STREAMING RANDOM FORESTS

by

HANADY ABDULSALAM

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen’s University
Kingston, Ontario, Canada
July 2008

Copyright © Hanady Abdulsalam, 2008
Abstract

Recent research addresses the problem of data-stream mining to deal with applications that require processing huge amounts of data such as sensor data analysis and financial applications. Data-stream mining algorithms incorporate special provisions to meet the requirements of stream-management systems, that is stream algorithms must be online and incremental, processing each data record only once (or few times); adaptive to distribution changes; and fast enough to accommodate high arrival rates.

We consider the problem of data-stream classification, introducing an online and incremental stream-classification ensemble algorithm, Streaming Random Forests, an extension of the Random Forests algorithm by Breiman, which is a standard classification algorithm. Our algorithm is designed to handle multi-class classification problems. It is able to deal with data streams having an evolving nature and a random arrival rate of training/test data records. The algorithm, in addition, automatically adjusts its parameters based on the data seen so far.

Experimental results on real and synthetic data demonstrate that the algorithm gives a successful behavior. Without losing classification accuracy, our algorithm is able to handle multi-class problems for which the underlying class boundaries drift, and handle the case when blocks of training records are not big enough to build/update the classification model.
الحمد لله الذي بنعمته تتر الصلاوات
Co-Authors


IN THE NAME OF ALLAH, THE MOST GRACIOUS, THE MOST MERCIFUL

As I reach the end of my Ph.D. journey, I first would like to express my countless Praise to ALLAH (GOD) for His graciousness and guidance, without which I could never have made it to the end.

I would also like to show my deep acknowledgments to all the people who helped me in accomplishing my goal. Special thanks to my supervisors, Prof. David B. Skillicorn and Prof. Patrick Martin for their great assistance, valuable advice, and experienced supervision.

I am so grateful to my parents, sisters, and brothers for their endless love and care, and continuous support. Mother, your sincere prayers for me were the spiritual lights that guided me through my course of study. Father, I will never forget your tremendous support and help at the time when I needed most. My sister Suha, thank you so much for persistently raising my self confidence and encouraging me to go for my goals. I am also grateful to my aunt, Amal, for always being there for me and lifting up my spirits.

I would like to express my thanks to all my friends who never gave up in me. I owe Bader Al-Manthari special thanks for his great assistance, precious advice, and
for standing by me throughout my journey. I would also like to thank Mohammed
Hussain and Asil Almonaies for their support, kindness, and continuous care.

A big thank you to Queen’s University for giving me this precious opportunity
to pursue my Ph.D. degree. Another big thank you to the members of the Database
Systems Laboratory, especially Wendy Powley, whose enthusiasm and inspiration
provided me a huge support.

Last but not least, I would like to thank Kuwait University, for their full financial
support.
Statement of Originality

I, Hanady Abdulsalam, certify that the work presented in this thesis is original unless otherwise noted.
## Contents

Abstract i  
Co-Authors iii  
Acknowledgments iv  
Statement of Originality vi  
Contents vii  
List of Tables x  
List of Figures xi  

### 1 Introduction  
#### 1.1 Motivation  
1.1.1 Data-stream classification 2  
1.1.2 Standard Random Forests 8  
#### 1.2 Thesis Statement 8  
#### 1.3 Contributions 9  
#### 1.4 Thesis Organization 11  

### 2 Background and Related work  
#### 2.1 Data Mining 12  
#### 2.2 Standard Data Classification 13  
2.2.1 Decision trees 15  
2.2.2 Classifier ensembles 17  
2.2.3 The standard Random Forests algorithm 19  
#### 2.3 Data-Stream Mining 21  
#### 2.4 Data-Stream Classification 23  
2.4.1 Decision trees for data streams 23
List of Tables

3.1 Definitions of parameters for Basic Streaming Random Forests . . . . 38
3.2 Implementation details of Basic Streaming Random Forests . . . . . 43
3.3 Classification errors and confusion matrices for Forest CoverType dataset 49

4.1 Definitions of parameters for Dynamic Streaming Random Forests . . 54
4.2 Implementation details of Dynamic Streaming Random Forests . . . . 64

5.1 Definitions of parameters for Autonomic Streaming Random Forests . 79
5.2 Implementation details of Autonomic Streaming Random Forests . . 90
# List of Figures

1.1 Email spam detection system model ........................................ 4
1.2 Possible scenarios for the embedding of labelled records in streams .. 5
2.1 Tabular structure of a data set ............................................. 14
2.2 Phases of standard classification systems .............................. 14
2.3 Example of a decision tree ................................................. 16
2.4 A classifier ensemble structure ............................................ 19
2.5 Effect of sliding window size to detect stream bursts or distribution changes ........................................... 29
2.6 Effect of sliding window size to define time delay of change detection .................................................. 29
3.1 Building a tree in Basic Streaming Random Forests .................. 41
3.2 Noise and classification error relation on synthetic data generated using the DataGen tool ........................................ 45
3.3 Classification errors for standard and streaming algorithms ........ 48
3.4 Per-record classification time ............................................ 50
4.1 Algorithm’s behavior in the case of new labelled data arrival ....... 61
4.2 Concept drift in the synthetic data ....................................... 65
4.3 Entropy versus number of records ....................................... 66
4.4 Classification error of the forest ......................................... 68
4.5 Classification error w.r.t. data noise .................................... 70
4.6 Values of $tree_{threshold}$ through execution ............................ 71
4.7 Values of $n_{min}$ through execution ..................................... 71
4.8 Values $tree_{min}$ through execution ..................................... 72
4.9 Number of replaced trees through execution ............................. 73
4.10 Time for building/updating the forest .................................. 74
5.1 $conf^r$ plots for $C = 2, 3, 5$ and $10$ .................................. 83
5.2 Algorithm’s behavior when a block of labelled records ends .......... 86
5.3 Milky Way top view with selected $RA$ values. Picture taken from http://www.msnbc.msn.com ........................................ 95
5.4 Milky Way side view with selected DEC values. Picture taken from http://www.astrosurf.com ................................................. 95
5.5 Values of $m_g$, $m_g'$, $mg_{threshold}$, and $mg'_{threshold}$ for synthetic data . . . 97
5.6 Values of $m_g$, $m_g'$, $mg_{threshold}$, and $mg'_{threshold}$ for PhotoObj dataset . 98
5.7 Classification errors for the evaluation sets (synthetic dataset) . . . . . . . . . 100
5.8 Classification errors for the evaluation sets (PhotoObj dataset) . . . 100
5.9 Confidence values for synthetic dataset ........................................... 102
5.10 Confidence values for PhotoObj dataset ......................................... 104
5.11 Classification errors of the forest for synthetic data ......................... 105
5.12 Classification errors of the forest for PhotoObj dataset ...................... 105
5.13 Classification errors of Random Forests for PhotoObj dataset .......... 107
5.14 Entropy values for SpecPhoto dataset ............................................ 108
5.15 Classification errors of Random Forests for SpecPhoto dataset .......... 109
A.1 Main flow of the algorithm ............................................................. 130
A.2 Build tree flow chart ................................................................. 131
A.3 Testing phase flow chart ............................................................. 132
A.4 Evaluation phase flow chart ......................................................... 133
A.5 Parameter derivation flow chart ..................................................... 133
Chapter 1

Introduction

Many recent applications such as telecommunication data management, financial applications, sensor data analysis, and web logs, deal with data streams, conceptually endless sequences of data records, often arriving at high rates. Analyzing or mining data streams raises several issues different from those of standard data-mining algorithms. Standard data-mining algorithms typically assume that records can be accessed multiple times. Data-stream mining algorithms, on the other hand, face more challenges since they must be able to extract all necessary information from records with only one, or perhaps a few, passes over the data.

Data-stream mining algorithms must, therefore, be online and incremental so that results can be generated at any time. They must also be able to process data records with high arrival rates. In addition, they should be adaptive, reflecting the current distribution of data in the stream, and able to approximate results that depend on observing the entire data, since an entire stream is never available.

The field of data-stream mining has attracted a great deal of attention, with research in, for example, detecting changes in data streams [25, 42–44, 63, 70, 73],
CHAPTER 1. INTRODUCTION

maintaining statistics of data streams [14, 21, 26, 52, 72], finding patterns in data streams [32, 41, 51, 67], data-stream prediction (classification/regression) [23, 24, 27, 29, 33, 37, 38, 45, 60, 61, 64, 71], and data-stream clustering [10, 11, 14, 15, 22, 32, 46, 62]. For more information on the subject, the readers are directed to our survey of stream-mining algorithms [7]. In this thesis, we consider the problem of classification for data-streams.

1.1 Motivation

1.1.1 Data-stream classification

Ordinary classification is usually considered in three phases. In the first phase, a model is built using data, called the training data, for which the property of interest (the class) is already known (labelled data). In the second phase, the model is used to predict the class of data (test data), for which the property of interest is known, but which the model has not previously seen. In the third phase, the model is deployed and used to predict the property of interest for (unlabelled data).

In stream classification, there is only a single stream of data, having labelled and unlabelled records occurring together in the stream. The training/test and deployment phases, therefore, interleave. Stream classification of unlabelled records could be required from the beginning of the stream, after some sufficiently long initial sequence of labelled records, or at specific moments in time or for a specific block of records selected by an external analyst.
Example Applications of Data-Stream Classification

In many stream applications, records can be labelled because the passage of time has revealed the appropriate class labels for each one. For example, a classification system for 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. The attribute-class relations of the emails arriving to the email server can evolve with time. Spam emails, for example, might have emails mostly related to banking services at a certain time, while they might be related to advertisements for university education at another time. This change might create different learning rules, and therefore, the system should be able to reflect the new rules. Figure 1.1 illustrates the email spam-detection system model.

