Streaming Random Forests

Hanady Abdulsalam, David B. Skillicorn, and Patrick Martin
School of Computing
Queen's University
Kingston, ON Canada, K7L 3N6
{hanady,skill,martin}@cs.queensu.ca

Abstract

Many recent applications deal with data streams*, conceptually endless sequences of data records, often arriving at high flow rates. Standard data-mining techniques typically assume that records can be accessed multiple times and so do not naturally extend to streaming data. Algorithms for mining streams must be able to extract all necessary information from records with only one, or perhaps a few, passes over the data. We present the* Streaming Random Forests *algorithm, an online and incremental stream classification algorithm that extends Breiman's Random Forests algorithm. The Streaming Random Forests algorithm grows multiple decision trees, and classifies unlabelled records based on the plurality of tree votes. We evaluate the classification accuracy of the Streaming Random Forests algorithm on several datasets, and show that its accuracy is comparable to the standard Random Forest algorithm.*

Keywords Data mining, Classification, Decision trees, Data-stream classification, Random Forests.

1. Introduction

Many applications such as internet traffic monitoring, telecommunications billing, near-earth asteroid tracking, closed-circuit television, and sales tracking produce huge amounts of data to be analyzed. It is not usually cost-effective to store all of this data. Such data is a conceptually endless, real-time, and ordered sequence of records and so is modelled as a *stream*. In order to extract knowledge from stream data, existing data-mining algorithms must be adapted to reflect the properties of streams.

Data stream mining algorithms face several issues that are not critical for ordinary data-mining algorithms:

- Algorithms must be online and incremental, so that results can be produced at any time (perhaps after some initial starting period). In particular, the relationship of the accuracy of the model to the amount of data seen must be understood.
- Algorithms must be fast enough to handle the rate at which new data arrives (which means effectively amortized $\mathcal{O}(1)$ time for both learning and prediction/clustering).
- Algorithms should be adaptive to changes in the distribution of values in the underlying stream as the result, for example, of concept drift, since they may run for long periods of time.
- Results that depend on observing an entire data set cannot be computed exactly, so the results of stream mining must necessarily be an approximation.

Data-stream mining has attracted a great deal of attention, with research in, for example, detecting changes in data streams [15, 16], maintaining statistics of data streams [7, 20], data-stream classification, and data-stream clustering [3, 11]. In this paper, we consider the problem of classification for stream data.

Classification is normally considered as requiring three phases, each with its associated data. In the first phase, a model is built using labelled training data; in the second phase, the model is tested using previously-unseen labelled data (test data); and in the third phase, the model is deployed on unlabelled data. In stream classification, there is only a single stream of data, so the problem must be formulated in a different way. We assume a setting in which some records in the stream are labelled, which are used for building or testing the model, while others are not, and the goal is to predict the class of the unlabelled records.

Within this setting there are several different scenarios, depending on how the (labelled) training examples are distributed through the stream. Some possibilities are shown in Figure 1. In Scenario 0, labelled data records occur only in some initial segment of the data stream. In this case, the

only new issues are related to performance. The classifier can be built in a standard (offline) way, but must be built quickly enough to keep up with the arrival rate of the labelled records. Similarly, classification must be fast enough to keep up with the arrival rate of unlabelled records. Some subset or suffix of the labelled records can be used as test data. In this scenario, there is no way to respond to changes in the underlying distribution of the records such as concept drift.

In Scenario 1, labelled records occur regularly in the stream but, when they occur, there are enough of them to build and test a robust classifier. This classifier can then be used to classify the subsequent unlabelled records. However, after some time, new labelled records arrive in the stream, and the existing classifier must take these into account, either building a new model to reflect the structure of the new labelled data, or altering the existing model (if necessary) to reflect the new information. The decision about which approach to take depends on how much variability is expected, but a full range of responses are available.

In Scenario 2, labelled records again occur regularly in the stream, but there are not enough of them in each batch or block to build and test a robust classifier. Classification of unlabelled records could begin very early, with appropriate caveats about accuracy, or might be postponed until the classifier is observed to be working well, which might require several batches of labelled records. However, in this scenario, the classifier can be very sensitive to changes in the labelled data, and reflect them in classification of unlabelled records very rapidly. In both Scenarios 1 and 2, the known frequency of labelled records enables the algorithm to amortize the cost they induce to train the classifier more efficiently.

