Stream Mining Algorithms for Sensor Data Classification

Yue Dong
Department of Mathematics and Statistics
University of Ottawa
Ottawa, Ontario
ydong029@uottawa.ca

Philippe Paradis
School of Mathematics and Statistics
Carleton University
Ottawa, Ontario
philippe.paradis@gmail.com

Nathalie Japkowicz
School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Ontario
nat@site.uottawa.ca

***Abstract*—Learning from data streams is a research area of increasing importance [1]. Nowadays, several stream mining algorithms have been developed. Most of them learn decision tree models that continuously evolve over time, such as Very Fast Decision Trees (VFDT). However, it is not natural to use decision trees when learning has to be continuous — which it frequently does due to concept drift — because it is costly to adjust the tree's shape with new data. We thus propose two efficient incremental learning algorithms based on neural networks, Sliding Mini-batch Neural Networks and Sliding Chained Neural Networks, which avoid this problem. The results of our experiment on the Dodgers Loop Sensor Dataset indicate that the performances of our algorithms converge to that of popular batch learners.**

I. Introduction

Learning from data streams is a research area of increasing importance [1]. Many datasets are no longer static, instead they continuously get updated, sometimes with millions or even billions of new records per day. In most cases, this data arrives in streams, and if it is not processed immediately or stored, it will be lost forever. Moreover, the data arrives so rapidly that it is usually not feasible to store it all. To extract useful knowledge from this tremendously valuable data, stream mining became popular.

Data stream mining is the process of extracting knowledge from an ordered sequence of data instances. One goal of data stream mining is to predict the class of new data instances based on previous instances in the data stream (classification). In many applications, data analysis can only be performed in one pass over the data due to the sheer size of the dataset. Because of this, traditional batch learning algorithms are often unsuitable for data stream mining. We thus need algorithms which can summarize information about prior data instances without keeping all of them in memory.

Several attempts have been made to modify the existing batch learners for data stream learning. According to Gama et al. [1], most of them use decision trees "that continuously evolve over time, taking into account that the environment is non-stationary and computational resources are limited." Examples of these algorithms are Very Fast Decision Trees (VFDT) [2] and Concept-adapting Very Fast Decision Trees (CVFDT) [3].

However, it is not natural to use decision trees for data stream mining, because they are very inefficient at dealing with concept drift – an important issue in data stream mining. Decision trees work top-down in a way involving choosing the best attributes to split data at each level. To grow one level, decision tree algorithms need to scan the whole dataset once. Then, once the attributes are chosen, it is costly to change the shape of the trees or to change the nodes which have already been chosen. In the worst case scenario, if the new training data requires the root attribute to be changed, then all the nodes have to be chosen again. Since the characteristics of streaming data require the algorithm to constantly adjust to the new data, decision trees are not the best for data streams mining.

On the other hand, it is easy to see that neural networks (NN) are suitable for dealing with concept drift. In fact, they naturally handle well the task of incremental learning. A typical feed-forward neural network has one input layer, one output layer and several hidden layers. It learns by adjusting the weights between neurons in iterations. When continuous, high-volume, open-ended data streams come, it is necessary to pass the data in smaller groups (mini-batch learning) or one at a time (online learning) before updating the weights. In this manner, neural networks can learn by seeing each example only once and therefore do not require examples to be stored.

In this paper, we build two models based on mini-batches of neural networks and sliding windows.

The first model, ***S**liding **M**ini-batch **N**eural **N**etworks* (SMNN), works by training a neural network on mini-batches of data instances selected from overlapping sliding windows over the streaming data. The second model, ***S**liding **C**hained **N**eural **N**etwork* (SCNN), works by constructing a finite ensemble of neural networks trained on sliding windows, where the final weights of a neural network are passed as the initialization weights of the subsequent neural network.

SMNN and SCNN satisfy the desirable properties of learning systems for efficient data stream mining proposed by G. Hulten and P. Domingos [4]: they require small constant time per data example; they use fixed amount of main memory, irrespective of the total number of examples; they build models using a single scan over the training data; they generate anytime models independent from the order of the examples; and they have the ability to deal with concept drift.

The rest of this paper is organized as follows: in Section II, we discuss the related work. In Section III, we propose

our methods for mining streaming data. In Section IV, we discuss the data pre-processing and the experimental design. In Section V, we present the results and compare them with the popular batch learners and stream mining algorithms. In Section VI, we conclude with remarks, discussion of the limitations of the experiment and suggestions for further work.

II. Related work

In this section, we first review recent studies on decision trees based stream mining algorithms. Then we discuss recent studies on stream mining algorithms based on incremental neural networks and ensemble methods.

Stream data mining has become prevalent in the last two decades and many different algorithms for stream data mining have already been proposed in the literature. Based on the paper "Introduction to stream: An Extensible Framework for Data Stream Clustering Research with R" [5], several classification methods suitable for data streams have been developed. Examples are Very Fast Decision Trees (VFDT) [2] based on Hoeffding trees, Online Information Network (OLIN) [6], On-demand Classification [7] based on micro-clusters found with the data-stream, and the clustering algorithm CluStream [8]. There are many other popular algorithms, such as UFFT (ultra fast forest trees) [9] and CBDT (concept based decision tree) [10].

As we can see, most of these algorithms are based on decision trees. However, it is not natural to use decision trees when learning has to be continuous — which it frequently does due to concept drift. When concept drift presents, the decision tree structure is unstable in the setting of stream mining and it is costly to adjust the trees' shape with new data. We study VFDT in this paper as an illustration.

The traditional decision tree classifier uses a divide-and-conquer approach. Typically, it needs to scan the whole dataset once to grow one level (excluding elements that have already reached a leaf at a lower level). After the scan, it finds the best attribute for splitting the data at each node by information gain or other criteria. To develop one decision tree, multiple scans of the whole dataset are required as the following figure indicates:

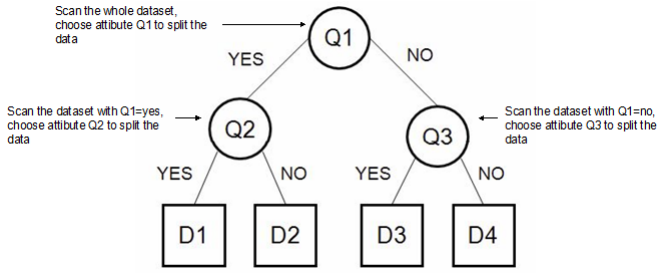


Fig. 1: An example of decision tree

As we can see, decision trees need to scan the whole dataset to grow one level (if no leaves have been formed yet). However, we cannot keep the entire dataset in memory for typical stream mining datasets. Therefore, traditional decision tree algorithms don't suit the job of stream mining. An algorithm that can learn *incrementally* is required.

