitly treats each feature separately, which can be mapped to our approach of creating per-feature classifiers and pruning those that are too error prone. Katakis et al. compare their approach with using a collection of k-NN and SVM, both of which have severe computational time penalties and end up with more error than their approach.

The AdaBoost algorithm is an ensemble method that derives weights to the ensemble members in order to produce an additive prediction model. For the sake of space, we will not re-print the algorithm here as it is well documented and discussed in [2][6][7] among others. Freund and Shapire, in [6], present both a binary classifier called AdaBoost and the often cited multi-class extension, AdaBoost.M1. Using Freund and Shapire's boosting method [6] for streaming data is not without precedence. In [10], Kolter and Maloof explore using an online additive weighted majority algorithm based on AdaBoost to mitigate concept drift in a stream. However, they only address concept drift, and collect a large number of heavy expert classifiers within their ensembles. The approach of using heavy base classifiers and re-evaluating the entire ensemble at each data instance, as shown in [10], does not scale well and only leads to moderate accuracy. In addition, the method in [10] must choose a priori whether to run in discrete or continuous mode. We leverage the AdaBoosting algorithm to create more robust non-linear separators for our per-feature classifiers. Likewise, the error approximation as outlined in [2] for AdaBoost helps to minimize the error propagation up through our hierarchy of ensembles. Like in [4], we add new learners and assign a contribution weight to them based on their error, but instead of pruning the oldest, we prune the weakest and use an error function that better matches the estimated error from the underlying single-feature learner.

Note that AdaBoost derives two sets of weights - one set for the training instances, and the other for the classifiers. Within the algorithm, the instance weights adaptively drive the ensemble to focus on the harder to classify data points, balanced by the weak classifier weights that give more sway to the stronger classifiers. In fact, the AdaBoost algorithm's main

loop is often run for multiple iterations, ideally until the error converges to a stable value [14]. If the weak classifiers represent individual attributes in the feature space, the sum of a particular attribute-decider's a weight corresponds to the decisive contribution that feature has in the data set. [8] and [15] discuss the performance and accuracy contributions of AdaBoosting in this context. Likewise, [8] also points out that training instances with high weights indicate noise or potential outliers. In fact, several attempts have been made to improve a boosted ensemble's accuracy by eliminating such outliers and re-training the ensemble [10]. However, instead of eliminating the outliers all together, in our streaming data situation, we can examine the set of high-weighted instances along with the low weighted features and discover clusters of yet undiscovered label groups - novel classes.

We did not find any sources that use a multi-tiered approach adding weights to the tiers based on error approximations, nor sources that grouped heterogeneous base learners in the same way. However, [16] did use Hoeffding trees to train perceptrons with one perceptron per class, trained with stochastic gradient decent. The perceptrons were then members in a final ensemble. This approach yielded moderate accuracy results, but at a severe computational time penalty. However, one can map the concept of base learners feeding perceptrons, which subsequently feed a higher ensemble as a complex perceptron network. If we assign weights to the additive contributors based on exponential loss error estimates instead of gradient decent, we have a forward feeding network of boosted ensembles, which maps very nicely to our approach. In fact, the hierarchical nature of our ensemble lends itself easily for distribution among network notes, even allowing the per-feature learners to be collocated with the feature sensors in a distributed sensor network, similar to how [20] and [21] distributes the local data caches in the network for optimal queries. In the case of HSMiner, the per-class per-feature learner would instead be localized.

## VI. Conclusion

By combining variations of well known machine learning algorithms in a novel hybrid construction, our approach makes effective use of per-feature base classifiers combined in a hierarchy of weighted ensembles, and concisely and efficiently addresses the issues faced in mining streaming data. Our approach is scalable, as it does not need to retain old instance data, nor does it indefinitely retain all models, nor all features spaces within the sub-models. The iterative approach to modifying the ensemble hierarchy efficiently keeps the overall model relevant to the evolutionary and novel changes within the data stream.

We plan to continue development on HSMiner, with improvements to the Sieve approach, further mitigating error by dynamically maintaining inter-tier weights as the stream progresses, and testing alternative novel class detection approaches. In addition, further experimentation on distributing the learners across a true sensor network instead of emulating via loosely connected threads may help identify potential areas for improvement.