In Scenario 3, labelled records occur at random in the stream. This is the most challenging, but also the most realistic, situation since the classifier must be able to keep up with using the labelled examples to build or modify the classifier, even when they appear close together in the stream.

Classification of unlabelled records could be required from the beginning of the stream, after some sufficiently long sequence of labelled records, or at specific moments in time and for a specific block of records selected by an external analyst.

In many situations, records are able to be labelled because the passage of time has revealed the appropriate class labels for each one. For example, a classification system for approving mortgage applications is based on historical data about which applicants previously granted mortgages repaid them without problems. Clearly, the problem is a two-class classification problem, that is either the application is approved or not. Another example for stream classification is spam detection, where an incoming stream of emails (unlabelled records) could be supplemented by emails that have already been delivered to users' mailboxes and have been (implicitly) labelled as spam or not based on whether the user deleted them unread or read them. These labelled records could help to improve the spam classification model continuously based on user feedback.

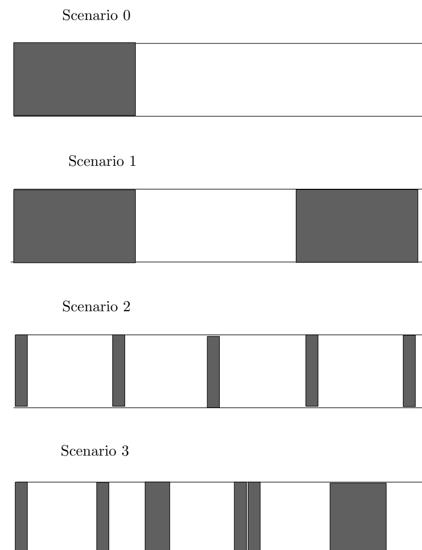

Figure 1. Possible scenarios for the embedding of labelled records in streams

In other situations, *some* records are labelled because only *some* people or objects participate. For example, traffic delays in congested areas are beginning to be predicted based on the transit time of cell phones from tower to tower (or from specialized traffic direction systems available in some cars). Only a small percentage of cars provide this information, but the resulting data can be used to classify particular routes as free-flowing or congested. The data used to provide these predictions matches Scenario 3, since the labelled records (cars with known transit times along a particular route) arrive at random compared to the requests from cars for route classification.

Some applications, for example spam detection, do not require that unlabelled records are classified all the time. There may be periods where it is possible to ignore the unlabelled records and only consider the labelled ones to train the classifier. Such situation matches Scenario 1.

Decision trees have been widely used for data-stream classification [9, 13]. These are typically based on Hoeffding bounds, which indicate when enough data have been seen to make a robust decision about which attribute and which split value to use to construct the next internal node.

Tree ensembles have also been used for stream classification [8, 10, 12, 18, 19]. The Random Forest algorithm by Breiman [5] is a classification algorithm that grows mul-

tiple binary decision trees, each from a bootstrap sample of the data. The splitting decision at each node selects the best attribute on which to split from a small set of attributes chosen randomly. Classification is based on the plurality of votes from all of the trees. The use of bootstrap samples and restricted subsets of attributes makes it a more powerful algorithm than simple ensembles of trees.

The contribution of this paper is to define a new algorithm, *Streaming Random Forests*, a classification algorithm that combines techniques used to build streaming decision trees with the attribute selection techniques of Random Forests. We demonstrate that the streaming version of random forests achieves classification accuracy comparable to the standard version on artificial and real datasets using only a single pass through the data. Our Streaming Random Forest algorithm handles only numerical or ordinal attributes for which the maximum and minimum values of each attribute are known, but it can be easily extended to handle categorical attributes. It also handles multi-class classification problems, in contrast to many stream classification techniques that have been designed and/or tested only on two-class problems [8, 9, 13, 18, 19].

This paper is organized as follows: Section 2 describes some background and related work. Section 3 defines the Streaming Random Forest algorithm. Section 4 describes experiments and results. Finally, Section 5 draws some conclusions.

2. Related work

2.1. Attribute selection for standard decision trees

Each internal node of a decision tree is built by determining the attribute whose values are most discriminative among the target classes, based on one of a number of criteria such as information gain or Gini index importance [6]. For numeric attributes, each internal node is an inequality involving the selected attribute and the best split point for the inequality must also be determined.