Another example of stream classification is Pandora [1] and other streaming music stations. Each user labels an occasional song with ”I like” or ”I don’t like”, and the system tries to build a prediction model for all songs and play the songs that match the user’s taste.

Possible Scenarios for Labelled and Unlabelled Data Records

Depending on the labelled data distribution through the stream, we define some possible different scenarios shown in Figure 1.2. In Scenario 0, a block of labelled data records occur only in some initial segment of the data stream, and the size of the block is large enough to build a robust classifier. In this case, the only new issues
compared to standard classification 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 attribute-class relations of the data since no more labelled data are available for model update.

In Scenario 1, labelled records occur regularly in the stream, with enough of them occurring at a time to build/modify 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,
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/modify 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 Scenario 3, labelled records occur randomly 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 classification model, even when they appear close together in the stream. Scenario 3 can be considered...
the most general scenario, for which all other scenarios are special cases.

To understand the scenarios further, we relate them to the streaming music-stations example explained earlier. Newly registered users are often more enthusiastic, and so they recognize the importance of early rating. They, therefore, rate many songs so that the system is able to offer them the songs that match their taste. Their interest in rating, however, may reduce once they have rated a fair number of songs. This situation matches Scenario 0, where the labelled data block only appears at the beginning of the stream.

On the other hand, the rating of long-term users matches Scenario 3 in general, where users tend to rate the songs in a more random manner. As a special case, users usually tend to rate the songs played at the beginning of new sessions, whereas later on, they may involve themselves with other work while the music plays in the background. They may probably start rating again when the songs they are listening to become boring, or if they take a break. This special case matches Scenario 1 when the rating is done for a fair number of songs (larger blocks of labelled records), while it matches Scenario 2 when only few songs are rated periodically (smaller blocks of labelled records).

**Existing Stream-Classification Algorithms**

Decision trees have been widely used for data-stream classification [27, 30, 34, 35, 37, 39, 61, 64–66, 71]. These are typically based on Hoeffding bounds [36], 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.

Decision-tree ensembles have also been used for stream classification [24, 29, 33,
53, 57, 64, 69, 71]. Using classifier ensembles is a clever choice for data-streams because such algorithms are able to adapt to new stream properties easily by training new members of the ensemble and using them to replace old ones to give a smooth representation of concept changes.

Despite the problems already addressed in the field of data-stream classification, there are still limitations in the proposed algorithms. We list some of the most important ones here:

- Many of the current stream-classification algorithms are designed and/or tested on only two-class classification problems [24, 27, 37, 59, 64, 69, 71].

- Huge amounts of data are used in building the classifiers in current algorithms. The argument justifying this is that data streams are endless, and therefore, the amount of data used is not an issue. We, however, believe that algorithms should have better data utilization since this minimizes the building time of the classifier system, which in turn determines how soon it can be used for classification.

- The training time is directly proportional to the number of attributes of the data-stream, which therefore, results in algorithms with limited scalability.

- Many stream-classification algorithms have low classification accuracy and cannot handle changes in the distribution of the data caused by concept drift.

- Many stream-classification ensemble algorithms are based on some sort of batch learning for training their base classifiers [24, 29, 33, 53, 57, 64, 69, 71], which contradicts the requirement for a fast and incremental learning algorithm for stream mining.
1.1.2 Standard Random Forests

The Random Forests algorithm by Breiman [19] is an ordinary classification ensemble algorithm that depends on sampling records in two directions (rows and columns) for generating its training subsets. It grows multiple binary decision trees, each from a bootstrap sample [18] of the data. The splitting decision at each node selects the best attribute on which to split from a small set of randomly chosen attributes. Hence, the algorithm is scalable and can accommodate a dataset with a large number of attributes. Classification in the Random Forests algorithm 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. It was shown to build models with good accuracy when tested on different data sets.

1.2 Thesis Statement

We consider the problem of stream classification and we aim to overcome the deficiencies of existing stream-classification algorithms mentioned earlier. We, therefore, present a novel algorithm called Streaming Random Forests, which is an online and incremental stream-classification ensemble algorithm that extends Breiman’s Random Forests algorithm [19]. The Streaming Random Forests algorithm improves on other stream-classification algorithms by combining techniques used to build streaming decision trees with the technique of randomly selecting few attributes to participate in building each node used by Random Forests. To the best of our knowledge, our algorithm is the first decision-tree based streaming ensemble that builds its classifiers (trees) incrementally. Other algorithms depend on batch learning, so that they
need to buffer the training data records that are used to train each classifier.

Our algorithm handles only numerical or ordinal attributes for which the maximum and minimum values of each attribute are known. It assumes that the data records are approximately uniformly distributed across all classes, so that each class has enough data records to learn from. It also assumes that any random snapshot of the data can represent the true distribution of data records across classes.

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 Forests algorithm is a realistic algorithm that is designed to deal with streams of the kinds described in Scenarios 0 through 3. It handles multi-class classification problems, in contrast to many stream classification techniques that have been designed and/or tested only on two-class problems. It is an adaptive algorithm that is able to handle changes in the underlying relations of the data. It is also able to adjust its own parameters depending on the properties of the data records seen so far.

1.3 Contributions

We introduce an incremental stream classification algorithm with the ability to handle multi-class classification problems and successfully deal with changes in underlying relations of the data without losing performance with regard to classification accuracy. We believe that a realistic stream classification algorithm must be able to handle streams of Scenario 3 in Figure 1.2. We, therefore, design our algorithm with extensions so that it is able to handle streams of Scenarios 0 through 3.

Each scenario presents a new problem for the design of the algorithm beyond
the previous scenario. Scenario 0 only requires a streaming version of the standard
Random Forests so that a forest can be built from only one pass. Scenario 1 requires
the algorithm to be able to handle changes of the underlying relations of the data
and so always to reflect the current structure of the data relations. Scenarios 2 and
3 require the algorithm to be clever enough to decide when the forest needs to be
adapted and when it is accurate enough to be deployed.

The contributions of this work can be summarized as follows:

• **We extend the standard Random Forests algorithm to present Basic Streaming Random Forests** [8], an incremental stream-classification algorithm that handles Scenario 0 with a classification accuracy comparable to (or better than) standard classification algorithms. This basic version has all the necessary ideas to transform the standard Random Forests algorithm to a streaming version. We implement our algorithm and evaluate it on real and synthetic data. We base all our subsequent extensions of the algorithm on this basic version.

• **We extend the Basic Streaming Random Forests algorithm to introduce Dynamic Streaming Random Forests** [9], a self-adjusting stream-classification algorithm that handles Scenario 1 using an entropy-based change-detection technique. This version of the algorithm includes provisions to handle evolving streams. One major contribution for this version is the entropy-based change-detection technique, which we define based on ideas proposed by Vorburger and Bernstein [63]. We evaluate our algorithm on a synthetic evolving data having different changes in the underlying structure of relations.
• We further extend the Dynamic Streaming Random Forests algorithm to propose Autonomic Streaming Random Forests, a realistic stream-classification algorithm that handles Scenario 3. The key feature of this version is that it is able to take a decision of whether the current model is ready for deployment or not, when the number of labelled data records is not enough to completely build/update the model. A significant contribution of this version is, therefore, the definition of a threshold value on which the decision to deploy the current model is based. We evaluate our algorithm on synthetic and real data having blocks of labelled data records with random sizes that appear randomly in the stream.

1.4 Thesis Organization

The remainder of the thesis is organized as follows: Chapter 2 describes some background and related work. Chapter 3 defines the Basic Streaming Random Forests algorithm, which is the foundation of our approach to solving the stream-classification problem. Chapter 4 explains the Dynamic Streaming Random Forests algorithm, which is the extended version for handling Scenario 1. Chapter 5 describes the Autonomic Streaming Random Forests algorithm, which is the realistic version for handling Scenario 3. Finally, Chapter 6 summarizes the contributions, draws conclusions, and lists some future work opportunities.
Chapter 2

Background and Related work

2.1 Data Mining

Data mining is the process of analyzing large collections of data by extracting embedded and useful information from them. These techniques can be divided into two main categories, namely, predictive techniques, and descriptive techniques. The goal of predictive techniques is to build a model from a given data-set with a known property of interest or target (labelled data), such that the model relates the features of the data to this target property. The model then estimates the target property for new data with an unknown target (unlabelled data). If the target is a discrete variable (class label) for each record, then prediction is called classification. If the target is a numerical value, then prediction is called regression. For example, we may want to predict whether each frame captured by a closed-circuit television camera contains a human face or not (classification), or we may want to predict the wind velocity in each frame based on the position of objects such as flags (regression). Predictive techniques are also called supervised techniques since they need supervision to build
their models by learning from the labelled data.

The goal of descriptive techniques, on the other hand, is to understand and discover previously unknown implicit relations or properties of unlabelled data. They are called unsupervised techniques since they do not learn from historical data with known labels, hence, they perform without any supervision. Some common descriptive techniques include clustering, summarization, and change-detecting techniques. Clustering is the process of gathering input data into a number of groups or clusters depending on similarities between data records. Summarization techniques are involved in representing the data in a compact form, such as histograms, averages, and variances. Change-detection techniques find the times at which a significant change in the underlying relations has occurred.

A common structure of the sets used in the field of data-mining follows a tabular form of size $n \times m$, where $n$ is the number of records (rows) and $m$ is the number of attributes or features (columns). Each record has an associated measure for each attribute. Attributes can represent primitive types of data such as numerical, ordinal, and categorical values, or they can represent more complicated data structures such as arrays and lists. Labelled data include one attribute that represents the target value for each record (class label or a numerical value). Figure 2.1 represents the dataset tabular structure.

### 2.2 Standard Data Classification

A standard classification system has normally three different phases: 1) the training phase, during which the model is built using labelled data; 2) the testing phase, during which the model is tested by measuring its classification accuracy on withheld labelled
data; and 3) the deployment phase during which the model is used to predict the class of unlabelled data. The three phases are carried out in sequence. See Figure 2.2 for the standard classification phases.