The VFDT overcomes this limitation by building each node of the decision tree based on a small amount of data instances. According to Domingos [2], the VFDT works as follows: "given a stream of examples, the first ones will be used to choose the root test; once the root attribute is chosen, the succeeding examples will be passed down to the corresponding leaves and used to choose the appropriate attributes there, and so on recursively." An illustration of how VFDT is built is shown as Figure 2.

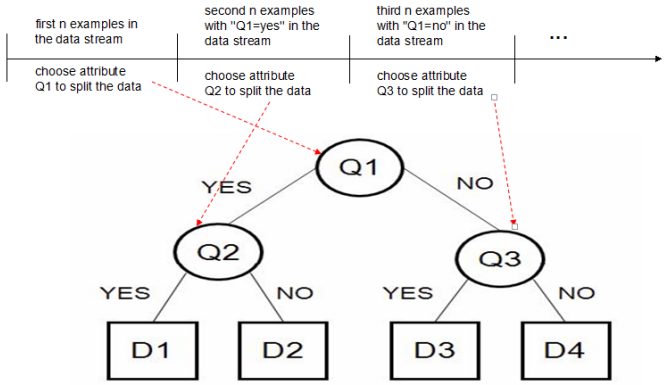


Fig. 2: An illustration of VFDT

To decide how many examples (the size of n in the above Figure) are necessary at each node, the Hoeffding bound is used. With the Hoeffding bound, the VFDT can guarantee that the performance of Hoeffding trees converges to that of a batch learner decision tree [2].

The Hoeffding bound works as follows: consider r as a real valued random variable, n as the number of independent observations of r, and R as the range of r, then the mean of r is at least $r_{avg} - \epsilon$, with probability $1 - \delta$:

$$P(\bar{r} \geq r_{avg} - \epsilon) = 1 - \delta$$

where

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}$$

Although the performance of the VFDT converges to that of a batch learning tree, this data structure is not "stable" and only works when there is no concept drift. Subtle changes of the data distribution can cause the Hoeffding trees to fail. For example, in Figure 2, suppose that Q1 was chosen as the root attribute after the first n examples and trees were grown until reaching level 1000. Next, assuming a sudden change in the data distribution (due to concept drift), it would be possible that choosing a new root attribute, such as Q500 instead of Q1, would give a much better prediction. In this case, the tree would have to be rebuilt completely, but this is generally impossible since we can't retrieve the earlier data passed in the stream.

To mitigate the issue of concept drift, the same authors introduced Concept-adapting Very Fast Decision Trees

(CVFDT), which adapt better to concept drift by monitoring changes of information gain for attributes and generating alternate subtrees when needed. However, it still can't solve the issue completely because the algorithm only monitors the leaves.

Due to this disadvantage of decision tree based stream mining algorithms, other researchers tried to stay away from having to re-grow decision trees at all by using incremental neural networks or ensemble methods.

Many researchers have proposed algorithms based on incremental neural networks, such as the papers [11], [12], [13], [14] which use incremental neural networks to train on the data points one by one and update the weights every time after scanning a data point. The shortcomings of this incremental neural network is that the results are highly dependent on the randomized initial weights since data is only passed once. As we know, in batch learning, we pass the data in many epochs to avoid the effect of randomized initial weights. This is impossible in incremental neural networks since the data only gets passed once and previous data is discarded.

Other researchers proposed methods based on ensemble classifiers. For example, Wang et al.[15] proposed a general framework for mining data streams with concept drift using weighted ensemble classifiers. The authors train an ensemble of classification models from sequential chunks of the data stream. However, building a classifier independently on each chunk can be inefficient since the data from all chunks are usually dependent.

Overall, most stream mining algorithms involve some stream summarization. There are different ways to summarize data streams. Some methods will only keep a portion of the data in streams by using sampling. Such approaches including reservoir sampling [16], concise sampling [17], chain sampling [18], order statistics with sampling approximate quantiles [19], and sampling from sliding windows. Another common method for processing the stream data is filtering, such as bloom filters [20]. The third most common method is to construct a probabilistic summary of the data with hash functions and randomized algorithms to get approximations [20]. In this paper, we use sampling from sliding windows to summarize the data streams.

The models proposed in this paper overcome the disadvantages of incremental neural networks and ensemble classifiers. The SMNN algorithms learns from stream data in mini-batches with sliding windows. In each mini-batch, we run the algorithm in many epochs to reduce the effect of randomized initial weights. Moreover, we pass the weights from one mini-batch to the other mini-batch in order to obtain better weights. The method of SCNN trains multiple neural networks on sliding windows and uses plurality voting on the collection of neural networks to classify new data. Moreover, we pass the weights learned from a sliding window to the subsequent sliding window and use them as the initial weights of the next neural network. By doing so, the training is much faster compared to using random weights every time. We will also show that the individual classifiers in the ensemble are diverse enough to give a good prediction.

III. Sliding Neural Network Algorithms

As discussed in Section II, each stream mining algorithm usually involves some summarization method. In both of our models, we used the *sliding windows* to summarize the data streams.

Sampling from a moving window is to maintain a *sliding window* of the most recently arrived data. In other words, the window has a fixed size L and it works as a first in first out queue. we chose sliding windows to summarize and sample our data, as it is simple and appropriate for the task of classification.

Usually, the sliding window slides one data instances each time ($d = 1$). However, it can be inappropriate if the streaming data arrives so rapidly and running algorithms on the sliding windows gets too slow. This problem can be easily fixed by increasing the length of the step for the sliding window ($d > 1$).

A. Sliding Mini-Batch Neural Networks

The first model we introduce for mining a data stream is called Sliding Mini-batch Neural Networks. Essentially, this algorithm involves "passing" the weights of neural networks. In other words, we passed the final weights of a trained neural network on a sliding window and used them as the initial weights for training a neural network on the next sliding window.

