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@carleton.ca

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 learning 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 Ensemble 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 a stream or in streams, and if it is not processed immediately or stored, then 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, 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, storing at most a small fraction of data points. Because of this, traditional batch learning algorithms are often unsuitable for data stream mining. We thus need algorithms that 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.

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 passing the final weights from the trained neural network on one sliding window to be the initial weights at the subsequent window. The second model ***S**liding **E**nsemble **N**eural **N**etwork* (SENN) works by training one neural network – completely independently – in each sliding window and taking a majority vote from all neural networks in the ensemble.

SMNN and SENN 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 stream mining algorithms. In Section VII, we conclude with remarks, discussion of the limitation 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 have become prevalent only in the last few decades; however, many literature already proposed different algorithms for stream data mining. Based on paper "Introduction to stream: An Extensible Framework for Data Stream Clustering Research with R" [14], 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) [15], On-demand Classification [16] based on micro-clusters found with the data-stream, and clustering algorithm CluStream [17]. There are some other methods we found online such as UFFT (ultra fast forest trees) [18] and CBDT (concept based decision tree) [19].

As we can see, most of these algorithms are based on decision trees. However, the decision tree structure is unstable 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 etc. To develop one decision tree, multiple scans of the whole dataset are required as the following figure indicates:

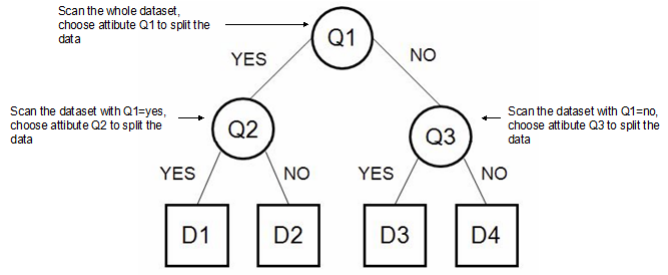

Fig. 1. An example of decision tree

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 trees 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 the following: "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.

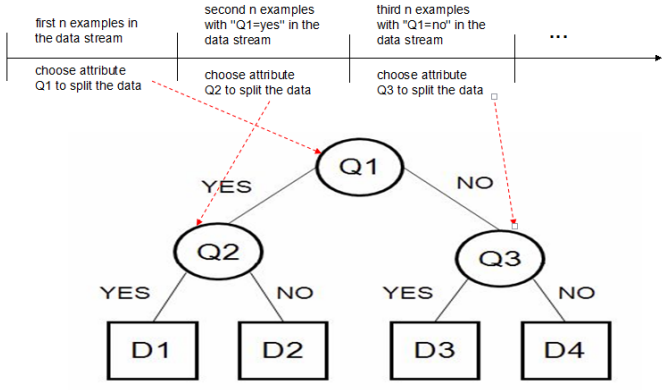

Fig. 2. An illustration of VFDT

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 tree 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 distribution of the data can cause the Hoeffding trees to fail. For example, in Figure 2, suppose that the first n examples choose Q1 as the root attribute, and we kept growing trees until we reach level 1000. Suddenly, the distribution of the data changed, choosing Q500 as the root attribute will give a much better prediction as choosing Q1. Then the tree has to rebuild again, but we can't retrieve the data passed in the stream, and we might not have enough data to rebuild another tree.

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 streaming 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 present algorithms based on incremental neural networks, such as the paper [5], [6], [7], [8] which use incremental neural networks to training data one by one, and update the weight every time after scanning a data points. The shortcoming of this incremental neural networks is that the result is highly depends on the randomized initial weight since we only pass the data once. As we know, in batch learning, we pass the data in many epochs to avoid the effect of randomized initial weight. This is impossible in incremental neural networks since the data only got passed once, and we can't retrieve the lost data.

Other researchers present methods based on ensemble classifiers, such as paper [11] propose a general framework for mining concept-drifting data streams 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 each chunk are usually dependent.

Overall, most of 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 [20], Concise Sampling [21], Chain Sampling [22], Order Statistics with sampling approximate quantiles [23], and Sampling from a Moving Window, such as sliding windows. Another common method for processing the stream data is filtering, such as bloom Filters. The third common method is to construct a probabilistic summary of the data with hash functions and randomized algorithms to get approximation.