Figure 2.1: Tabular structure of a data set

Figure 2.2: Phases of standard classification systems
2.2.1 Decision trees

A popular technique for standard classification is the Decision Tree. A decision tree is a tree data structure in which internal nodes contain tests on attribute values, and leaves have assigned class labels. Attributes can be numerical, ordinal, or categorical. The nodes that represent numerical or ordinal attributes are associated with tests that usually employ relational operators on the range the attributes’ values. Nodes that represent categorical attributes are associated with tests that usually employ (in)equality on the categories of the attributes.

Decision trees are built from training data by determining the attribute whose value is most informative, based on one of a number of criteria such as information gain (ID3 or C4.5 learning algorithms) and gain ratio [28,40], or Gini index importance [20], and using a test based on that attribute as the root of the tree. Tests at the next level of the tree are determined in a similar way using the records selected by the initial test and the remaining attributes. Each non-leaf node, therefore, has a number of outgoing branches equal to the number of possible outcomes of the test assigned to this node. Records arriving at any non-leaf node are therefore partitioned among the node’s branches based on the result of the node’s test.

In some cases, attributes can be selected more than once for formulating tests on different nodes. This is only possible for numerical or ordinal attributes. A test for a node located at a high level of the tree for a numerical or an ordinal attribute can, for example, be a relational operator on one value that divides the range of the attribute values into half. This attribute can be selected again at a node located at a lower level of the tree to divide the upper half of the values range into two further halves.

When the set of training records that remain at the end of a branch is entirely or
almost entirely from a single class, the branch is not extended further, and the leaf is labeled with this majority class label. An example of a simple decision tree is shown in Figure 2.3.

![Figure 2.3: Example of a decision tree](image)

Once the decision tree is built, its potential classification accuracy is calculated using test data. Each test record is tested against the test at the root of the tree, then passed down to one of the branches depending on the outcome of the test. This process is repeated until the record reaches a leaf. The class label of that leaf is used as the predicted classification for the record. The classification error is then calculated by finding the ratio of the number of records that have an incorrect class assigned to them by the tree to the total number of test records.

One way to improve the classification accuracy of decision trees is **pruning** [48]. Pruning is the process of modifying the decision-tree structure by replacing one or more of the subtrees with leaves or other subtrees using some criteria, which insures that the new overall decision-tree structure has a better classification accuracy. One
popular method of pruning is the *post-pruning* method [20, 54], that decides which parts of the tree need to be pruned, if any, after the whole tree is built. In order to make this decision, extra extensive calculations are added to the tree building algorithm.

**The Gini-index measure for building decision trees**

The Gini index [20] is one example of a measure of impurity among the data records that are considered at a node of the tree. It measures the divergence among the distributions of the classes of the data records. For a dataset $D$ that contains $n$ records from $C$ classes,

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

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 $Gini(D) - Gini(D)_{split}$.

### 2.2.2 Classifier ensembles

A well-known method of building classification systems is to build multiple classifiers, each from a subset of the original training set, such that the final classification decision is aggregated from all classifiers’ decisions. This method is called the *classifier ensemble* method [47, 55]. For example, five classifiers could be built independently using five different subsets of the original training set. These five classifiers would
produce five predictions of the class label for each new record, and the class with a plurality of votes would be the prediction of the entire ensemble. It is also possible to extend this simple voting scheme so that each individual classifier prediction is given a weight, perhaps based on its test accuracy. The overall prediction becomes the plurality of the weighted votes. Another possible extension is to use only the prediction of the most-accurate classifiers out of a number of total classifiers in the ensemble, based on some test criteria.

Classifiers in an ensemble can all have the same type, or they can be of different types. For example, an ensemble with three classifiers can consist of three decision trees, or it can consist of a decision tree, a neural network \[40\], and a Bayesian network \[28\]. Both kinds of ensembles are known to perform better than single classifiers. The variance between classifiers is reduced in the case of classifiers of the same type, and the bias between classifiers is reduced for ensembles with different types of classifiers. The classification models of ensembles for both kinds are, therefore, more representative of the data than a single classifier. In other words, having multiple strong classifiers each built from a different sample of the dataset leads to a final classification decision with higher accuracy than a single classifier. A general structure of classifier ensembles is shown in Figure 2.4.

Generating the datasets used for training the classifiers in an ensemble can be done by different methods such as bootstrap sampling (bagging) \[18\], and boosting \[31\]. Suppose that a dataset contains \(n\) records, each with \(m\) attributes. Bootstrap sampling or bagging generates the datasets each of size \(n\) by randomly sampling the records with replacement. Hence the training dataset for each tree contains multiple copies of some of the original records.
Boosting maintains weights for records in the training set, such that these weights are updated after each classifier is trained according to the difficulty of classifying the current set of records. The weights are then used to derive the new sampling for the dataset.

2.2.3 The standard Random Forests algorithm

The Random Forests algorithm is a classification technique developed by Breiman [19]. Superficially, Random Forests are similar to ensembles of binary decision trees. A set of decision trees are grown, each from a bootstrap sample of size \( n \). Random selection with replacement for bootstrap sampling 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
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 can be used to formulate the test at this node. By contrast, in the Random Forests algorithm, the set of attributes considered at each internal node is a different randomly-chosen subset of the attributes, of size $M \ll m$. In addition, the Random Forests algorithm grows the trees to the maximum possible size, that is no pruning is performed as in standard decision-tree algorithms.

The Random Forests algorithm has its own way of dealing with categorical attributes. The standard structure of a node that is associated with a categorical attribute is to have a number of outgoing branches equal to the number of categories of the attribute. Tests on categorical attributes with more than two categories in Random Forests, however, can not have a number of branches equal to the number of categories since the trees are binary trees, and so only have two outgoing branches for each node. To handle this situation, tests are formulated as (in)equalities that separate the possible categories of each attribute into two groups. Categorical attributes that have more than two categories can, therefore, be selected more than once for different nodes, such that the categories are partitioned into two smaller groups each time the attribute is selected at a node.

When the model is built, the forest is deployed as if it was a typical ensemble classifier, that is the classification for each new record is the plurality of the votes from each of the trees. As with any classifier ensemble algorithm, the classification error of the Random Forests algorithm is affected mainly by two things:

- The correlation among the trees: the smaller the correlation among the trees
the more variance canceling takes place as the trees vote, and therefore the lower the expected error rate.

- The strength of each individual tree: the more accurate each tree is, the better its individual vote, and therefore the lower 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.

The Random Forests algorithm defines a measure of the strength of the ensemble of classifiers, called the *margin function*. The margin function measures how far the average number of correct votes for a specific record is from the average number of votes of the next candidate class for the same record. Given a Random Forest having trees $tree_1$, $tree_2$, ..., $tree_U$, the margin function is defined as

$$mg(X,Y) = \text{avg}_U I(tree_u(X) = Y) - \max_{j \neq Y} \text{avg}_U I(tree_u(X) = j)$$

where $I(.)$ is the indicator function, $X$ is a labelled record having a class $Y$, and $tree_u(X)$ is the vote class of $tree_u$ for the record $X$.

### 2.3 Data-Stream Mining

Many recent applications such as internet traffic monitoring, telecommunications billing, and sales tracking produce a huge amount of data to be monitored. It is not practical to store all of the data indefinitely. The data is instead presented as continuous streams. We define a data stream to be an endless, real-time, and ordered sequence of records. Systems that manage such streams have been called Data-Stream Management Systems (DSMSs).
A number of example DSMSs appear in the literature. Some are general DSMSs; for example STREAM [12, 13], the Stanford data stream management system, and Aurora [6]. Other systems are for special applications; for example, COUGAR [17] is a sensor system developed at Cornell University, used in sensor networks for monitoring and managing data; and the Tribeca network monitoring system [58] is a DSMS designed to support network traffic analysis.

One important application of DSMSs is data-stream mining. Data-stream mining is the process of applying well-known static data-mining techniques to data streams in order to extract useful and interesting information from them continuously in an online fashion. Data-stream mining poses new challenges such as understanding the trade-offs between accuracy and limited access to the data records, developing new algorithms that avoid multiple passes over the data while still producing similar results, and understanding the relationship between the amount of data seen and accuracy. Performance issues are of critical importance since they determine how much processing can be applied per data object.

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).

- Algorithms must be fast enough to handle the rate at which new data arrives (which means effectively amortized $O(1)$ time for both learning and prediction/clustering).

- Algorithms should be adaptable to changes in the distribution of values in the underlying stream (concept drift problem), since they may run for a long time.
Results that depend on observing an entire dataset can never be computed exactly, so the results of stream mining must necessarily be an approximation.

2.4 Data-Stream Classification

As mentioned in Section 1.1.1, in stream classification there is no clear separation between phases since we assume that there is only one stream of data containing labelled and unlabelled data records. According to this assumption, streams used in classification systems can follow any of the scenarios described in Figure 1.2.

2.4.1 Decision trees for data streams

Many data-stream classification techniques are based on decision trees with some variations to satisfy the requirements of streams [24, 27, 29, 30, 33–35, 37, 39, 61, 64–66, 71]. The biggest problem in extending decision trees to data streams is that the measures of attribute importance used to determine the best choice of attributes require counts or probabilities computed over all of the training data. Clearly, this is not possible when the data is a stream. One solution, proposed by Domingos and Hulten [27], is to use the Hoeffding bound (also called Chernoff bound) [36] to estimate when the number of records accumulated at a node is ‘enough’ for a robust decision.