The Gini index is one example of a measure of impurity among the data records that are considered at a node of the tree. For a dataset D that contains n records from k classes,

$$Gini(D) = 1 - \sum_{i=1}^{k} p_i^2$$

where p_i is the ratio of the number of records in class i to the total number of records in the set D. If the set D is split into two subsets D_1 and D_2, each having a number of records n_1 and n_2 respectively, then

$$Gini(D)_{split} = \frac{n_1}{n} Gini(D_1) + \frac{n_2}{n} Gini(D_2)$$

The best attribute on which to split is the one that maximizes $Gini(D) - Gini(D)_{split}$.

2.2. Decision trees for data streams

The biggest problem in extending decision trees for data streams is that the measures of attribute importance used to determine the best choice of attribute require counts or probabilities computed over all of the training data. Clearly this is not possible when the data is a stream.

A tree under construction consists of internal nodes, containing an inequality on one of the attributes, *frontier nodes*, nodes that have not yet been either split or turned into leaves, and leaf nodes. Initially, a tree consists of a single frontier node. As each frontier node is considered, a mechanism is needed to decide when to make a selection of the 'best' attribute, or perhaps to not split this node further and convert it to a leaf. The solution is to let each new training record flow down the tree according to the inequalities of the existing internal nodes until it reaches a frontier node. When a frontier node has accumulated 'enough' records that the standard technique for splitting will give a robust result, it is split and its descendants become new frontier nodes. Alternatively, if a frontier node has accumulated records that are predominantly from one class, it may become a leaf.

The Hoeffding bound provides a way to estimate when the number of records accumulated at a node is 'enough' for a robust decision [9]. The Hoeffding bound states that, given a random variable r in the range R, and n independent observations of r having mean value $\overline{r}$, the true mean of r is at least $\overline{r} - \epsilon$, where

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

with probability $1 - \delta$, where δ is a user-defined threshold probability [9]. The Hoeffding bound ensures that no split is made unless there is a confidence of $1 - \delta$ that a particular attribute is the best attribute for splitting at the current node.

The Hoeffding bounds is used in following way. Assume that we have a general function G that checks an attribute's goodness for splitting at a specific internal node of the decision tree. At each point in tree construction, G is calculated for all attributes and the best and second best attributes are chosen to calculate $\Delta G = G_{highest} - G_{second_highest}$. The algorithm then recalculates G for all attributes as each new record arrives, and updates ΔG continuously until it satisfies a stopping condition, $\Delta G > \epsilon$. At this point, the true value of the largest G is within ϵ of the approximated G with probability $1 - \delta$ and therefore the attribute with the highest G is the best choice for splitting at the current node with confidence $1 - \delta$.

2.3. The standard Random Forest algorithm

The Random Forests algorithm is a classification technique developed by Breiman [5]. Superficially, random forests are similar to ensembles of binary decision trees. Suppose that a dataset contains n records, each with m attributes. A set of decision trees are grown, each from a subset of the n records, chosen from the dataset at random with replacement. Hence the training dataset for each tree contains multiple copies of the original records. Random selection with replacement ensures that about $n/3$ of the records are not included in the training set and so are available as a test set to evaluate the performance of each tree.

The construction of a single tree uses a variant of the standard decision tree algorithm. In the standard decision-tree algorithm, the set of attributes considered at a node is the entire set of attributes that have not yet been used in parent nodes. By contrast, in the Random Forest algorithm, the set of attributes considered at each internal node is a randomly chosen subset of the attributes, of size $M \ll m$. Trees are not pruned. A random forest is deployed as if it was an ensemble classifier, that is the classification for each new record is the plurality of the votes from each of the trees.

The Random Forest algorithm classification error depends on two things:

- The correlation among the trees: the smaller the correlation among the trees the more variance cancelling takes place as the trees vote, and therefore the smaller the error rate.
- The strength of each individual tree: the more accurate each tree is, the better its individual vote, and therefore the smaller the error rate.

The value of M is a parameter to the algorithm and must be chosen with care. A small value for M decreases the correlation between the trees, while a large value increases the strength of each individual tree.

3. The Streaming Random Forest algorithm

We extend the standard Random Forest algorithm so that it can be applied to streaming data. The difficulty in doing this is that the Random Forest algorithm makes multiple passes over the training data, in two different ways. First, a pass through the data is made to create the training data for each tree that will be built. Second, for each tree, a pass is made through (some columns of the data) for each internal node of the tree, to generate the counts that are used to decide which attribute and split point to select for it.