The models present in this paper overcome the disadvantages of incremental neural networks and ensemble classifiers. Our method of SMNN learn the 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 weight. Moreover, we pass the weight from one mini-batch to the other mini-batch in order to obtain a better weight. The method of SENN trains multiple neural networks on sliding windows and uses plurality voting on the collection of neural networks to classify new data. Moreover, whenever this is no concept drift, 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 as compared to using random weights every time.

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 quite 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 came up for mining a stream of data is the Sliding Mini-batch Neural Networks. As the name of Sliding Mini-batch Neural Networks indicated, what we did is to "pass" 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 the size n. Let us fix L as the size of sliding windows and d as the length of the step we slide each time, then we will have $(n - L + d)$ **mod** d ($(n - L + d)\%d$) sliding windows in total. We train our model in the following steps:

1) We randomly initialize a weight matrix $\mathbf{W}_1$
2) Train the neural network with random assigned weight on the L data points in sliding window for up to m epochs. Obtain $\mathbf{W}_1$ which is the final weight matrix in the neural network. [1]
3) Train the neural network with $\mathbf{W}_1$ as starting weights on the L data points in sliding window 2 for m epochs. Obtain $\mathbf{W}_2$, the final weights of the neural network.

   ...
4) Repeat until we reach the last sliding window and obtain the final weights $\mathbf{W}_{(n-L+d)\%d}$.

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

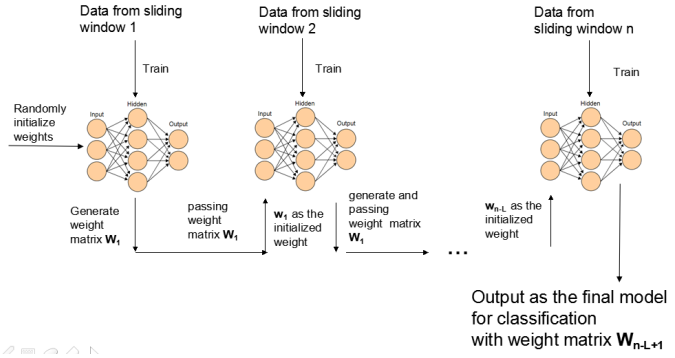

Fig. 3. Sliding Mini-Batch Neural Networks

After passing all the data in sliding windows for training, the neural network model is ready for classification. The weight matrix $\mathbf{W}_{(n-L+d)\%d}$ 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 initialized weight is adjusted constantly by the L data instances in one sliding

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

window. By feeding the data instances in one sliding window in many epochs, the order of feeding the data become less important. Therefore, the performance of SMNN is less random than incremental neural networks algorithm. Therefore, SMNN has smaller bias and variance in classification.

B. Sliding Ensemble Neural Networks

The second model we came up with is based on ensemble of Neural networks with sliding windows. As shown in paper [11], ensemble of classifiers usually works better than a single classifier in mining data streams. Therefore, we decided to build one neural network on each sliding window and take a majority vote. However, most of the time, the data instances in data stream are dependent. For example, the temperatures measured by the sensor are usually dependent on or correlated to last temperature measure.

Because data instances in a data stream are usually dependent, it is nature 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, we thus concluded that it is inefficient to build a neural network on each sliding window independently.

We thus further improved the model by passing the weights from one neural network to the succeeding neural network when there is no concept drift. By doing so, the training time would be reduced when using "good weights" as the initialization weights. Therefore, we came up with the second idea called *Sliding Ensemble Neural Networks*. This model works as the following figure shows.

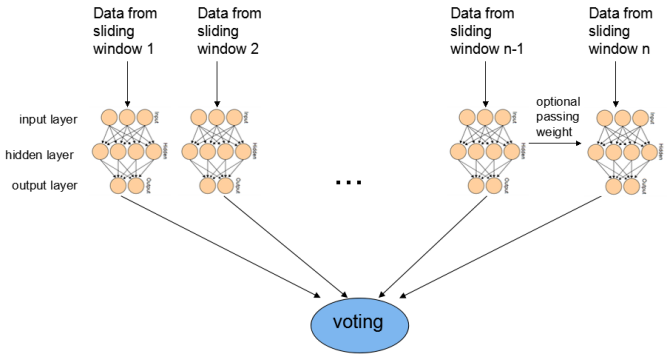

Fig. 4. Sliding Ensemble Neural Networks

One of the main requirements for a stream mining algorithm was *constant memory*, which we have not explained in the previous sections yet. This is not an issue for the Sliding Mini-batch Neural Networks algorithm, since the memory used is constant and only depends on the number of weights of the neural network. The same weights matrix is reused when moving from one sliding window to the next, so memory use does not change.