The Hoeffding bound

The Hoeffding bound [36] states that, given a random variable $r$ in the range $L$, and $n$ independent observations of $r$ having mean value $\bar{r}$, the true mean of $r$ is at least
with probability $1 - \delta$, where $\delta$ is a user-defined threshold probability. 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 bound is used in the 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 $\Delta G = G_{\text{highest}} - G_{\text{second \_highest}}$. The algorithm then recalculates $G$ for all attributes as each new record arrives, and updates $\Delta G$ continuously until it satisfies a stopping condition, $\Delta G > \epsilon$. At this point, the true value of the largest $G$ is within $\epsilon$ of the approximated $G$ with a probability of $1 - \delta$, and therefore the attribute with the highest $G$ is the best choice for splitting at the current node with a confidence of $1 - \delta$.

**Hoeffding trees**

A streaming decision tree based on the Hoeffding bound (Hoeffding tree) consists of three types of nodes while it is under construction: internal nodes, which contain inequalities on attributes; frontier nodes, which are 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’ split attribute, or perhaps to not split this node further and instead 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 the Hoeffding bound test indicates that the frontier node has accumulated ‘enough’ records so 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.

Stream classification decision-tree based algorithms

Domingos and Hulten [27] propose the VFDT (Very Fast Decision Tree) algorithm for classifying high-speed streaming data. VFDT is the base for many algorithms that use decision trees. It is capable of building decision trees that require constant memory and constant per-record construction time. VFDT follows the basic steps of building a Hoeffding tree described above. To improve the classification accuracy, VFDT uses a standard decision-tree algorithm to create an initial tree offline, using a small number of records. This provides a reasonable initial approximation to the online classification tree.

VFDT does not consider the concept drift problem. An improvement for VFDT was, therefore, proposed by Hulten et al. [37], called the Concept-adapting Very Fast Decision Tree (CVFDT) algorithm, which addresses concept drift while maintaining similar efficiency and speed to VFDT. The main idea behind CVFDT is to grow alternative subtrees for internal nodes. Whenever there is an internal subtree that poorly reflects the current concept, CVFDT replaces this node with the alternative subtree that has better performance. Depending on the available memory, CVFDT defines a maximum limit on the total number of alternative subtrees that can exist at the same time. If there are alternative subtrees that are not making any progress,
CVFDT prunes them to save memory. Other improvements on the VFDT approach involve improving the split-point search algorithms, such as using binary search trees for finding more accurate split points for each node [65, 66].

Decision-tree ensembles are widely used in the field of stream classification since classification results that are based on ensembles are more accurate and robust than results of single classifiers. Tree ensembles are decision trees built using different parts of the data stream. These trees are deployed as an ensemble by using each of them to classify unlabelled records and then their votes are combined using different methods [47]. A simple and popular method of combining classifier votes is uniform voting, which simply considers the class with the maximum number of votes as the final classification decision.

Algorithms of tree ensembles usually build each of their base classifiers from a buffered block of training data records [24, 29, 33, 53, 57, 64, 69, 71]. They use standard learning algorithms, such as the C4.5 algorithm, to train their trees. This leads to algorithms that are not totally incremental and so lack efficiency.

Sun et al. [59], propose an algorithm for stream classification based on multi-classifiers. They state that the learning algorithm of the base classifiers is incremental. It is not clear, however, what learning method they use. Their experiments are conducted on the Weka platform [68], a freeware data mining tool, which requires the whole dataset to be available to learn a classifier.

2.5 Concept-Drift Detection in Data Streams

Concept drift in data-streams is the result of changes that might occur in the distribution of records among classes. Detecting changes in data streams has many
applications in areas such as network monitoring, traffic management, intrusion detection, and system analysis. In addition, change detection can be considered as a precursor to some stream applications that build models from the data. Data-stream change-detection algorithms can be applied to detect long-term changes in the distribution of the underlying process that is generating the data [25, 42, 43, 63], or stream bursts, which are sudden large and unusual changes in a data stream [70, 73]. Stream bursts can be of interest for applications such as stock markets.

To be able to detect a change in a data stream, some summary information for data seen so far must be stored and compared with calculations from new data. The comparison is performed using two approaches. In the first approach, the system keeps track of two windows of data: the current window of data, and a reference window that captures the most recent underlying distribution of the stream. The system compares the calculations obtained from the two windows and records a change in the distribution whenever the two calculated values are significantly different. If a change is detected, the current window is made the reference window to reflect the new data distribution, and the current window fills with new data records.

In the second approach, the system uses the old data distribution to predict the new incoming data distribution, and then compares the predicted distribution with the actual one. If the prediction error is significant, then the distribution of the stream has changed.

In both approaches, the size of the window defines the smoothness/sharpness of the expected changes and the time delay between the moment at which the change has actually occurred and the moment at which the system detects the change. A system with a small window is designed to detect sudden stream bursts, whereas a
system with a large window is designed to detect long-lasting distribution changes. Furthermore, small windows are able to quickly report concept changes, whereas very large windows can have a delay, perhaps significant, in reporting changes of the stream.

Figures 2.5 and 2.6 show the effect of the sliding window size in detecting changes type (sudden bursts or distribution changes), and the time delay of change detection respectively. The figures considers sliding windows with different sizes; a small window of size 100 records, and a large window of size 500 records.

It can be shown from Figure 2.5 that a sudden burst appears at record 1450, where the concept parameter changes from 0.1 to 0.4. The small window detects the burst, whereas the larger window might consider it as only noise.

The figure also shows an example of gradual concept drift. The concept parameter is equal to 0.1 at record 3400 and it gradually increases until it reaches 0.3 at record 3900. This type of concept drift is better detected by the large window, since the entire increase in concept parameter can be captured by one window, whereas the small window might not be able to notice the gradual change.

Figure 2.6 shows a concept drift at point 2500. From the figure, the large window detects the drift after receiving 500 records generated from the new concept, whereas the small window detects the drift after receiving 100 records only. Hence, the delay of detecting the drift using the large window is 5 times greater than using the small window.
Figure 2.5: Effect of sliding window size to detect stream bursts or distribution changes

Figure 2.6: Effect of sliding window size to define time delay of change detection
2.5.1 Detecting changes using entropy

Shannon’s entropy [56] is a measure of the disorder or randomness associated with a random variable. It is defined as:

\[ H(x) = -\sum_i p_i \log_2(p_i), \]

where \( x \) is a discrete random variable, and \( p_i \) is the occurrence probability of \( x_i \). To calculate the entropy for a data set with a certain distribution, the following equation is used:

\[ H(x) = -\sum_x P(x) \log_2(P(x)), \]

where \( P(x) \) is the probability mass function of \( x \). In order to use entropy in the context of detecting concept changes in data streams, the two-window paradigm [25, 63] is usually used. Entropy algorithms for detecting concept changes in data streams compare the entropies of the reference window and the current window at time \( t \). If the entropies are different according to some defined rules then a change has occurred. The rules are defined depending on the strength of the expected change. The stricter the rule, the sharper the expected change.

2.6 Limitations of Stream-Classification Algorithms

The decision-tree based stream-classification algorithms that appear in the literature have some disadvantages. The area is, therefore, still rich in open research problems. We consider some of the major problems in the area that require further study to improve on the overall performance of the algorithms.
2.6.1 Multi-class classification problem

Many of the stream classification algorithms that were mentioned earlier are designed and/or tested for two-class classification problems [24,27,30,33–35,37,39,59,64,69,71]. Obviously, two-class problems are simpler for an algorithm to handle than multi-class problems for a number of reasons:

- **Learning rules:**
  The learning rules for the data concepts are defined to separate the input data into a number of groups that is equal to the number of classes. The complexity of the boundaries between classes of two-class problems is usually less than it for multi-class problems for two main reasons: 1) more time will be needed by the algorithm to learn the properties of the classes, and 2) records that are close to boundaries for two classes are easier to handle that for multi classes because there are more choices to classify the records laying in these areas for multi classes, and so the algorithm has to be able to distinguish between them. Hence, the larger is the number of classes, the more complicated is the structure of the boundaries expected to be, and therefore, the harder for a classifier to learn.

- **Required number of records:**
  Assuming similar level of complexities for the boundaries between classes of a two-class problem and a multi-class problem, the total number of data records required for a classifier to learn the characteristics of the classes for the two-class problem is fewer than for the multi-class problem. To illustrate this point, we give the following example. Consider two datasets $D_1$ and $D_2$, representing a two-class problem and $C$-class problem, respectively. For simplicity, we assume
that the classes are well separated in each of the datasets. Each dataset consists of \( n \) records, with the records distributed uniformly among the classes. \( D_1 \), therefore, has around \( n/2 \) records for each class, whereas \( D_2 \) has around \( n/C \) records belonging to each class. Clearly, as \( C \) increases, the number of records that lie in each class decreases and hence might not be enough for a classifier to learn the characteristics of the class. Assume the \( n \) records of the two-class problem are enough for a classifier to learn both classes’ characteristics, then the required total number of records for a \( C \)-class problem is around \( \frac{n}{2}C \), for \( C > 2 \).

- **Dealing with noisy data:**

  In the case of classifying noisy datasets, the classification accuracies for two-class problems are usually higher than for multi-class problems. This is because whenever the classifier is not capable of classifying a record correctly, it still has a probability of 0.5 of getting it correct for two-class problems, whereas it only has a probability of \( \frac{1}{C} \) of getting it correct for \( C \)-class problems, assuming the worst-case scenario where the voting is uniformly random among the classes.

### 2.6.2 Number of training data records

Since having a data stream means having an infinite number of records flowing into the system, using a large number of data records to build a stream classifier is not a major issue by itself. It is, however, time consuming. Several stream-classification algorithms do not take this aspect into consideration, relying on the fact that the data stream is endless and, therefore, records are always available.