In our model, we combined the neural networks with sliding windows to train the data. Suppose the training dataset has size n. Let us fix L as the size of sliding windows and d as the length of the steps we slide each time, then we will have $s = \lfloor (n - L + d)/d \rfloor$ sliding windows in total. Denote the sliding windows by $Y_1, \ldots, Y_s$. The model is trained with following steps:

1) Randomly initialize a weight matrix $\mathbf{W}_0$.
2) Train the neural network with the L data points in sliding window Y_1 for up to m epochs. Obtain $\mathbf{W}_1$ which is the final weights matrix of the neural network. [1]
3) Train another neural network with $\mathbf{W}_1$ as initial weights on the L data points in sliding window Y_2 for m epochs. Obtain $\mathbf{W}_2$, the final weights of the neural network.
   ...
4) Repeat until we reach the last sliding window Y_s and obtain the final weights $\mathbf{W}_s$.

A graph of the Sliding Mini-batch Neural Networks is as follows:

[1] An epoch is a single pass through the entire training set

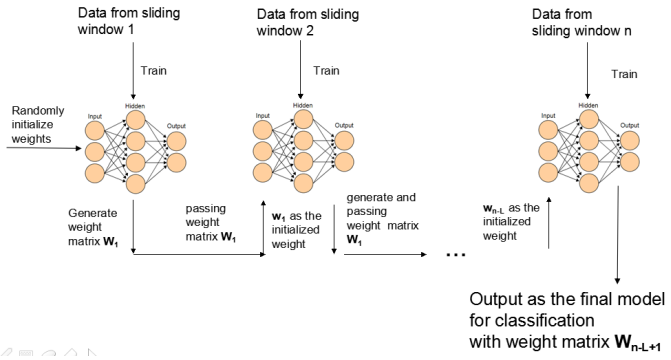


Fig. 3: Sliding Mini-Batch Neural Networks

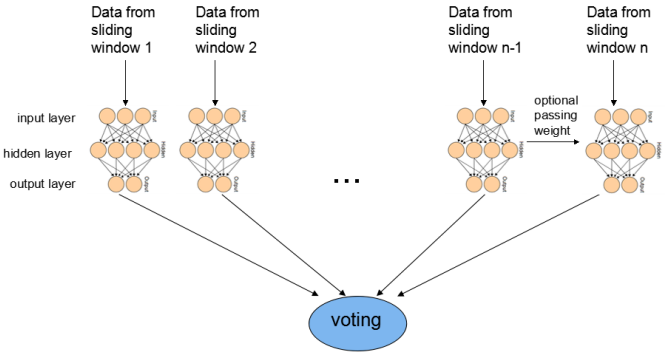


Fig. 4: Sliding Chained Neural Networks

After passing all the data in sliding windows for training, the neural network model is ready for classification. The weights matrix $\mathbf{W}_{\lfloor (n-L+d)/d \rfloor}$ will be the weights of connections for the final neural network used to classify testing data.

As we discussed in section II, SMNN optimizes the incremental neural networks algorithm since the initial weights are adjusted constantly by the L data instances in one sliding window. By feeding the data instances from one sliding window in many epochs, the order of feeding the data becomes less important. Therefore, the performance of SMNN is less random than that of incremental neural networks algorithm. Therefore, SMNN has smaller bias and variance in classification.

B. Sliding Chained Neural Networks

The second model we introduce is based on ensembles of neural networks with sliding windows, where successive models in the ensembles are correlated to each other. As shown in the paper [15], ensemble of classifiers usually works better than a single classifier in mining data streams. Therefore, the main idea is to build one neural network on each sliding window and take a majority vote. However, most of the time, the data instances in data streams are dependent. For example, the temperatures measured by a sensor are usually dependent on or correlated to the last temperature measure.

Because data instances in a data stream are usually dependent, it is natural to assume that the data instances in a sliding window are correlated to those in the previous sliding window. Due to the dependency of data instances between different sliding windows, it would be inefficient to build a neural network on each sliding window independently.

We further improved the model by choosing to pass the weights from one neural network to the subsequent neural network when there is no concept drift. By doing so, the training time gets reduced, since passing weights provides "good weights" as initial weights for each new neural network model.

This new classification model is called *Sliding Chained Neural Networks* or *SCNN*. This model works as the following figure shows.

In order for SCNN, as an ensemble method, to be more accurate than any of its individual members, it is necessary and sufficient for the individual classifiers to be accurate and diverse [21]. In this context, an *accurate* classifier is one that has an error rate better than 50%. A set of classifiers is said to be *diverse* if the classification errors of the classifiers are independent from each other [22]. Informally, this means that they don't make classification errors on the same new data points (more than would be allowed by chance alone).

In SCNN, there is a tradeoff between accuracy *of individual models* and diversity of the ensemble. The building of SCNN is counter-intuitive in terms of diversity since the classifiers in the ensemble would be more similar with weights passing. However, passing the weights in SCNN can improve the accuracy of each single classifier. In the case where the final weights of a neural network are passed and used as the initial weights of the subsequent neural network, that is considered equivalent to having trained each neural network on more data points than are available in a sliding window. In turns, this increases the accuracy of individual classifiers in the ensemble.

Of course, the end goal is to obtain the best accuracy from ensemble voting and not from individual neural networks. Thus, it is not immediately obvious whether the SMNN, SCNN or a simple ensemble of neural networks (without passing weights) would be the best approach as there is a delicate balance to establish, which we study in more details in Section V-C2.

However, the diversity issue can be further improved by careful models selection. Since the size of ensembles is fixed due to memory restrictions, an algorithm for models picking is necessary and a good algorithm can improve diversity of SCNN by picking the most diverse models to be in the ensemble.

C. Model picking methods

Models picking will help to reduce the memory needed to run the algorithm. One of the main requirements for a stream mining algorithm was *constant memory*. Models (neural networks) are added to an ensemble until its size reaches C. At this point, the ensemble may not be optimal yet, so we do not want to discard all further models and we need to determine a way to *continue learning*. In order to do so, some models will have to be dropped from the collection so that new models

can be included. The principle of models picking is that the models in the ensemble are as diverse as possible in order to give better prediction.

Moreover, in the case where the streaming datasets we are working with admits concept drifting, it is absolutely critical to have a good model picking method that will be able to keep up with the concept drift.