However, it is necessary to consider this issue for Sliding Ensemble Neural Networks, since it involves building a collection of classifiers. Of course, one cannot keep adding models (one per sliding windows) indefinitely.

We fix the maximum number of classifiers that a collection will use as a constant, denoted by K. This number is generally much much lower than n, the total number of sliding windows.

Models (neural networks) are added to the collection until its size reaches K. At this point, the collection is full and we need to determine a way to *continue learning*, so that the algorithm can adapt to the changing data (in the case of concept drift). In order to do so, some models will have to be dropped from the collection so that new models can be included.

We introduce two methods that we designed to pick the models.

1) 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) 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 exponential 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.

IV. Experimental Design

A. Dataset

The dataset we choose for our experiment is the Dodgers Loop Sensor Data Set from the UCI repository [12]. 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, Begin event time, End event time, Game attendance, Away team, and W/L score.

According to the dataset description [12], "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 in every 5 minutes. Thus, 288 time data instances were generated per day with each data instance representing count of cars in a 5 minutes slot. The goal of this dataset is to predict the presence of a game at Dodgers stadium.

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 indicate the end of games.

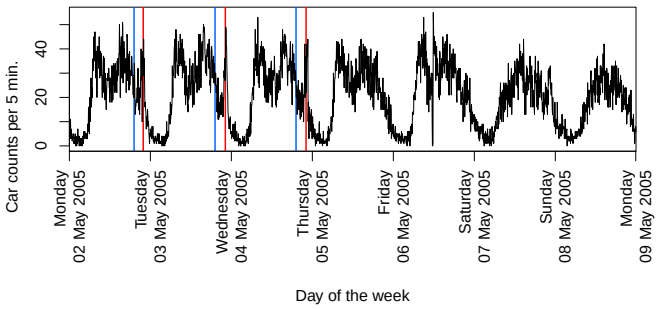

Fig. 5. A typical week for Dodgers Loop Dataset

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 labelled 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.

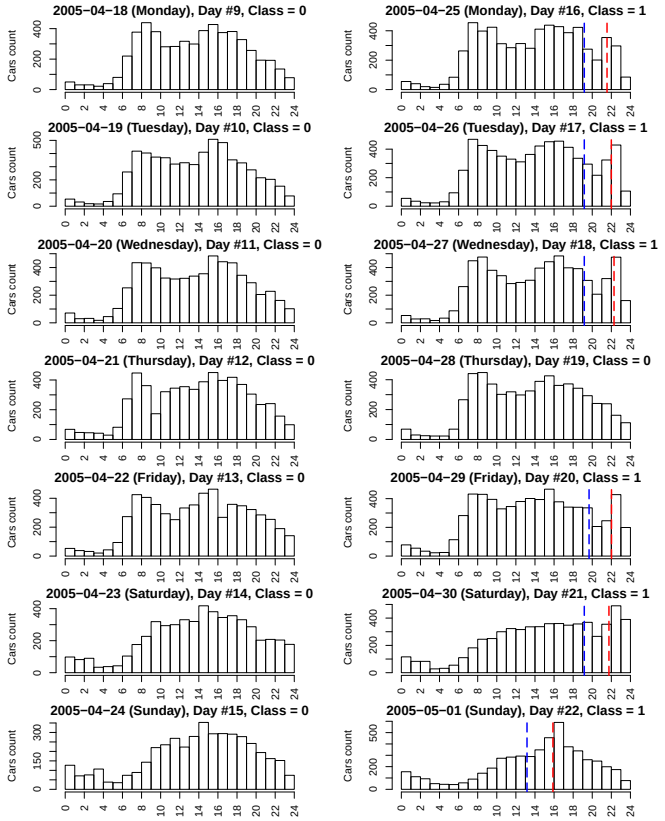

Fig. 6. Two weeks of dimensionality reduced traffic data using 24 bins per day.

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 SENN.

- 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 89 days;

- We used a value of $K = 20$ 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 after data aggregation is small 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 Benchmark 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.