The VFDT algorithm by Domingos and Hulten [27], and its extensions CVFDT
[37] and VFDTc [35], in addition to the algorithm by Jin and Agrawal [39], have only been tested on two-class datasets, and yet they use around $5 \times 10^6$ records on average to build their classification models. Note that they build only one decision tree using this number of data records. Applying these algorithms on multi-class problems with $C$ classes requires the use of around $\frac{5 \times 10^6}{2} C$ records (according to Section 2.6.1) on average. This adds significant time for building the model.

Tree-ensemble algorithms have better utilization of the data records. They build an ensemble of classifiers using the same number, or typically fewer, records than single classifiers. Some examples of stream-classification ensembles were mentioned earlier in Section 2.4.1. Although they use a reasonable amount of data, they have a major limitation, namely they are only designed and/or tested for two-class problems.

### 2.6.3 Training time

The training time for a decision-tree-based stream-classification system is mainly affected by the number of records used to train the classifier and the algorithm on which the split decision for each node is based. We already mentioned the problem of using a huge number of training records in the previous section. We now consider the splitting algorithms.

Two common algorithms to measure how informative an attribute is are information gain [28, 40] and Gini index importance [20]. Both algorithms need to search each attribute and all its available values to decide on the best splitting point for a node. The training time is, hence, directly proportional to the number of attributes. Clearly, this restricts the scalability of the algorithms. New techniques are required for the case of data sets with a large number of attributes.
2.6.4 Learning algorithms for base classifiers of ensembles

Many existing tree-ensemble algorithms are either based and/or tested on standard learning algorithms [24, 29, 33, 53, 57, 59, 64, 69, 71]. Algorithms are therefore required to buffer data records until a defined number of records is reached, and then, run the standard algorithm on the buffered data. This results in non-incremental algorithms, which need extra time to train their base classifiers, and so might not be able to cope with the high arrival rate of records in data streams.
Chapter 3

Basic Streaming Random Forests

We extend the standard Random Forest algorithm by accommodating special provisions so that it can be applied to streaming data. The difficulty in doing this is that the Random Forests 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 sample 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. This is not an issue for streaming algorithms though, since data is assumed to be infinitely flowing into the system. Streaming algorithms should however be able to use as few records as possible so that the time they use in building or updating their models is not significant. It would be conceivable to use a separate batch of labelled data to make the decision about each
internal node, but this would increase the time before robust classifications could be made for unlabelled records. Instead, we adapt ideas from streaming decision tree algorithms [27, 37] 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.

3.1 Algorithm Details

We design the Basic Streaming Random Forests algorithm [8], an extension of the standard Random Forests algorithm with features to meet the requirements of stream-classification algorithms. This basic algorithm handles Scenario 0. It does not take the concept-drift problem into consideration since it assumes that there are no labelled records after the model building phase is over. It handles only numerical or ordinal attributes for which the maximum and minimum values of each attribute are known. It sequentially builds a set of binary trees, just like in the standard Random Forest algorithm. The algorithm takes the required number of trees as a parameter and builds each tree from a different block of labelled data.

3.1.1 Sampling for labelled data

The standard Random Forest algorithm relies on samples chosen randomly 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 because data records are not stored, so we must consider whether this affects the properties of the new algorithm. Although our algorithm accesses
the records on an one-by-one basis as they arrive, it is considered random sampling without replacement since we assume that the order of records does not matter, and so the records are arriving randomly.

For an infinite data set 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 random variables \( x_i \) and \( x_j \), where \( i \neq j \), sampled randomly without replacement into a buffer of data, depends on the dataset size: 

\[
\text{cov}(x_i, x_j) = -\frac{\sigma^2}{n-1},
\]

where \( \sigma^2 \) 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.

To safely apply sampling without replacement to our algorithm, we consider the fact that during the very early stages of tree construction, only a small number of records have been seen and so the sampling without replacement might have a noticeable effect on algorithm performance, perhaps distorting the behavior of the first few trees. We, therefore performed experiments simulating sampling with replacement. We did this by running our Basic Streaming Random Forests algorithm on a dataset with a random selection of 33% of the labelled data records and applying them as training examples twice (by double incrementing the counts). The average covariance of the observed values of records’ attributes is continuously calculated until it values became small. We chose to use a percentage of 33% for selecting the records that doubly increment the counters, because the bootstrap sampling (explained previously in Chapter 2), roughly samples one third of data records twice and one third only once, while leaving one third of the data not sampled at all. We run our experiments
with and without applying the simulated method of sampling with replacement, mea-
suring the classification accuracy of the model for both cases. The results show that,
in practice, sampling without replacement does not decrease the accuracy of the con-
structed forest since we got approximately similar accuracies for sampling with and
without replacement.

Part of the motivation for sampling with replacement for standard-classification
algorithms is also to increase the effective size of the training and test sets. Clearly,
this is not necessary for streams since the data is infinite.

### 3.1.2 Tree building

The parameter *tree window* is a stopping criterion for the algorithm to stop building
a tree and begin the construction of the next tree. The algorithm continuously builds
trees, each from a block of labelled records of size *tree window*, until the required
number of trees have been built. The parameter, *tree window*, for this version of
the algorithm is a parameter defined by the analyst. Table 3.1 defines the newly
introduced parameters for the Basic Streaming Random Forests algorithm.

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>tree window</em></td>
<td>number of records required to build a tree</td>
</tr>
<tr>
<td><em>class counts</em></td>
<td>number of records per class for each node</td>
</tr>
</tbody>
</table>
| \( n_{\text{min}} \) | number of records that a node must process before applying the Hoeffd-
                          ing and Gini tests                                      |
| *node window*    | maximum number of records allowed for building a node                    |
| \( \alpha \)     | a constant that is defined empirically to shrink the *node window*       |

As in the standard Random Forests algorithm, the Basic Streaming Random
Forests algorithm only considers a different randomly selected set of $M \ll m$ attributes, where $m$ is the total number of attributes for building each node. Upon the arrival of a new labelled record, the record is routed down the tree under construction, 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. Class counts represent the distribution of each attribute’s values across the classes. Since the algorithm deals with ordinal or numerical attributes, computing the counts for all possible values of each attribute is impractical (or perhaps impossible). Attributes are therefore 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, $n_{\text{min}}$, is used to decide how often to check whether a frontier node should be considered for transformation. Whenever a node accumulates $n_{\text{min}}$ 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. 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 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 value of the node window 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:

$$\text{node window} = \alpha\left(\frac{\text{tree window}}{2^{\text{node level}}}\right)$$

so that the node window shrinks exponentially with the depth in the tree. The value of $\alpha$ is determined empirically based on running a number of experiments with different values of it and choosing the value that gives a more accurate classification.

### 3.1.3 Tree pruning

A limited form of pruning is necessary because a node may generate two descendant frontier nodes that do not then see enough subsequent records to be considered for splitting themselves. If two sibling nodes fail 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.

Since the purity is only calculated when too few records have arrived at the two sibling leaf nodes, the calculations of purity are based on the counters of the parent node that represent the records that would have flowed down to the leaves. If the purities of both siblings are less than $1/C$, where $C$ is the number of classes, then both nodes are pruned and their parent node is changed to 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 3.1.
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 arrived 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 majority class to it
        end while
    end

Figure 3.1: Building a tree in Basic Streaming Random Forests
3.2 Implementation and Experimental Settings

In implementing the Basic Streaming Random Forest algorithm, we assume our stream follows Scenario 0 in Figure 1.2. The labelled records at the beginning of the stream are used as training records, with later labelled records of the same block used as test records. 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 chosen as suggested by Breiman to be $M = \text{int}(\log_2 m + 1)$.

3.2.1 Testing criteria

The criteria we use to evaluate the performance of the Basic Streaming Random Forests algorithm are the classification accuracy and the per-record classification time. We explain these two criteria further below.

- *Classification accuracy.* We measure the classification accuracy by calculating the classification error percentage of the algorithm during the testing phase. The smaller the classification error, the higher the accuracy of the algorithm. We evaluate the performance of the algorithm by comparing its classification error 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. The classification accuracy should be close to what is expected (according to previous knowledge of the data or by comparison with other classification techniques).
• *Per-record classification time*. The per-record classification time is the rate-limiting step of our algorithm because each new record must be evaluated by all of the trees. We therefore include this metric in our performance evaluation. The per-record classification time should be small enough that the algorithm is able to handle streams arriving at fast rates. We measure the classification time per unlabelled record for different forest sizes to estimate the flow rate that the algorithm can handle.

### 3.2.2 Testing platform

The experiments of all extensions of the algorithm are conducted on a Pentium 4 machine. The machine has a 3.2 GHz processor and 512 MB RAM.

We implement our algorithm based on the open-source Random Forest Fortran FORTRAN77 code by Breiman and Cutler [2]. A summary of the work done to extend the original Random Forests algorithm to the Basic Streaming Random Forests algorithm is shown in Table 3.2

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Random Forests</th>
<th>Basic Streaming Random Forests</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of code lines</td>
<td>1600 lines</td>
<td>1700 lines</td>
</tr>
<tr>
<td>used:</td>
<td>300 lines ≡ 18.75% of original code</td>
<td></td>
</tr>
<tr>
<td>altered:</td>
<td>400 lines ≡ 25% of original code</td>
<td></td>
</tr>
<tr>
<td>deleted:</td>
<td>900 lines ≡ 56.25% of original code</td>
<td></td>
</tr>
<tr>
<td>new added code lines:</td>
<td>1000 lines</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No. of main subroutines</th>
<th>9 subroutines</th>
<th>5 subroutines</th>
</tr>
</thead>
<tbody>
<tr>
<td>used:</td>
<td>0 lines ≡ 0% of original code</td>
<td></td>
</tr>
<tr>
<td>altered:</td>
<td>2 lines ≡ 22.22% of original code</td>
<td></td>
</tr>
<tr>
<td>deleted:</td>
<td>7 lines ≡ 77.78% of original code</td>
<td></td>
</tr>
<tr>
<td>new subroutines:</td>
<td>3 subroutines</td>
<td></td>
</tr>
</tbody>
</table>
3.2.3 Datasets

We base our testing on both synthetic and real data sets. All the synthetic datasets we use in our testing are generated using the DataGen data-generation tool [50]. We also base our testing for the basic Streaming Random Forests on a real dataset, namely the Forest CoverType dataset [3], from the UCI repository.