The current model picking methods are:

1) *First In, First Out.* The first method is to use a first in first out (FIFO) queue. The oldest model in the collection is dropped to make room for the new model. This is the simplest method and it allows the learning algorithm to adapt very quickly to changing concepts. However, it is susceptible to develop bias quickly if presented with a lot of unrepresentative training data.
2) *Random Replacement*. The second method involves random replacement with a non-uniform distribution. There are many non-uniform distributions that would do, the point is to make it such that models at the end of the collection rarely get replaced (long-term memory of the classifier), whereas models at the beginning of the collection frequently get replaced (ability to adapt quickly to change).
   For example, here is a valid way to do this. First, generate a random integer r chosen according to the geometric distribution $\exp(\lambda = 0.1K)$. Then, let $n = min(floor(r), K)$. We would then discard the model at position n (counting from 0) in our ordered collection of models and replace it with a new model.
3) *Gramian Matrix Method.* The third method is to use the error correlation rate to pick up models in order to achieve diversity in the ensemble. We want to replace those models who give the errors on the same data points. To decide how two models are diverse from each other, we compute the error correlation of all models on a testing dataset. For example, we use all data in one sliding window as the testing data. For each model, we print all errors made by this model. Therefore, we obtain a set of error vectors $X_1, \ldots, X_m \in \{0,1\}^n$, where m is the number of models in the ensemble and n is the number of data instances in the testing set. The attributes of the vectors are either 1 or 0, with 1 indicating a classification error and 0 indicating success. In order to measure the degree of diversity among the models, we then compute the pairwise dot product of all error vectors. In other words, we construct the $m \times m$ matrix $A = (a_{i,j})$ where $a_{i,j} = X_i^T X_j$. Small values of A are thus considered ideal, as they indicate strong pairwise independence between models, whereas large values of A are undesirable. Note that A is the *Gramian matrix* of $X_1, \ldots, X_m$. Models who tends to make the mistakes on the same data are undesirable in an ensemble [22] and they can readily be identified from the matrix A.

IV. Experimental Design

A. Dataset

The dataset we chose for our experiment is the Dodgers Loop Sensor Data Set from the UCI repository [23]. It comes with two file: Dodgers.data and Dodgers.events. Dodgers.data contains 50,400 instances with 3 attributes (date, time, count of cars). Dodgers.events contains 81 instances with 6 attributes representing *date*, *beginning of event time* and *end of event time*.

According to the dataset description [23], "this loop sensor data was collected for the Glendale on ramp for the 101 North freeway in Los Angeles. It is close enough to the stadium to see unusual traffic after a Dodgers game, but not so close and heavily used by game traffic so that the signal for the extra traffic is overly obvious."

The observations of this dataset were taken over 25 weeks with count aggregates every 5 minutes. Thus, 288 data instances were generated per day with each data instance representing the cars count in a 5 minutes slot. This dataset is typically used with the goal of predicting the presence of a baseball game at Dodgers stadium based upon traffic patterns.

B. Dealing with Missing Values

Our dataset has a large amount of missing values (2903 missing points in total out of 50400). We imputed the data by the following actions:

- We discarded any day with more than 10 missing values in it.
- We replaced missing values by 0 in those days containing 1–9 missing values.

After doing the above, we discarded 24 days and were left with a total of 151 days.

C. Aggregating Data

Figure 5 is the plot of a typical week of the Dodgers Loop Dataset. The blue lines indicate the beginning of games and the red lines indicates the end of games.

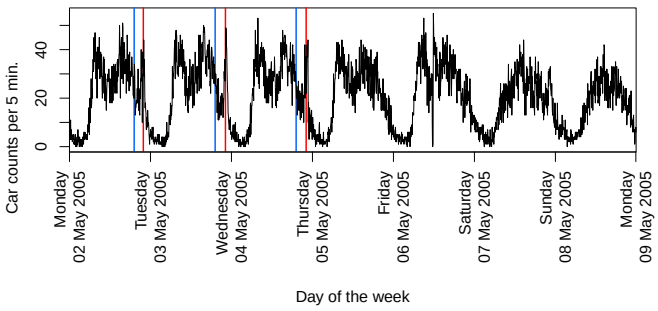


Fig. 5: A typical week for Dodgers Loop Dataset

Instead of predicting if individual data points fall within the window of a Dodgers game (between a blue line and a red line on the same day), we chose to work on the simpler

task of predicting whether or not there was a Dodgers game held on any given day. Therefore, we aggregate the data into days containing 288 integers each (as there are 288 five-minute intervals in a day). We then labeled days on which a Dodgers game was held with "1" and a day without a game as "0".

A brief observation of the figure 5 shows that there is always a spike in traffic around the time a game ends, compared to other days without a game. The difference is particularly easy to spot when comparing days of the same game, as it abstracts away variation in traffic during different days of the week. Therefore, a good classifier should be able to detect this pattern and give the correct prediction.

D. Features Selection and Dimensionality Reduction

Since neural networks can be fairly slow to train, we decided to reduce the number of attributes in our dataset. We averaged the data into 24 bins (1 bin per hour), as the following figure shows.

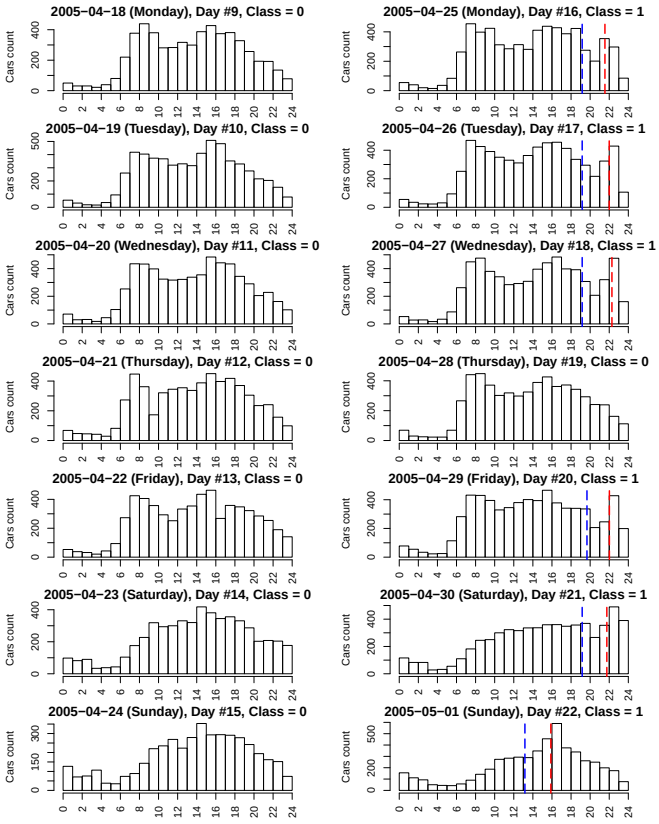


Fig. 6: Two weeks of dimensionality reduced traffic data using 24 bins per day.