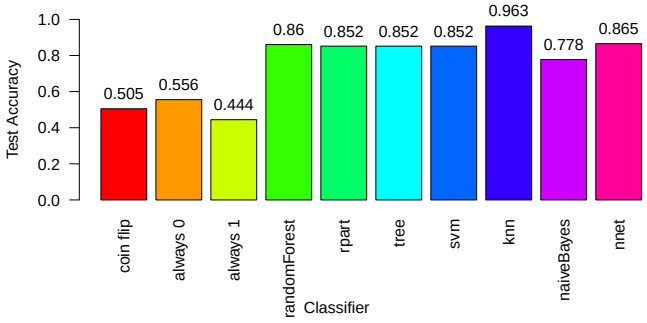

Fig. 7. Benchmark of popular classifiers trained on the full training set.

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 VFDT and incremental Neural Networks

need to perform the experiment and obtain the result for comparison.

C. 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.

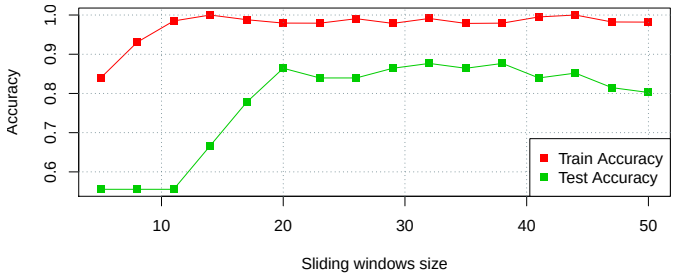

Fig. 8. Accuracy of Sliding Mini-Batch Neural Networks classifier on training and testing datasets.

The performance is poor with smaller window sizes since the training sets tend to be poorly balanced. The test accuracy with $L > 20$ is on average 85.5%. Similarly, the training accuracy for $L > 20$ is on average 98.6%.

D. Results of Sliding Ensemble Neural Networks

Next, we consider our second method for classification of streaming data: the Sliding Ensemble Neural Networks method. Again, all the parameters are the same as described in Experimental Design. 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.

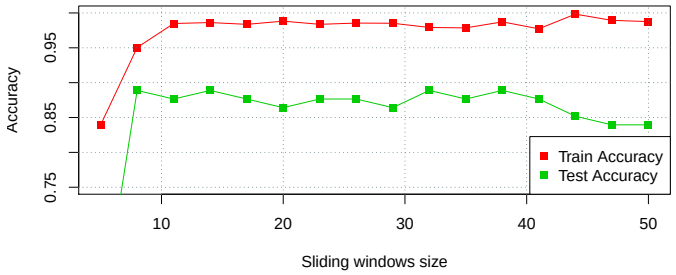

Fig. 9. Accuracy of ensemble of neural networks on training and testing datasets.

We get 86.0% test accuracy and 98.5% training accuracy. This is somewhat surprising for us, as the algorithm appears to overfit, yet the test accuracy is very good.

In the following experiments, we measured the time it took to train a neural network. We compare the Sliding Mini-Batch Neural Networks, where the weights are passed along from one sliding window to the next, to the Sliding Ensemble Neural Networks method, where neural networks are trained independently.

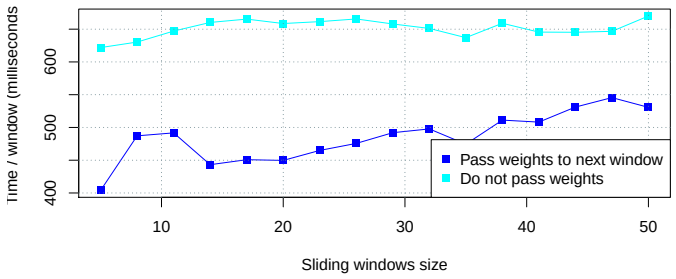

Fig. 10. Timing required to train an ensemble of neural networks if weights are passed between subsequent neural networks or if they are not passed.

On average it takes 35% more time to train a neural network from scratch than it does to train it when its initial weights are already "good" than when they are random.

Next, we looked at the accuracy of the two methods, based on the number of maximum training iterations used per sliding window. See the figure below for the results.