A streaming algorithm does not have the luxury of multiple passes over the data. To build each tree, a separate batch or block of labelled records is used. As a result, the Streaming Random Forest algorithm requires substantially more labelled data than the standard algorithm to build a set of trees. It would be conceivable to use a separate batch of labelled data to make the decision about each internal node, but this would require an even larger amount of labelled data, and would increase the time before robust classifications could be made for unlabelled records. Instead, we adapt ideas from streaming decision tree algorithms [9, 13] to route every labelled record to an appropriate node of the tree under construction, so that every labelled record contributes to a decision about one node.

The Streaming Random Forest algorithm builds a set of trees, just as the standard Random Forest algorithm does. For the time being, the algorithm takes the required number of trees as a parameter, but extensions in which the number of trees is derived from the classification accuracy, or the set of trees has new members added and old ones deleted to respond to changes in the labelled data are straightforward.

As a new labelled record arrives, it is routed down the current tree, based on its attribute values and the inequalities of the internal nodes, until it arrives at a frontier node. At the frontier node, the attribute values of the record contribute to *class counts* that are used to compute Gini indexes. To be able to maintain information about the distribution of attribute values and their relationship to class labels, attributes are discretized into fixed-length intervals. The boundaries between these intervals are the possible split points for each attribute.

The procedure for deciding when and how to change a frontier node into another kind of node is somewhat complex. A parameter, n_{min}, is used to decide how often to check whether a frontier node should be considered for transformation.

Whenever a node accumulates n_{min} labelled records, the Gini index and the Hoeffding bound tests are applied. If the Hoeffding bound test is satisfied, then the frontier node has seen enough records to determine the best attribute and split value. The Gini index test is then used to select the best attribute and split point, and the frontier node is transformed into an internal node with an inequality based on this attribute and split point [5]. The two children of this node become new frontier nodes.

If the number of records that have reached the frontier node exceeds a threshold called the *node window*, and the node has not yet been split, the algorithm checks to see if this node should instead be transformed into a leaf. If the node has accumulated records that are predominantly from one class, then it is transformed into a leaf. Otherwise, the node is transformed to an internal node using the best attribute and split point so far.

The size of the node window threshold depends on the depth of the node in the tree, because fewer records will reach deeper nodes. We therefore define the node window as a function of the tree window and the node level:

$$\frac{1}{\alpha}(\text{tree window}/(2^{nodelevel})$$

so that the node window shrinks linearly with depth in the tree. The value of the parameter α is determined empirically.

The construction of a tree is complete when a total of *tree window* records have been used in its construction. The algorithm then begins the construction of the next tree, until the required number of trees have been built.

A limited form of pruning is necessary, because a node may have generated two descendant frontier nodes that do not see enough subsequent records to be considered for splitting themselves. If two sibling nodes have failed to receive enough records when the *tree window* is reached, the node purity is calculated for both. Node purity is the ratio of the number of instances of records labelled with the most frequent class label to the total number of records. If both siblings' node purity is less than *1/number of classes*, then both nodes are pruned and their parent node is labelled as a leaf rather than an internal node. Otherwise, the sibling nodes become leaf nodes labelled with the majority class among the records in their parent that would have flowed down to them.

The entire tree building procedure for the Streaming Random Forest is shown in Figure 2.
This algorithm handles a single phase of learning and classification in Scenario 1. It can be extended to multiple phases by growing new trees from subsequent batches of labelled records, and discarding the oldest trees from the previous phase.

The standard Random Forest algorithm relies on samples chosen using random selection with replacement, both to guarantee attractive properties of the learned models and to provide a natural test set. Sampling with replacement is not possible when the data arrives as a stream, so we must consider whether this affects the properties of the new algorithm.