As we can see from the above Figure, the spikes at the end of the game are preserved after the feature extraction.

E. Assumption and Experiment Design

For the purpose of demonstrating algorithms for data stream mining, we assumed that our dataset cannot fit in our computer's fast memory (RAM). Obviously, this is not the case for the Dodgers Loop Dataset, but this assumption allows us to design methods that would work on a computer with fixed memory M and a dataset with size much larger than M.

In both of our models, we combined the neural networks with sliding windows to train the data. For our dimension reduced model, we have 24 attributes and 2 classes. Therefore, our neural network has 24 nodes in the input layer and 2 nodes in the output layer, and we chose 4 nodes in the hidden layer.

Throughout the experiment, there are many constants. For example, the training and testing datasets are always kept constant. We list below how we chose all the relevant parameters when running SMNN and SCNN.

- We discarded the bad rows as described in subsection IV-B;
- We aggregated the data into 24 bins per day. Hence, each example has 24 numeric features;
- The training dataset is the first 70 days after discarding bad rows;
- The testing dataset is the remaining 81 days;
- We used a value of $K = 70$ for the maximum number of models per collection;
- We used various values of L for the size of each sliding window, described in each experiment;
- The neural networks are trained with 200 maximum iterations (unless otherwise specified);
- The neural networks have 4 neurons in the hidden layer.

In order to provide a baseline comparison for our stream mining learning algorithms, we trained various popular classifiers on the Dodgers Dataset with the first 70 days for training. This dataset is small, particularly after data aggregation, and becomes a toy model. We thus can easily train the popular batch learning classifiers on it. However, this is useful because it gives us an idea of what kind of results a data stream-based classifier could hope to achieve.

V. Results

A. Results of Batch Learning Classifiers

All the parameters in the batch learners are optimally chosen from experience and experiment: the random forests classifier was built with 100 trees; the k-nearest neighbors classifier used $k = 3$; the neural network was built with 4 hidden layers, 300 iterations and a decay of 0.01.

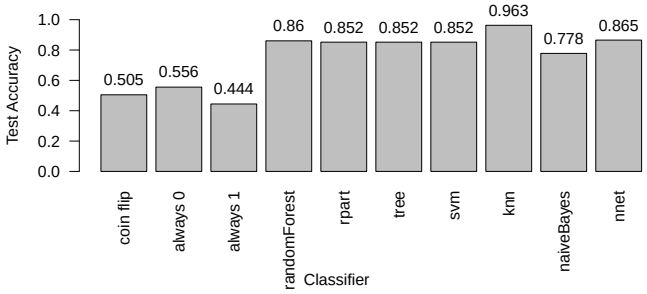


Fig. 7: Benchmark of popular classifiers trained on the full training set.

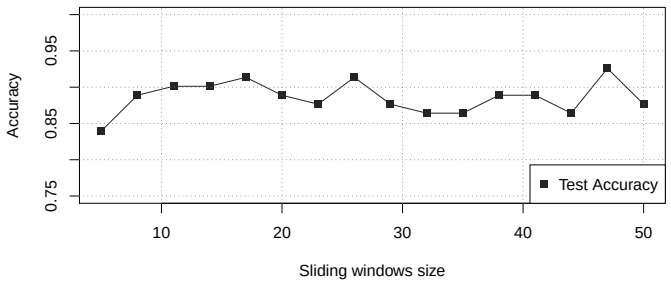


Fig. 8: Accuracy of ensemble of neural networks trained independently over all sliding windows.

Notice that k-NN performs extremely well on this dataset, with over 96% test accuracy. The other classifiers (excluding Naive Bayes, the coin flip and the constant choices) all hover around 85% to 86%, which suggests that an accuracy of 85% would be a good objective to try to match or get as close as possible to.

B. Results of Popular Stream Mining Algorithms

1) VFDT: For the VFDT, we ran the Hoeffding trees classifier in MOA (Massive Online Analysis) with the following parameter (after tuning): the grace period is 10, the test size is 81 and the sample frequency is 42 (with all other parameters using default values), this gave us a performance of 58.43%, which is the best VFDT performance we could get. This result is statistically lower than the performance of the batch leaner.

The reason may be that VFDT performs poorly on small dataset, as described in the VFML manual[24]: "VFDT will be most effective when learning from very large data sets, in the millions or billions, where there is plenty of data for it to make good statistical decisions. For smaller data sets you may consider the -batch parameter which makes it load all data into RAM and learn as a traditional decision tree learner. If you suspect that there is concept drift in the data set you may like to use the cvfdt tool instead".

2) Incremental Neural Networks: On the other hand, the performance of incremental neural networks is better. We passed the first 70 days of data one by one into a neural network while updating the weights after every single data instance. The data from the last 81 days formed the testing dataset. This incremental learning approach achieved an accuracy of 82.14%.

3) Ensemble of Neural Networks: We also trained one neural network on each sliding window independently without passing the weights. The results of this ensemble of neural networks with various size of sliding windows are as follows:

The average accuracy of this classifier is 89.0% for sliding window sizes ranging between 5 and 50.

C. Results of Sliding Neural Networks algorithms

1) Results of Sliding Mini-batch Neural Networks: We looked at the performance of the Sliding Mini-batch Neural Networks as the size of the sliding windows varies.