**Synthetic data**

We studied the data generated from the DataGen tool to discover the relation between the noise that is explicitly defined to generate a dataset, and the classification error expected from a classification algorithm running on the same dataset. To do this, we used the DataGen tool to generate four small sets of data records with varying noise levels of 0%, 1%, 2%, and 5% respectively. Each dataset has 3 numerical attributes, 2 classes and contains 100 records. The random seed for generating the data was fixed to a constant value so the effects of changing the noise rate can be compared. We plot the datasets in three dimensions. Each dimension represents the values of one attribute. Figure 3.2 demonstrates the four test cases we conducted that clarify how the classification error of the four datasets depend on the percentage of noise.

The plots show that the data is separated into two classes, Class 1 represented by red asterisks, and Class 2 represented by blue circles. For the dataset with 0% noise, it is clear that the records are grouped into the two classes with no outliers around, which means the classification error for this dataset should be as low as zero. The plots of datasets with noises 1%, 2%, and 5%, have candidate records that are likely to be misclassified, marked by green circles. The number of records that might get misclassified increases with the noise rate as shown. The plot that represents the
Figure 3.2: Noise and classification error relation on synthetic data generated using the DataGen tool.

dataset with 5% noise, however, has records that are not exactly positioned to be in the groups of classes but are not marked by the green circles. The reason is that although these records do not appear to belong to a specific class, they have a high probability of being correctly classified since they still appear to be closer to their class than the other class. We conclude from this study that the inserted noise of the generated data is directly proportional to how much classification error is expected from a classification algorithm running on this data.
CHAPTER 3. BASIC STREAMING RANDOM FORESTS

For running the experiments using the Basic Streaming Random Forests algorithms, we generate 6 datasets. Each dataset has one million records, 5 numeric attributes, and 5 target classes. The difference between the datasets is the noise level used to generate them: 1%, 3%, 5%, 7.5%, 10%, and 15%, respectively. Each dataset is used by both the standard and Streaming Random Forests algorithms. The training set for the standard Random Forests algorithm is 1% of the records randomly chosen from the original dataset, 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.

Real data

The Forest CoverType dataset from the UCI repository [3] 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 [16], for example, are 30% using neural network and back propagation, and 42% using linear discriminant analysis, with 11,340 training records where the number of records from each class are equal.

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 10 attributes. This new derived 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.

3.2.4 Parameters settings

For testing the Streaming Random Forest algorithm on the synthetic datasets, we discretize each attribute’s values into 200 intervals, and set $n_{\text{min}} = 200$ and $\alpha = \frac{1}{8}$. Both algorithms (standard and streaming) grow 50 trees with $M = 3$ attributes considered at each node. The tree window of the streaming version equals the total number of training records divided by the number of desired trees, giving around 19,960 records. Note that at this stage of the algorithm the tree window is constant for all trees and set to the maximum possible. However, the number of training records is still much smaller than the number used in other streaming decision-tree constructions [27,37,39].

For testing the algorithm on real data, both standard and streaming algorithms grow 150 trees using $M = 4$ attributes for building each node. The value of $\alpha$ is $\frac{1}{4}$, the value of $n_{\text{min}}$ 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.
3.3 Results and Discussion

3.3.1 Classification accuracy

Figure 3.3 shows the classification error rates of both algorithms for the six synthetic datasets. The two algorithms have comparable classification error rates. The confidence intervals for each test point of the two algorithms overlap at a confidence level of 95%, except for the dataset with 3% noise (the second recorded point on the graph). For this specific dataset, the standard algorithm records an average classification error of 2.05%, with confidence interval of [2.02%, 2.08%], and our algorithm records an average error of 2.14%, with a confidence interval of [2.09%, 2.19%]. Even though the confidence intervals do not overlap, our streaming algorithm gives an error that is still less than the actual noise of the data (3% noise).
Table 3.3 presents the classification errors and confusion matrices of both algorithms for the Forest CoverType dataset. Both algorithms have a classification error of approximately 25%, with a confidence interval of ±0.35% at a confidence level of 95%. 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.

### Table 3.3: Classification errors and confusion matrices for Forest CoverType dataset

<table>
<thead>
<tr>
<th></th>
<th>Standard Random Forest</th>
<th>Streaming Random Forest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classification Error%</td>
<td>24.73%</td>
<td>24.96%</td>
</tr>
<tr>
<td>Test Set</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>219 from class 1</td>
<td>285 from class 2</td>
</tr>
<tr>
<td>Confusion Matrix</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>True class</td>
<td>True class</td>
</tr>
<tr>
<td></td>
<td>1  2</td>
<td>1  2</td>
</tr>
<tr>
<td>1</td>
<td>147 53</td>
<td>149 56</td>
</tr>
<tr>
<td>2</td>
<td>72 232</td>
<td>70 229</td>
</tr>
</tbody>
</table>

### 3.3.2 Classification time per record

We base our classification time measurements on the synthetic data having a noise level of 15%. We consider the effect of different forest sizes, that is different numbers of trees. The forest sizes we use are 5, 50, 100, 150, 200, 250, 300, 350, 400, 450, and 500 trees.

The per-record classification times are shown in Figure 3.4. The times are measured in microseconds. The increase in per-record classification time with the number of trees in the forest is no worse than linear, which gives a per-record classification complexity of $O(U)$, for a forest consisting of $U$ trees. The average flow rate of a stream that this implementation of the Basic Streaming Random Forest algorithm
can handle is $1.7 \times 10^4$ 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 [19]) which would allow a stream rate of up to $2.8 \times 10^4$ records/sec.

These results are based on a small machine which, in practice, might not be suitable for use for streaming applications. This is one factor that explains why the rate of streams that our algorithm can handle is not very high. Another reason is that our code is not optimized, so execution time savings are not likely to be achieved.

### 3.4 Summary

This chapter has defined the Basic Streaming Random Forests algorithm, an extension of the standard Random Forest algorithm due to Breiman [19]. 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 the parameters $n_{\text{min}}$, node window and tree window to decide when to transform frontier nodes, carry out a limited form of pruning, or to begin constructing a new tree. These refinements mean that the algorithm requires many fewer labelled records for training than other stream-based decision-tree algorithms [27, 37, 39].

The Streaming Random Forests algorithm handles multi-class problems successfully as opposed to many stream classification algorithms that are designed and/or tested on only two-class classification algorithms. It is fast enough to handle stream rates of up to $2.8 \times 10^4$ records/sec, when executed on a fairly small machine (Pentium 4 machine with 3.2 GHz processor and 512MB RAM). Its per-record classification time complexity is $O(U)$, where $U$ is the number of trees in the forest.

The limitations of this version of the algorithm include:

- The algorithm only handles streams from Scenario 0, which means it does not take the concept-drift problem into consideration.

- The algorithm is based on some defined parameters, node window, tree window, and $n_{\text{min}}$, that stay constant during execution. This might cause the classification accuracy to drop if the parameters values have not been chosen carefully. Having fixed parameters limits the algorithm flexibility since the analyst must be familiar with the data to be able to choose good values for these parameters.

- The code of the algorithm is not optimized, which leads to extra time for execution. We believe that the algorithm would be able to handle streams with higher rates if the code was optimized.
Chapter 4

Dynamic Streaming Random Forests

We extend the Basic Streaming Random Forests algorithm to the Dynamic Streaming Random Forests algorithm [9], that can accommodate Scenario 1 of Figure 1.2. The Dynamic Streaming Random Forests algorithm is a self-adjusting stream-classification algorithm that is able to reflect when concept drift occurs in the underlying data.

The major improvements of the Dynamic Streaming Random Forest over the basic version include:

- Dynamic behavior of the algorithm, so that most of the algorithm parameters are assigned values only for the initial forest-building phase; then, for each of the subsequent building phases, the parameters adjust their values depending on the data records seen so far.

- The ability to manage the case where concept drift occurs in the data, so the model always reflects the recent data.
CHAPTER 4. DYNAMIC STREAMING RANDOM FORESTS

4.1 Algorithm Details

As for the basic version of the algorithm, the extended versions for Scenario 1 and all the upcoming scenarios only handle numerical or ordinal attributes for which the maximum and minimum values of each attribute are known, and deal with their values by discretizing the range of each attribute into fixed-length intervals by the algorithm. The total number of trees grown is supplied as an input parameter. Trees can be replaced in the ensemble but the total number of trees is kept constant.

4.1.1 Initial forest-building phase

The dynamic algorithm, like the Basic Streaming Random Forests algorithm, initially grows a defined number of trees. The difference is that the tree window is not constant for all trees. Instead, its value depends on the performance of the tree under construction based on the data records seen so far. This is accomplished by the following. Whenever $\text{tree}_{\text{min}}$ records have contributed to building a tree during the training phase, the algorithm computes the current classification error of the tree based on the next $0.1 \times \text{tree}_{\text{min}}$ incoming unseen records. If the tree error is less than a threshold called the tree threshold, then the algorithm stops building this tree and begins to build the next tree. Otherwise, the algorithm continues building the tree using half the previous value of $\text{tree}_{\text{min}}$ records, before the classification error of the tree is computed again. This technique of reducing the number of records by half each time the classification error of the tree is computed follows the geometric series, which states: $\sum_{i=0}^{\infty} \frac{1}{2^i} = 2$. Each tree has, therefore, a different tree window that is never greater than $2 \times \text{tree}_{\text{min}}$.