For an infinite stream of data drawn from the same distribution, the results produced by sampling with replacement and sampling without replacement are not distinguishable, since each outcome is independent of the previous one. This is because the covariance between two records x_i and x_j, where $i \neq j$, sampled without replacement, depends on the dataset size: $cov(x_i, x_j) = -\frac{\sigma^2}{n-1}$, where σ^2 is the population variance and n is the set size. As n becomes large, the covariance tends to zero, and sampling is effectively independent, exactly as if sampling with replacement had been used. Part of the motivation for sampling with replacement

```
procedure BuildTree
/*grow tree*/
 while more data records in the tree window
  read a new record
  pass it down the tree
  if it reaches a frontier node
   if first record at this node
    randomly choose M attributes
   find intervals for each of the M attributes
   update counters
   if node has seen n_min records
    if Hoeffding bounds test is satisfied
     save node split attribute
     save corresponding split value
   if no more records in the node window
    if node records are mostly from one class
      mark it as leaf node
      assign majority class to node
    else
     save best split attribute seen so far
     save corresponding split value
 end while
/* prune tree */
 while more frontier nodes
  if node has records arrive at it
   mark it as leaf node
   assign majority class to it
  else /* node has zero records */
   if sibling node is frontier with no records
    calculate purities of both sibling nodes
    if purities < pre-defined threshold
     prune both nodes
     mark parent node as a leaf
     assign majority class to it
  else
    mark node as leaf node
    assign dominant class to it
 end while
end
```

Figure 2. Building a tree in Streaming Random Forest algorithm

is also to increase the effective size of the training and test sets, and this is clearly not necessary for infinite datasets.

However, during the very early stages of tree construction, only a very small number of records have been seen, and the covariance will not be close to zero. To make sure that sampling without replacement does not have a noticeable effect, perhaps distorting the behaviour of the first few trees, we performed experiments simulating sampling with replacement. We did this by randomly retaining labelled records, with probability 1/3, and applying them as training examples twice (by double incrementing the counts), until the observed covariance values become small. The results of these experiments show that, in practice, sampling without replacement does not decrease the accuracy of the constructed forest.

4. Experimental Results

Our implementation of the Streaming Random Forest algorithm is based on the open-source Random Forest Fortran code by Breiman and Cutler [1]. We implement Scenario

1 from Figure 1. The labelled records at the beginning of the stream are used as training records, with later labelled records used as test records.

We evaluate the performance of the Streaming Random Forests algorithm by comparing its classification accuracy with that of the standard Random Forest algorithm using the same datasets. The training set for standard Random Forests is, however, a small randomly chosen subset of the training set for Streaming Random Forests. The reason is, as mentioned before, that the streaming algorithm observes data records only once, and therefore requires much more data than the standard algorithm, which can use data records many times for building different nodes. We also consider the classification time per unlabelled record for different forest sizes to estimate the flow rate that the algorithm can handle.

Results for each dataset are averages over 50 runs, selecting different random subsets of attributes for each run. The number of attributes considered at each internal node is M, chosen as suggested by Breiman to be $M = int(\log_2 m + 1)$.

4.1. Classification accuracy

Synthetic data

We generate synthetic data sets using the DataGen data-generation tool [17]. Each dataset has 1 million records, 5 numeric attributes, and 5 target classes. We vary the noise to generate 6 datasets containing 1%, 3%, 5%, 7.5%, 10%, and 15% noise, respectively. Each dataset is used by both the standard and streaming random forest algorithms. The training set for the standard random forest is 1% of the records randomly chosen from the original data set, giving about 10,000 records. The test set for both the standard and streaming algorithms is 0.2% of records randomly chosen from the original dataset, giving about 2000 records. There is no overlap between the training sets and the test set. The training set for the Streaming Random Forest algorithm is therefore the remaining 99.8% of the original data set, about 998,000 records.

For the Streaming Random Forest algorithm, we discretize each attribute's values into 200 intervals, and set $n_{min} = 200$ and $\alpha = 8$. The tree window is set to be the total number of training records divided by the number of desired trees, giving around 19,960 records to be used for growing each tree. This is a much smaller number of records than the number used in other streaming decision tree construction [9, 13, 14].

For each of the synthetic datasets, both algorithms grow 50 trees with $M = 3$ attributes considered at each node. Figure 3 shows the classification error rates of both algorithms for the six synthetic datasets.