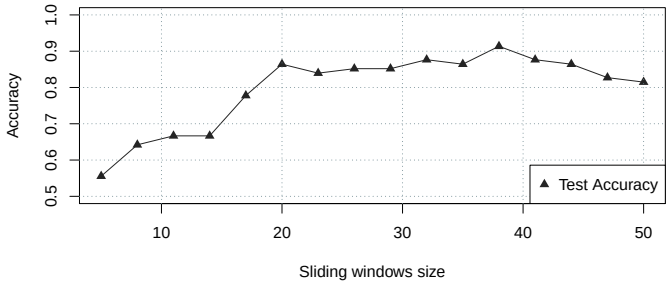


Fig. 9: Accuracy of Sliding Mini-batch Neural Networks classifier on testing datasets.

The performance is poor with smaller window sizes since the training sets tend to be poorly balanced. The average test accuracy is 79.7% for L ranging from 5 to 50. However, the average test accuracy improves to 85.9% if we restrict to $L > 20$.

2) Results of Sliding Chained Neural Networks: Next, we consider our second method for classification of streaming data: the Sliding Chained Neural Networks method. Again, all the parameters are the same as described in Section IV. The size of the hidden layer is 4, the maximum number of epochs is 50 and the decay is 0.1. We vary L from 5 to 50.

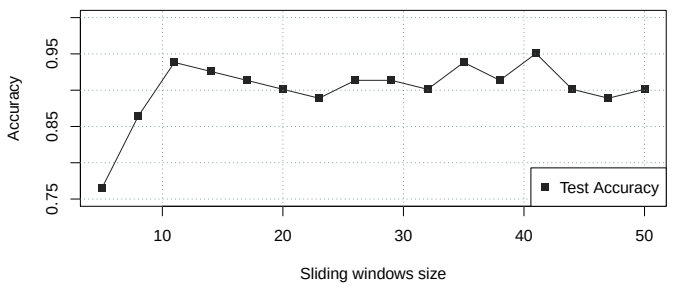


Fig. 10: Accuracy of Sliding Chained Neural Networks on testing datasets.

We get 90.1% test accuracy on average for SCNN (for sliding window sizes L ranging from 5 to 50). Moreover, as we can see from Figure 10, when the size of sliding window is small, the accuracy of SCNN is still close to that of the batch learners.

D. Comparison

Overall, both SCNN and SMNN performed better on the Dodgers Loop dataset than all the other batch learners that we tested in Section V-A, except for k-nearest neighbors. Moreover, both SCNN and SMNN outperformed the popular stream mining algorithms including VFDT and incremental neural networks on this dataset. The performance of ensembles of neural networks trained independently on each sliding window perform fairly similarly to SCNN.

1) Comparison of SMNN and SCNN: Finally, we compare the performance of SMNN and SCNN in the following figure:

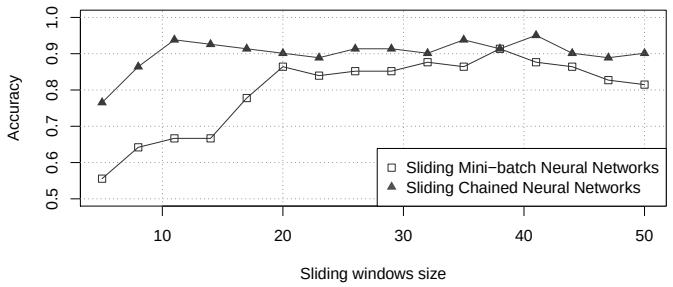


Fig. 11: Comparison of the test accuracy of the SMNN and SCNN stream mining algorithms.

Overall, Sliding Chained Neural Networks have the best performance. In particular, when the sliding window size is above 10, Sliding Chained Neural Networks significantly outperforms the other stream mining methods and benchmark classifiers except k-nearest neighbors, and the performance of SCNN converges to that of batch learners. Therefore, this model has the potential to generalize to massive streaming datasets with only a small fixed memory required for the computation, while the accuracy converges to that of batch learners.

The following table summarizes the accuracy of the various models for L ranging from 23 to 50 with steps of 3.

Size of sliding windows	Sliding Mini-batch Neural Networks (SMNN)	Sliding Chained Neural Networks (SCNN)	Ensemble of Neural Networks (ENN)
23	0.840	0.889	0.914
26	0.852	0.914	0.877
29	0.852	0.914	0.877
32	0.877	0.901	0.889
35	0.864	0.938	0.901
38	0.914	0.914	0.889
41	0.877	0.951	0.889
44	0.864	0.901	0.901
47	0.827	0.889	0.877
50	0.815	0.901	0.877
Average	**0.859**	**0.910**	**0.889**

TABLE I: Summary of test accuracy for all models.

The Sliding Chained Neural Networks performed best on our dataset, followed by the Ensemble of Neural Networks and finally by Sliding Mini-batch Neural Networks.

To see if these differences are statistically significant, we performed a Friedman test on the results of Table 1. The test results returned $\chi_F^2 = 14.889$, df = 2, p-value = 0.0005847. For a two tailed test at the 0.05 level of significance, the critical value is 7.8. Since $\chi_F^2 > 7.8$, we reject the null hypothesis that all classifiers perform the same on all 10 domains (we treat each case with a different size of sliding window as one domain).

To pinpoint where the difference lies, we applied the Nemenyi test and obtained $q_{\text{SCNN-SMNN}} = 35.78$, $q_{\text{ENN-SMNN}} = 24.60$, $q_{\text{SCNN-ENN}} = 11.18$. Then, from a q-test table, we obtain that $q_\alpha = 4.02$ for $\alpha = 0.05$ and $df = (n-1)(k-1) = 9 \times 3 = 27$ for the Tukey test. For the Nemenyi test, we divide this value by $\sqrt{2}$. This yields $q_\alpha = 2.84$. Therefore, the null hypothesis can be rejected in all cases and the difference in the performances of SMNN, SCNN, and Ensemble of Neural Networks are statistically significant.

2) Comparison of SCNN with Ensemble of Neural Networks: As shown in the previous section, the performance of SCNN is slightly better than an ensemble of neural networks trained independently on sliding windows. In this section, we want to compare these two methods in more details. We will present our results that SCNN is significantly faster to train. Moreover, we will compare the characteristics of the models in the ensembles constructed in each method and compare their accuracy and diversity.

One advantage of SCNN over Ensemble of Neural Networks is that the training time gets reduced as shown in the following experiment. We measured the time it took to train SCNN and Ensemble of Neural Networks with various sizes of sliding windows.