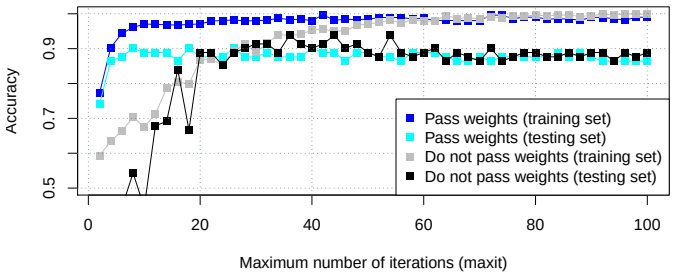

Fig. 11. Comparison of Sliding Ensemble Neural Networks trained with passing weights between subsequent neural networks and without passing them.

Without a surprise, we find that Sliding Mini-Batch Neural Networks require a lot less training iterations to reach their maximal accuracy (this makes sense, since the weights are carrier from one neural network to the next). What is really interesting, however, is that the model where the neural networks are trained independently without passing weights seem to perform slightly better when the window size is big.

There could be many reasons for this:

- Letting the initial weight matrix be random every time might help some of the neural networks reach a better global minimum, whereas passing the weights along might leave the neural networks "trapped" in a local minimum.
- Using random initial weights might simply allow the neural networks to find more varied solutions, which helps with generalization and avoiding overfitting when taking votes over those neural networks. The Sliding Mini-batch Neural Networks might be "too similar" and all make the same overfitting errors.

We studied this problem in more details. We attempted to vary the size of the hidden layers of the Sliding Mini-batch Neural Networks. We were hoping that reducing the model's complexity might help prevent overfitting. We also reduced the maximum number of iterations during training to aid in reducing overfitting as well. In either cases, the performance did not get better, but only worse.

Therefore, this gives some support to the latter theory that the neural nets simply get "stuck" in local minima.

E. Comparison

Finally, we compare the performance of both models in the following figure:

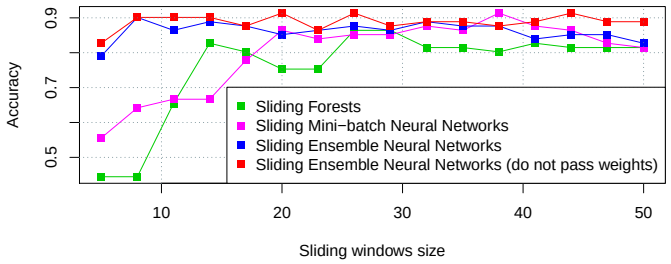

Fig. 12. Comparison of the test accuracy of the 3 streaming data classification methods.

It seems that the Sliding Ensemble Neural Networks (pass or not pass the weight) performs the best. Especially, when the sliding window size is small, Sliding Ensemble Neural Networks outperforms the other models significantly, and the performance of SENN converges to that of batch learners. Therefore, this model has a potential to generalize to massive stream datasets with only a small fixed memory required for the computation, while the result is converged to that of batch learners.

The following Table summarize the accuracy of the two models from window size=23 to 50 with step=3.

Size of sliding windows	Sliding Mini-batch Neural Networks	Sliding Ensemble Neural Networks	Sliding Ensemble Neural Networks (no weights pass)
23	0.840	0.864	0.877
26	0.827	0.864	0.864
29	0.864	0.864	0.914
32	0.889	0.864	0.889
35	0.877	0.877	0.914
38	0.889	0.877	0.889
41	0.877	0.852	0.914
44	0.827	0.840	0.914
47	0.852	0.840	0.889
50	0.802	0.840	0.926
Average	**0.855**	**0.860**	**0.897**

TABLE I. SUMMARY OF TEST ACCURACY FOR OUR THREE MODELS

The Sliding Ensemble Neural Networks in which weights were *not* passed between sliding windows performed best on our dataset, followed by Sliding Ensemble Neural Networks with passing weights, then finally 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 return $\chi_F^2 = 19.7191$, df = 3, p-value = 0.0001941. For a two tailed test at the 0.05 level of significance, the critical value is 7.8. Since the $\chi_F^2 > 7.8$, 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 $q_{\text{SMNN-SENNp}} = 1.732051$, $q_{\text{SMNN-SENNnp}} = 25.114737$, $q_{\text{SENNp-SENNnp}} = 23.382686$. Check the table of q test, 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 except the case of

$q_{\text{SMNN-SENNp}} = 1.732051$, the comparison of Sliding Mini-batch Neural Networks and Sliding Ensemble Neural Networks with passing weight.