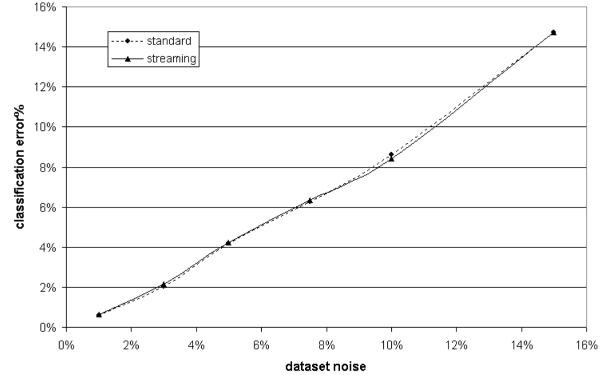

Figure 3. Classification error rates for standard and streaming random forest algorithms

The algorithms have comparable classification error rates. The confidence intervals for each test point at a confidence level of 99% overlap except for the dataset with 3% noise. The confidence intervals for this dataset are [2.02%–2.08%] and [2.09%–2.19%] respectively.

Real data

We use the Forest CoverType dataset from the UCI repository [2]. The dataset has 12 attributes (10 numerical and 2 categorical), 581,012 records, and 7 classes with frequencies of about 37%, 48.5%, 6%, 0.5%, 1.5%, 3%, and 3.5%, respectively. This dataset has been widely used in experiments reported in the literature. The classification error rates reported by Blackard [4], for example, are 30% using a neural network and back propagation, and 42% using linear discriminant analysis.

We first test the standard Random Forest algorithm by randomly sampling 0.2% of the data records for training, and 0.1% for testing. This gives a classification error of 23%. Deleting the 2 categorical attributes increases the error slightly to around 26%. Since the current implementation of the Streaming Random Forest algorithm handles only numerical attributes, we deleted the two categorical attributes. We also deleted the records for target classes 3 through 7, since they have very low frequencies compared to target classes 1, and 2. The resulting dataset has two target classes with frequencies of 43% and 57%, respectively, and ten attributes. This new dataset is used in our comparative experiments.

The training set for the standard Random Forest algorithm contains about 1000 records and the test sets for both algorithms contains about 500 records. The training set for the Streaming Random Forest algorithm therefore contains about 496,000 records.

For both standard and streaming algorithms, we grow

Table 1. Classification errors and confusion matrices for the Forest CoverType data set

	Standard Random Forest	**Streaming Random Forest**
Classification Error%	24.73%	24.96%
Test Set	219 from class 1 285 from class 2	
Confusion Matrix	*T*rue class *1* *2* *1* 147 53 *2* 72 232	*T*rue class *1* *2* *1* 149 56 *2* 70 229

150 trees using $M = 4$ attributes for building each node. The value of α used is 4, the value of n_{min} is 300, and the number of intervals into which each attribute is discretized is 300, since the ranges are quite large. As before, the tree window is the total number of training records divided by the number of trees, about 3300 records per tree, and errors are averages over 50 runs.

Table 1 presents the classification errors and confusion matrices of both algorithms for the Forest CoverType date set. Both algorithms have a classification error of approximately 25%, with a confidence interval of $\pm 0.4\%$ at a confidence level of 99%. The two algorithms have equivalent confusion matrices as well. This demonstrates that the Streaming Random Forest algorithm is as powerful as the standard Random Forest algorithm on real data.

4.2. Classification time per record

In the scenario we have been considering, the per-record classification time is the rate-limiting step because each new record must be evaluated by all of the trees (although in other scenarios, the per-record training time will also be important).

We base our classification time measurements on the synthetic dataset with a noise level of 15%. We use a Pentium 4 system with 3.2 GHz processor and 512MB RAM, and consider the effect of different forest sizes, that is different numbers of trees. The forest sizes use are 5, 50, 100, 150, 200, 250, 300, 350, 400, 450, and 500 trees.

The per-record classification times are shown in Figure 4. The times are averaged over 50 runs, and measured in microseconds. The increase in per-record classification time with number of trees in the forest is no worse than linear. The average flow rate of a stream that this implementation of the Streaming Random Forest algorithm can handle is 1.7×10^4 records/sec for forests with up to 500 trees. A more typical number of trees used in a random forest is perhaps 50 to 200 trees (according to Breiman's experiments [5]) which would allow a stream rate of up to 2.8×10^4 records/sec.