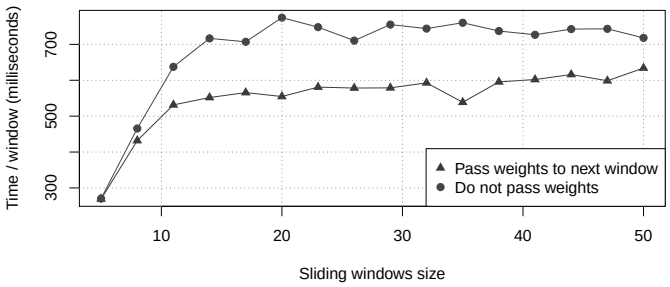


Fig. 12: Timing required to train a Sliding Chained Neural Networks (pass weights) and Ensemble of Neural Networks (not pass weights)

On average it takes 35% more time to train Ensemble of Neural Networks than SCNN, which is a significant speedup in favor of the Sliding Chained Neural Networks algorithm. This is because training a neural network from scratch with random initial weights takes more time than the situation when its initial weights are already "good", and passing weights in SCNN provides "good weights" for each new neural network.

Another advantage of SCNN is that the individual classifiers in the SCNN ensemble are averagely more accurate as the following experiment shows. We computed the mean test accuracy of individual neural networks within an SCNN ensemble and Ensemble of Neural Networks using various sliding window sizes.

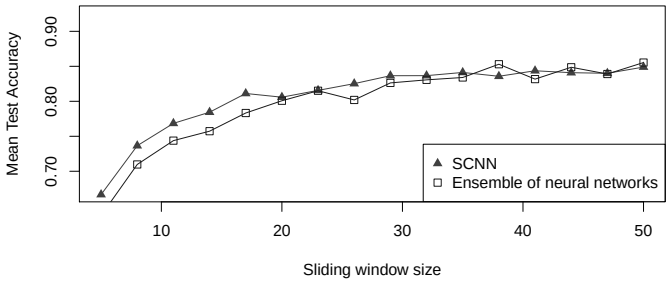


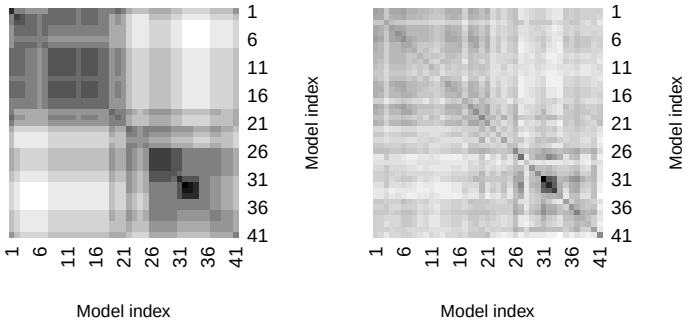
Fig. 13: Mean test accuracy of individual models within an SCNN ensemble and of individual models within an Ensemble of Neural Networks.

We can see in Figure 13 that individual models have a slightly higher mean accuracy in the SCNN methods. Yet, this does not necessarily lead to a higher ensemble accuracy since models diversity is another factor for the ensemble accuracy, which is discussed next.

In Section III-B, we suggested that there was a tradeoff between accuracy of individual models and ensemble diversity in SCNN. Intuitively, passing weights in SCNN would lead to a less diverse ensemble than Ensemble of Neural networks without passing weights. This is confirmed by the following experiment which measure the diversity in SCNN ensemble and Ensemble of Neural Networks with the Gramian matrices as discussed in Section III-C.

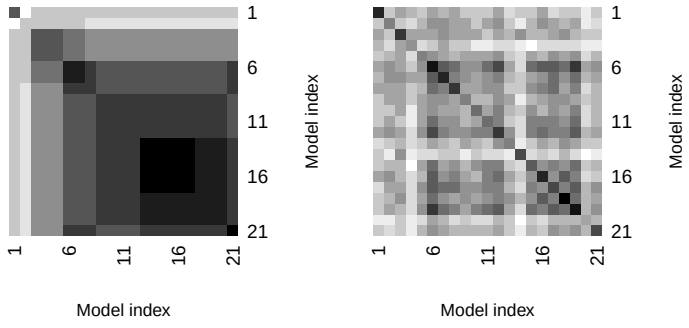
Follow the procedures in Section III-C of the Gramian matrices Methods, we measure the diversity of ensembles by computing the error vectors X_i on the testing set of each neural network in an ensemble. Then, we compute the matrix $A = (a_{i,j})$ of pairwise dot products $a_{i,j} = X_i^T X_j$ (this is called the *Gramian matrix* of $X_1, \ldots, X_m$).

We displayed the Gramian matrices as heatmaps in Figure 14 and Figure 15 for a SCNN and an Ensemble of Neural Networks with sliding window size of 30 days and 50 days, respectively.



(a) SCNN. (b) Ensemble of neural nets.

Fig. 14: Gramian matrices with sliding window of size 30. The matrix has size 41×41 since there are 41 neural networks in the ensembles. The matrices are represented as heatmaps, where the color indicates how many common mistakes in classification two models are making. The darker the color, the more common mistakes two models are making together.



(a) SCNN. (b) Ensemble of neural nets.

Fig. 15: Gramian matrices with sliding window of size 50. The matrix has size 21×21 since there are 21 neural networks in the ensembles.

As we can see from Figure 14 and Figure 15 , the heatmaps of SCNN are darker compared to Ensemble of Neural Networks. This suggests that in the SCNN algorithm, the errors matrices are highly correlated. In other words, the models in SCNN are less diverse compared to Ensemble of Neural Networks, since the models in SCNN make more mistakes on the same data instances. This corresponds to what we expected, since it is natural to assume that more correlation would be present between the models making up the ensemble when weights are being passed.

Since SCNN perform slightly better than Ensemble of Neural Networks, models diversity plays an less important role on performance of the classifiers than the mean accuracy of individual models in our experiment. This explains why SCNN still performs best in the Dodgers Dataset — in addition to providing other advantages, such as reduced processing time.

VI. Conclusion and Further Work

In this paper, we tested out two models: Sliding Mini-batch Neural Networks and Sliding Chained Neural Networks. They were both trained on sliding windows, which allows learning on labeled data of arbitrary sizes, as well as learning in an incremental fashion on continuous data that may even change with time (concept drift). We used a fairly small dataset called the Dodger Loop Sensor Data set, but the methods were developed with generalization to datasets of arbitrary size as the goal.