Since the value of tree window is not known until the tree is built, the value of the
parameter node window for the dynamic version of the algorithm is derived based on the parameter $tree_{min}$ instead of tree window. It is defined as

$$node \ window = \alpha (tree_{min}/2^{node \ level})$$

The two parameters $tree_{min}$ and tree threshold, are initially assigned constant values defined by the user. Their values are, however, automatically adjusted during the later building phases of the forest based on the data records seen so far. A typical initial value for tree threshold is $1/C$, where $C$ is the number of classes. This threshold value ensures that none of the trees performs worse than a random tree that classifies records uniformly randomly. Table 4.1 describes the parameters that are defined for designing the Dynamic Streaming Random Forests algorithm.

Table 4.1: Definitions of parameters for Dynamic Streaming Random Forests

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree threshold</td>
<td>an error threshold that is used to as an upper bound to stop the current tree building</td>
</tr>
<tr>
<td>$tree_{min}$</td>
<td>number of records that a tree must process before comparing its classification error with the tree threshold</td>
</tr>
<tr>
<td>$tree_{size}$</td>
<td>average number of nodes per tree</td>
</tr>
<tr>
<td>$tree_{error}$</td>
<td>the classification error for $tree_{i}$</td>
</tr>
<tr>
<td>$w$</td>
<td>percentage of trees to replace when there is no concept drift, set to 25</td>
</tr>
<tr>
<td>$H_{i}$</td>
<td>entropy value of attribute $i$</td>
</tr>
<tr>
<td>$H$</td>
<td>final entropy: average of the entropies of attributes</td>
</tr>
<tr>
<td>$H'$</td>
<td>normalized entropy, which has a value in range [0,1]</td>
</tr>
<tr>
<td>$H_{AVG}$</td>
<td>accumulated average of $H'$ computed since the last detected concept drift</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>threshold that defines when there is a change</td>
</tr>
<tr>
<td>$R$</td>
<td>number of trees to replace in case of concept drift</td>
</tr>
</tbody>
</table>
4.1.2 Derivation of parameters for subsequent building phases

Upon the completion of the initial training phase (i.e. the total number of trees have been grown), the algorithm enters a test phase where the classification accuracy for the forest is calculated using previously unseen labelled data records. The classification error, $tree_{error_i}$, for each individual tree $i$, is also calculated and used to assign a weight to the tree. The algorithm is then expected to classify unlabelled data records with an accuracy that is approximately equal to the calculated forest accuracy.

Assume a new data record arrives at time $t + 1$, where the time is incremented upon the arrival of each new block of labelled records. The algorithm derives new values of the parameters to be used in the subsequent building phase as follows:

- A new tree threshold is calculated as the average error of trees at time $t$:

$$tree \text{ threshold}(t + 1) = \frac{1}{U} \sum_{i=1}^{U} tree_{error_i}(t)$$

where $U$ is the number of trees in the forest. Although this derivation seems simple, experiments show that it gives good results. The derivation was chosen after extensive studies of various possibilities for deriving a tree threshold.

We first attempted to base the derivation of tree threshold on probability theory. The threshold is set to the average probability of getting a record misclassified given the classification error of each tree and the number of trees. We achieved this by using combinatorial mathematics to count all possible combinations of tree votes and exclude those that result in correct voting since we are seeking to find the classification error. The formula becomes complicated with large number of trees (50 trees), and implies unrealistic tree threshold values. The correlation between the trees is not taken into consideration, and so the
calculated errors do not represent the real situation.

We next attempted to base the derivation of tree threshold on Breiman’s generalization error for standard Random Forests [19]. Breiman defines an upper bound on the forest error in terms of the strength of trees, \( s \), and the variance between the tree strengths, given by:

\[
\text{Forest error} \leq \sum_c \text{var}(P(\text{correct classification}) - P(\text{wrong classification classified as } c))/s_c^2
\]

where \( c \) is the class label, and \( s_c \) is the strength of the tree in terms of classification accuracy with respect to a class \( c \), calculated as:

\[
s_c = E(P(\text{correct classification}) - P(\text{wrong classfication classified as } c))
\]

Note that we change the notation used in the original equation for simplicity.

We reverse the equation of Forest error such that we calculate tree threshold at time \((t + 1)\) as the expected classification error of the trees, given the forest calculated error and the variance calculated from the run at time \( t \). The results of using this derivation were not as expected. The values for tree threshold are too big for a classification error. We believe the reason is that the Forest error equation has weak bounds, and the actual forest error is much smaller than the defined bound, so reversing the equation does not lead to reasonable values.

- A new \( n_{\text{min}} \) is derived from the stopping condition of the Hoeffing bound equation when using Gini index; \( \Delta \text{Gini} > \epsilon \), where \( \epsilon = \sqrt{\frac{L^2 \ln(1/\delta)}{2n}} \). The parameter \( n_{\text{min}} \) is set to the number of records \( (n) \) of \( \epsilon \), and \( L \) is set to 1 since the Gini index evaluates to values in the range between 0 and 1. This definition of \( n_{\text{min}} \) calculates the average number of records that each node needs to process for satisfying the Hoeffding bound test. The new \( n_{\text{min}} \) is therefore defined as:
\[ n_{\text{min}}(t + 1) = \frac{\ln(1/\delta)}{2(\Delta \text{Gini}(t))^2} \]

where \( \Delta \text{Gini}(t) = \text{Gini}_{\text{highest}} - \text{Gini}_{\text{second highest}} \) has been computed during the building phase at time \( t \), and \( \overline{\Delta \text{Gini}(t)} \) is the average value of \( \Delta \text{Gini}(t) \).

- A new \( \text{tree}_{\text{min}} \) is calculated as the multiplication of the minimum number of records each node is required to process, \( n_{\text{min}} \), and the average size of trees that are currently in the forest:

\[ \text{tree}_{\text{min}}(t + 1) = n_{\text{min}}(t + 1) \times \overline{\text{tree}_{\text{size}}(t)} \]

where \( \overline{\text{tree}_{\text{size}}(t)} \) is the average size of trees from the building phase at time \( t \), measured in number of nodes per tree.

### 4.1.3 Forest update

Whenever a new block of labelled data records arrives, the next training phase begins. The algorithm replaces \( w\% \) of the total number of trees with new trees, grown from newly-arrived data records. The trees with the worst weights, that is those with the largest \( \text{tree}_{\text{error}} \), are replaced. We believe that, even if there are no drifts in the data, the algorithm should always update the classification model to represent any small changes that are not recorded as real drifts. The value of \( w \) is obtained empirically. We choose the value of \( w \) to be 25. We consider that this percentage is enough to keep the learned concept up-to-date, while not adding too much time to each building phase.

New trees are grown using the parameters derived from the previous training phase, namely \( \text{tree}_{\text{threshold}}(t + 1) \), \( n_{\text{min}}(t + 1) \), and \( \text{tree}_{\text{min}}(t + 1) \). If there is a
concept drift in the data, then the values of these parameters are reset to their initial values since their most recent values were derived based on data records generated from the old concept. More trees are replaced in the case of concept drifts to reflect the new distribution. The number of trees to replace is determined based on the significance of the concept drift.

**Concept-drift detection**

Concept drift in data streams can be thought of in different ways. There are three categories:

- **Category 1:** changes that are generated from variation in noise. In this case, the hyperplanes that define the boundaries between classes move slightly, but the underlying structure defining the classes has not really changed.

- **Category 2:** changes that are generated from a gradual change in the underlying structure defining the classes. In the period during which the structure is changing, the stream contains a mixture of data records reflecting the old structure and data records reflecting the new structure. The percentage of the records that reflect the new structure gradually increases until the stream reflects only the new structure.

- **Category 3:** changes that are generated by a sudden change in the underlying structure. The records suddenly reflect the new structure.

We use an entropy measure to detect concept changes in streams. We base our idea on the work of Vorburger and Bernstein [63]. We use the two-windows paradigm
described in Chapter 2. Since the distribution of the data is not known, the probability mass function cannot be used directly. Instead, we use counters to find the probabilities of occurrences of values associated with a specific interval and specific class within each attribute. We use an incremental method to maintain the probabilities of occurrences, so that the corresponding counters are updated upon each record arrival. Hence, the algorithm does not store records for updating the counters.

Each attribute’s entropy is calculated as:

\[
H_i = \sum_{c=1}^{C} \sum_{k=1}^{K} (u_{kc_{\text{cur}}} \log_2 u_{kc_{\text{cur}}} - u_{kc_{\text{ref}}} \log_2 u_{kc_{\text{ref}}})
\]

where \(C\) is the number of classes, \(K\) is the number of intervals for attribute \(i\), and \(u_{kc_{\text{cur}}}\) and \(u_{kc_{\text{ref}}}\) are the ratios of the number of records within an interval \(k\) and assigned to class \(c\), to the total number of records of the current and reference windows, respectively. The absolute value of \(H_i\) is used because the magnitude of the change is what really matters. It does not matter which window has the greater entropy.

The final difference in entropy is the average of the differences between the entropies of the current and the reference windows for each attribute. For a data stream with \(m\) attributes, the final entropy is calculated as:

\[
H = \frac{1}{m} \sum_{i=1}^{m} H_i.
\]

The possible values of \(H\) are in the range \([0, |\log_2 1/C|]\). \(H\) is zero when entropies of both windows are equal, and hence there is no drift between the two windows, and \(H\) is maximized and equal to \(|\log_2 1/C|\) when a large drift appears. The minimum value of an individual window entropy occurs when the records of the window are all from one class while the maximum value occurs when the records are evenly distributed.
among the classes.