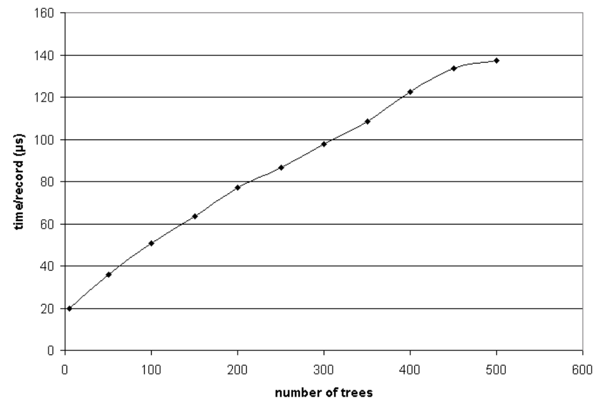

Figure 4. Per-record classification time

5. Conclusion

This paper has defined the *Streaming Random Forest* algorithm, an online and incremental stream classification algorithm. It is an extension of the standard Random Forest algorithm due to Breiman [9]. The algorithm gives comparable classification accuracy to the standard Random Forests algorithm despite seeing each data record only once. Because stream algorithms can never see 'all' of the data, our algorithm uses node windows and tree windows to decide when to begin constructing new trees, transform frontier nodes, or carry out a limited form of pruning. These refinements mean that the algorithm requires many fewer labelled records for training than other stream-based decision tree algorithms. The Streaming Random Forests algorithm is fast enough to handle streams in many applications. Its per-record classification time complexity is $\mathcal{O}(t)$, where t is the number of trees in the forest.

References

[1] Random Forest FORTRAN Code. Available from http://www.stat.berkeley.edu/breiman/RandomForests/cc_software.htm/.

[2] Forest CoverType data set. Available from http://kdd.ics.uci.edu/.

[3] C. Aggarwal, J. Han, J. Wang, and P. Yu. Sa framework for clustering evolving data streams. In *Proceedings of 29th International Conference on Very Large Data Bases(VLDB)*, pages 81–92. Berlin, Germany, 2003.

[4] J. Blackard. *Comparison of Neural Networks and Discriminant Analysis in Predicting Forest Cover Types*. PhD thesis, Department of Forest Sciences. Colorado State University, Fort Collins, Colorado, 1998.

[5] L. Breiman. Random forests. Technical Report, 1999. Available at www.stat.berkeley.edu.

[6] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth International, Belmont, Ca., 1984.

[7] A. Bulut and A. Singh. A unified framework for monitoring data streams in real time. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pages 44–55. Tokyo, Japan, 2005.

[8] F. Chu, Y. Wang, and C. Zaniolo. An adaptive learning approach for noisy data streams. In *Proceedings of the 4th IEEE International Conference on Data Mining (ICDM)*, pages 351–354. Brighton, UK, November 2004.

[9] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 71–80. Boston, MA, August 2000.

[10] W. Fan. A systematic data selection to mine concept-drifting data streams. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 128–137. Seattle, Washington, August 2004.

[11] M. Gaber, S. Krishnaswamy, and A. Zaslavsky. Cost-efficient mining techniques for data streams. In *Proceedings of the 1st Australasian Workshop on Data Mining and Web Intelligence (DMWI)*, pages 81–92. Dunedin, New Zealand, 2003.

[12] J. Gama, P. Medas, and R. Rocha. Forest trees for on-line data. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC)*, pages 632–636. Nicosia, Cyprus, March 2004.

[13] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data mining (KDD)*, pages 97–106. San Francisco, CA, August 2001.

[14] R. Jin and G. Agrawal. Efficient decision tree construction on streaming data. In *Proceedings of 9th International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 571–576. Washington, DC, August 2003.

[15] D. Kifer, S. Ben-David, and J. Gehrke. Detecting change in data streams. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 180–191. Toronto, Canada, 2004.

[16] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen. Sketch-based change detection: methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM Internet Measurement Conference (IMC)*, pages 234–247. Miami Beach, FL, 2003.

[17] G. Melli. Scds-a synthetic classification data set generator. Simon Fraser University, School of Computer Science, 1997.

[18] H. Wang, W. Fan, P. Yu, and J. Han. Mining concept-drifting data streams using ensemble classifiers. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 226–235. Washington, DC, August 2003.

[19] X. Zhu, X. Wu, and Y. Yang. Dynamic classifier selection for effective mining from noisy data streams. In *Proceedings of the 4th IEEE International Conference on Data Mining (ICDM)*, pages 305–312. Brighton, UK, November 2004.

[20] Y. Zhu and D. Shasha. Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, pages 358–369. Hong Kong, China, 2002.