The results indicate that the performances of the two models we proposed are all converging to that of popular batch learners, while Sliding Chained Neural Networks performed better than Sliding Mini-Batch Neural Networks. We believed that the relatively worse performance of Sliding Mini-Batch Neural Networks came from the overfitting of the classifier. We tried reducing the number of nodes in the hidden layers and reducing the training iterations, but the result didn't improve that much. One future work can be done is to investigate on this overfitting and resolve the issue. In addition to using ensemble methods, a few other possible solutions could be to use an incremental version of PCA to reduce the number of features to make the input layer smaller.

We also noticed that there seems to be a definite relationship between the size of sliding windows L and the test accuracy of the classifiers. It is clear that for very small L, the sliding windows contain too few examples to be representative of the actual data and are frequently unbalanced, which explains the poor performance. However, for larger values of L, it is not clear what the nature of the relationship is and more research is required.

One key property of Sliding Mini-Batch Neural Networks and Sliding Chained Neural Networks is that they can be trained incrementally. This is especially important in data stream mining as the nature of the data sources is such that not only is incremental learning necessary due to technical reasons such as the size of the training set, but also the continuous stream of data is generally subject to slow changes over time, which requires the learning algorithm to adapt. This is called *concept drift* and our algorithms are able to adapt to handle such changes, as the collection of trees or neural networks keeps getting replaced as new training data comes along. Further work is required however to quantify how well our algorithms handle concept drift compared to other algorithms.

A limitation in our project was that after data preprocessing, the dataset we obtained was fairly small. Therefore, we developed and tested our methods on a toy example. However, all the methods we used should in theory generalize to a massive dataset. On such datasets, if the data streams arrive very rapidly, the sliding windows could be chosen more sparsely by using larger steps between sliding windows. We used step size of 1 in our experiments, but in theory the step size could be as large as L, the size of sliding windows. We expect that this would have a minimal impact on the classification performance of our algorithms, while providing a massive speed-up.

References

[1] Gama, João, Raquel Sebastião, and Pedro Pereira Rodrigues. "Issues in evaluation of stream learning algorithms." Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2009.

[2] Domingos, Pedro, and Geoff Hulten. "Mining high-speed data streams." Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2000.

[3] Hulten, Geoff, Laurie Spencer, and Pedro Domingos. "Mining time-changing data streams." Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2001.

[4] Domingos, Pedro, and Geoff Hulten. "Catching up with the data: research issues in mining data streams." Proc. of Workshop on Research issues in Data Mining and Knowledge Discovery. ACM, 2001.

[5] Hahsler, Michael, Matthew Bolanos, and John Forrest. "Introduction to stream: An Extensible Framework for Data Stream Clustering Research with R."

[6] M., Last. "Online Classification of Nonstationary Data Streams." Intelligent Data Analysis, 6, 129–147. ISSN 1088-467X. 2002.

[7] CC., Aggarwal, Han J, Wang J, and Yu PS (2004). "On Demand Classification of Data Streams." Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, KDD '04, pp. 503–508. ACM, New York, NY, USA.

[8] CC., Aggarwal, Han J, Wang J, and Yu PS (2003). "A Framework for Clustering Evolving Data Streams." Proceedings of the International Conference on Very Large Data Bases (VLDB '03), pp. 81–92.

[9] Gama, João, Pedro Medas, and Pedro Rodrigues. "Learning decision trees from dynamic data streams." Proceedings of the 2005 ACM symposium on Applied computing. ACM, 2005.

[10] Hoeglinger, Stefan, Russel Pears, and Yun Sing Koh. "Cbdt: A concept based approach to data stream mining." Advances in Knowledge Discovery and Data Mining. Springer Berlin Heidelberg, 2009. 1006-1012.

[11] Polikar, Robi, et al. "Learn++: An incremental learning algorithm for supervised neural networks." Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on 31.4 (2001): 497-508.

[12] Carpenter, Gail A., et al. "Fuzzy ARTMAP: A neural network architecture for incremental supervised learning of analog multidimensional maps." Neural Networks, IEEE Transactions on 3.5 (1992): 698-713.

[13] Furao, Shen, Tomotaka Ogura, and Osamu Hasegawa. "An enhanced self-organizing incremental neural network for online unsupervised learning." Neural Networks 20.8 (2007): 893-903.

[14] Shen, Furao, and Osamu Hasegawa. "Self-organizing incremental neural network and its application." Artificial Neural Networks–ICANN 2010. Springer Berlin Heidelberg, 2010. 535-540.

[15] Wang, Haixun, et al. "Mining concept-drifting data streams using ensemble classifiers." Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2003.

[16] Vitter, Jeffrey S. "Random sampling with a reservoir." ACM Transactions on Mathematical Software (TOMS) 11.1 (1985): 37-57.

[17] Gibbons, Phillip B., and Yossi Matias. "New sampling-based summary statistics for improving approximate query answers." ACM SIGMOD Record. Vol. 27. No. 2. ACM, 1998.

[18] Babcock, Brian, et al. "Models and issues in data stream systems." Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. ACM, 2002.

[19] Manku, Gurmeet Singh, Sridhar Rajagopalan, and Bruce G. Lindsay. "Random sampling techniques for space efficient online computation of order statistics of large datasets." ACM SIGMOD Record. Vol. 28. No. 2. ACM, 1999.

[20] Gama, João. "Knowledge discovery from data streams." CRC Press, 2010.

[21] Hansen, Lars Kai, and Peter Salamon. “Neural network ensembles.” IEEE transactions on pattern analysis and machine intelligence 12.10 (1990): 993-1001.

[22] Dietterich, Thomas G. “Ensemble methods in machine learning.” Multiple classifier systems. Springer Berlin Heidelberg, 2000. 1-15.

[23] A. Ihler, J. Hutchins, and P. Smyth. “Adaptive event detection with time-varying Poisson processes.” Proceedings of the 12th ACM SIGKDD Conference (KDD-06), August 2006.

[24] Domingos, Pedro, and Geoff Hulten. ”vfdt File Reference”. http://www.cs.washington.edu/dm/vfml/vfdt.html