## VII. Acknowledgment

The authors would like to thank Dr. Robert Herklotz for his support. This work is supported in part by the Air Force Office of Scientific Research, under grant FA9550-08-1-0260.

## References

[1] Katakis, I., Tsoumakas, G., Vlahavas, I. Dynamic Feature Space and Incremental Feature Selection for Classification of Textual Data Streams. ECML/PKDD International Workshop on Knowledge Discovery from Data Streams. (2006)

[2] Hastie, T., Tibshirani, R., Friendman, J.: The Elements of Statistical Learning: Data Mining, Inference, and Prediction 2nd Ed. Springer. (2009)

[3] Wang,H., Wei, F., Yu, P.S., Jiawei,H. Mining Concept-Drifting Data Streams Using Ensemble Classifiers. KDD (2003)

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

[5] Oza, N., Russell, S. Online Bagging and Boosting. Artificial Intelligence and Statistics. (2001)

[6] Freund, Y., Shapire, R.E.: A Decision-theoretic Generalization of On-line Learning and an Application to Boosting. Journal of Computer Science and System Sciences, 55(1):pp. 119-139. (1997)

[7] Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. 2nd Ed. Pearson Education, Inc. (2003).

[8] Freund, Y., Schapire, R.E., A Short Introduction to Booston. Journal of Japanese Society for Artificial Intelligence. (1999)

[9] Masud, M.M., Gao, J., Khan, K., Han, J., Thuraisingham, B.M.: Integrating Novel Class Detection with Classification for Concept-Drifting Data Streams. In ECML PKDD '09., volume II, pp. 79-94. (2009)

[10] Kolter, J., Maloof, M.: Using Additive Expert Ensembles to Cope with Concept Drift. ICML, pp. 449-456, Bonn, Germany (2005)

[11] Masud, M.M., Chen, Q., Gao, J., Khan, L., Han, J., Thuraisingham, B.M.: Classification and Novel Class Detection of Data Streams in a Dynamic Feature Space. ECML/PKDD (2010)

[12] Masud, M.M., Chen, Q., Gao, J., Khan, L., Han, J., Thuraisingham, B. Classification and Novel Class Detection of Data Streams in a Dynamic Feature Space. PKDD (2010)

[13] Masud, MM., Gao, J., Khan, L., Han, J., Thuraisingham, B.M.: A Practical Approach to Classify Evolving Data Streams: Training with Limited Amount of Labeled Data. ICDM '08. pp. 929-934. (2008)

[14] Jaksch, T. Boosting and Noisy Data - Outlier Detection and Removal. Masters Thesis. (2005)

[15] Shapire, R.E., Singer,Y. Improved Boosting Algorithms Using Confidence-rated Predictions. Machine Learning. pp297-336. (1999)

[16] Bifet, A., Frank, E., Holmes, G., Pfahringer, B. Accurate Ensembles for Data Streams: Combining Restricted Hoeffding Trees using Stacking. ACML (2010)

[17] Street, N., Kim, Y. A Streaming Ensemble Algorithm (SEA) for Large-Scale Classification. KDD'01. (2001)

[18] Albert Bifet, Geoffrey Holmes, Bernhard Pfahringer, Richard Kirkby, and Ricard Gavalda.New ensemble methods for evolving data streams. In KDD, pages 139-148. (2009)

[19] Santos Teixeira,PH., Milidiu, R.L. Data Stream Anomaly Detection through Principal Subspace Tracking. SAC'10. (2010.)

[20] A. Cuzzocrea, F. Furfaro, S. Greco, E. Masciari, G.M. Mazzeo, D. Saccà: A Distributed System for Answering Range Queries on Sensor Network Data. PerCom Workshops 2005, pp. 369-373. (2005)

[21] A. Cuzzocrea, F. Furfaro, D. Saccà: Enabling OLAP in mobile environments via intelligent data cube compression techniques. Journal of Intelligent Information Systems 33(2), pp. 95-143. (2009)