Notice that both the Sliding Ensemble Neural Networks and the Sliding Mini-batch Neural Networks methods are algorithms in which the neural network connection weights are being passed from one neural net to the other (i.e. the final weight matrix of one neural network is used as initialization weights for the next neural network to be trained). Those two methods perform fairly similarly and they are both worse than the method which involves *not* passing the weights. We attribute this difference due to what is probably getting stuck at a local minimum.

Indeed, if good weights are reused, it makes it difficult or unlikely for the optimization algorithms to climb outside of the local minimum and find a better global minimum. Moreover, since the Sliding Ensemble Neural Networks method is an ensemble method, it benefits from having models that are as distinct and uncorrelated as possible.

VI. Streaming Context

NOTE: We might try to merge this with the previous sections so that it looks "seamless", but I decided to write it separately since it is simpler and less confusing.

NOTE: Need to do a section with and withotu concept drift.

In the previous section, we constructed the SMNN and the SENN algorithms in the batch learning context. By this, we mean that in this context, the entire dataset is available, it is practical to hold it entirely in memory and the entire dataset is partitioned into mutually exclusive training and testing sets.

However, SMNN and SENN are stream mining algorithms – the above was *the first step* in the construction. In this section, we use much larger datasets from real-world sources and show SMNN and SENN work in the stream mining context.

A. Training and testing sets in the streaming context

In the stream mining context, we make the following assumptions:

1) A fraction $\alpha \geq 0$ of the dataset is static and can be pre-loaded, call it X_α. This subset can be used for pre-training ()whenever $\alpha > 0$).
2) Rows that are outside o
3) A fraction $\beta \geq \alpha$ of the dataset, called X_β, is labeled ($\beta = 1$ is possible).
4) The rows in X_α ...
5) The rows in X_β can only be accessed in a single pass (to be more precise, the sliding windows — which contain a set of rows in X_β — can only be accessed in a single pass)
6) The memory available is limited such that it is only possible to hold at most a fraction $\gamma \ll 1$ of the dataset in memory (in practice, the value of γ depends entirely on specifics of the model, including scaling coefficients and constant memory used – but we ignore such matters for simplicity's sake).

Moreover, we assume that the dataset is *ordered*, such that all rows belong to the fraction α described in 1) appear at the beginning of the dataset, before any other row which does not belong to 1). Similarly, the fraction β of rows described by 2) appear at the beginning of the dataset.

VII. Conclusion and Further Work

In this paper, we tested out two models: Sliding Mini-batch Neural Networks, and Sliding Ensemble Neural Networks. They were both trained on sliding windows, which allows learning on labelled 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, with Sliding Ensemble 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 data or getting "trapped" in a local minimum. 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.

There are other limitations in this project which could be improved in the future work. For example, the time spent to run the algorithms was mostly ignored in this project. However, it is a very important aspect in stream data mining. Stream data mining requires algorithms that can process the data extremely fast, otherwise valuable data will simply have to be discarded. Also, neural networks are usually slow to train. Therefore, further work would involve tracking time spending on each algorithm to compare how fast they are – not only during training, but especially when making prediction for new data.

One key property of Sliding Mini-Batch Neural Networks and Sliding Ensemble 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] Geoff Hulten and Pedro Domingos. Catching up with the data: research issues in mining data streams. *Proc. of Workshop on Research issues in Data Mining and Knowledge Discovery*, 2001.

[5] 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.

[6] 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.

[7] 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.

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

[9] Abdulsalam, Hanady, David B. Skillicorn, and Patrick Martin. "Streaming random forests." Database Engineering and Applications Symposium, 2007. IDEAS 2007. 11th International. IEEE, 2007.

[10] Abdulsalam, Hanady, David B. Skillicorn, and Patrick Martin. "Classifying evolving data streams using dynamic streaming random forests." Database and Expert Systems Applications. Springer Berlin Heidelberg, 2008.

[11] 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.

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

[13] Tomasz Malisiewicz, cofounder vision.ai. http://www.meetup.com/Data-Mining/events/220316350/

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

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

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

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

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

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] Hinton,G.. Connectionist learning procedures. *Artificial Intelligence 40:185-234*. 1989.

[25] Japkowicz, Nathalie, Catherine Myers, and Mark Gluck. "A novelty detection approach to classification." IJCAI. 1995.

[26] Nathalie Japkowicz. "Supervised Versus Unsupervised Binary-Learning by Feedforward Neural Networks." Machine Learning. v:42, n:1/2, pp:97-122. 2001.