To better understand the significance of a concept drift, we normalize $H$ by dividing it by its maximum possible value: $H' = H' / |\log_2 1/C|$. The value of the normalized entropy $H'$ is hence in the range $[0,1]$ and represents the fraction of concept drift. We use a threshold value that depends on the accumulated average of the computed entropies seen since the last change. If a drift is detected, more trees must be replaced. The algorithm records a change in distribution if $H' > \gamma + H_{AVG}$, where $H_{AVG}$ is the accumulated average of the entropy differences computed since the last recorded change in distribution, and $\gamma$ is a constant threshold defined empirically.

**Tree replacement policy**

The number of trees to be replaced depends on both the presence of a concept change and the degree of the change. The larger the value of $H'$, the greater the number of trees that should be replaced. The algorithm calculates the number of trees to replace, $R$, using the equation:

$$R = \begin{cases} 
H'U & \text{if } (H'U + \frac{1}{C}\{(1-H')U\}) > U/2; \\
H'U + U/2 & \text{otherwise}
\end{cases}$$

This equation considers $H'$ as the fraction of the total number of trees to replace only if the remaining set of unchanged trees, $(1-H')U$, has a higher probability of contributing to making the correct classification decision when combined with the replaced set of trees. We approximate the fraction of the number of unchanged trees that might positively share in the classification decision as $1/C$, since an inaccurate tree that uniformly guesses the classes of the input records still has a probability of
1/C of getting the correct classification. If \( R \) turns out to be less than \( w \% \) of the number of trees \( U \), then it is set to \( R = R + \frac{1}{100} \times w \times U \). If the number of trees to be replaced plus the number of remaining trees that are expected to give correct classification is less than half of the total number of trees, then the number of trees to replace is calculated as \( H' \times U + U/2 \).

If two concept drifts occur close to each other, so that the algorithm detects a concept drift before it completes replacing the \( R \) trees, then it does not record a change, until it finishes its current update phase. This ensures that the algorithm does not go into an infinite number of update phases in the case of having many small concept drifts appearing close to each other.

The algorithm’s behavior for the case of a new block of labelled data records is shown in Figure 4.1.

```/*To update the forest at time t+1*/
compute normalized entropy (H')
compute H_{AVG}
if H' ≤ γ + H_{AVG}
    call BuildTree to replace the least accurate w% of trees using
tree threshold(t+1), n_{min}(t+1),
and tree_{min}(t+1)
else
    record a change in distribution
    reset tree threshold, n_{min}, and
tree_{min} to initial values
    calculate R
    call BuildTree to replace the least accurate R trees
end if
```

Figure 4.1: Algorithm’s behavior in the case of new labelled data arrival
4.2 Implementation and Experimental Settings

The Dynamic version handles Scenario 1 where the first block of labelled data is used for building the initial forest, and each of the the next blocks of labelled data is used for one of the subsequent training phases during which the forest is updated. As in the basic version of the algorithm, the number of attributes considered at each node for splitting is $M = \text{int}(\log_2 m + 1)$.

4.2.1 Testing criteria

The criteria by which we evaluate the performance of our algorithm are related to the way the algorithm reacts when new data records arrive, regardless of whether a change had occurred or not. We consider the following criteria:

- **The entropy-based concept-drift detection technique.** We evaluate our entropy based concept-drift detection technique by testing it on evolving datasets containing concept drifts from the three categories explained previously. A significant value of the $H'$ should be recorded at the occurrence of any concept change. The values of $H'$ at drift points should also reflect the significance of the drift associated with each point.

- **The classification accuracy of the model.** As with the basic version of the algorithm, we measure the classification accuracy of the algorithm with its new features by calculating the classification error. The error percentage is recorded at each test phase, showing the variation in error in the presence of concept changes. We evaluate the algorithm by comparing the calculated error to the error implied by the dataset’s noise.
• The dynamic nature of the algorithm. We examine the dynamic nature of the algorithm by which the algorithm automatically adjusts its parameters based on the data seen so far, and resets the parameters to their initial values in the case of concept drift.

• Model build/update time. We also include the time that the algorithm takes to process a stream from Scenario 1 in our performance-evaluation criteria, showing the average time it takes to access each data record and hence, the average stream rate the algorithm can handle. The running time of the algorithm also shows the relation between the strength of the concept changes, if any, and the time it takes the algorithm to rebuild the forest.

### 4.2.2 Testing platform

As in the basic version, the machine on which we conduct our experiments uses a Pentium 4 3.2 GHz processor and 512MB RAM. We base the implementation of the Dynamic Streaming Random Forests on the basic version [8], implemented using Fortran 77. Table 4.2 represents a summary of the work done for extending the Basic Streaming Random Forests algorithm to the Dynamic Streaming Random Forests algorithm.

### 4.2.3 Datasets

We generate a number of synthetic datasets using the DataGen data-generation tool [50]. Each data set has a different concept from which it was created. We then combine the datasets into one dataset with concept drift at the boundaries between datasets. We use the combined dataset for our experiments in order to show the
Table 4.2: Implementation details of Dynamic Streaming Random Forests

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Basic Forests</th>
<th>Streaming Random Forests</th>
<th>Dynamic Streaming Random Forests</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of code lines</td>
<td>1700 lines</td>
<td>2530 lines</td>
<td>used: 1450 ≡ 85.3% of basic version</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>altered: 250 ≡ 14.7% of basic version</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>deleted: 0 ≡ 0% of basic version</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>new added code lines: 830</td>
</tr>
<tr>
<td>No. of main subroutines</td>
<td>5 subroutines</td>
<td>7 subroutines</td>
<td>used: 3 ≡ 60% of basic version</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>altered: 2 ≡ 40% of basic version</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>deleted: 0 ≡ 0% of basic version</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>new subroutines : 2</td>
</tr>
</tbody>
</table>

properties of our algorithm. The combined dataset has 5 attributes (2 numeric and 3 ordinal) and 5 target classes. We simulate the three categories of concept drift in the combined dataset. The total size of the dataset is 7 million records, with concepts varying as follows:

- The first block of 1,500,000 records simulates a concept with changing noise (category 1). The noise starts at 5% for the first 500,000 records, then it drops to 3% for the next 500,000 records, and then decreases to end up with a noise of 1% for the last 500,000 records.

- The second block of 5,000,000 records simulates the second category of concept drift. A new dataset that also has 1% noise but is generated using different learning rules mixes with the records generated from the previous learning rules. Each block of 500,000 data records has mixed proportions of the data generated from old rules and the data generated from the new rules. The proportion of old data drops by 10%, while the new data proportion increases by 10% in successive blocks. In other words, the first 500,000 block of records has 90% old
data and 10% new data, the penultimate block has the percentages reversed, and the last block has data that is entirely generated from the new rules.

- The third block of 500,000 records simulates the third category of concept drift. The records for this block are generated from new learning rules with 5% noise.

Figure 4.2 shows the noise change throughout the dataset.

Figure 4.2: Concept drift in the synthetic data

The generated dataset is used for training and testing. The number of records used to test the forest each time it goes into a test phase is 5000, and, as mentioned before, the number of records used to test the tree under construction is $0.1 \times \text{tree}_{\min}$.

### 4.2.4 Parameters settings

For all our experiments, the algorithm grows 50 trees using $M = 3$ attributes for each node. The initial values of parameters are: $\text{tree}_{\text{threshold}} = 0.2$, $n_{\min} = 200$, 

and $\text{tree}_{\text{min}} = 3000$. The value of the entropy window of the current and reference windows is constant and set to 1000 records. The values of $\gamma$ and $\alpha$ we use are 0.05 and 0.5 respectively, and, as aforementioned, we set $w$, the percentage of trees to replace when no concept changes occur, to 25%.

4.3 Results and Discussion

4.3.1 Entropy-based concept drift detection technique

We first test our entropy-based concept-drift detection technique and record the points at which the technique detects concept changes. Figure 4.3 shows the entropy $H'$, the average entropy $H_{\text{AVG}}$, and the threshold $\gamma + H_{\text{AVG}}$ at positions in the stream.

![Figure 4.3: Entropy versus number of records](image)

The algorithm records seven concept drifts at points 501000, 1506000, 2049000, 5011000, 5533000, 6032000, and 6501000. The detected drift points are marked with asterisks and the letters A, B, C, D, E, F, and G. The points’ normalized entropy ($H'$) values are shown on the graph. Note that a window of 1,000 records (current window
size) reflecting the new concept needs to be seen before a change is detected. Actual change points therefore, occur earlier than the detected changes. The first recorded change (A) represents a drift of category 1, where the noise drops from 5\% to 3\% keeping the underlying rules unchanged. The entropy value of the second simulated drift, where the noise drops to 1\% at point 1,000,000 does not show any significant change. This drift is therefore not recorded. The subsequent five recorded changes (B, C, D, E, and F) represent the points during the gradually drifting concept (category 2 drift), where the noise is unchanged at 1\%. Some of the boundaries, however, are not recorded. This depends on how different the mixture of data records are from one set to the next.

The last recorded change (G) is the category 3 drift where the noise of the dataset increases and the underlying rules from which the new records are generated change. As shown in the Figure, all recorded changes have similar entropy values in the range of [0.07, 0.1] except for the last value, which is the maximum recorded value of 0.27. This is because both the noise and learning rules change at this point.

4.3.2 Classification accuracy

We measure the classification error of the system and record how far the error is from the expected error given the actual noise in the data. Figure 4.4 shows the classification error, versus the number of records seen. The recorded points (crosses) of the graph corresponds to each time the algorithm has finished building/updating the forest.

The results show that the algorithm successfully reacts to concept changes of categories 1 and 3 – the classification error changes with the sudden change in data
Figure 4.4: Classification error of the forest

noise and/or underlying rules. This can be seen at the first two drift points of the data (category 1), when the number of records is 500,000 and 1,000,000, and the last point (category 3) when the number of records processed is 6,500,000 records. Although the entropy change-detection technique did not record a change for the second drift point (when the noise drops from 3% to 1% at record 1,000,000), the algorithm still performs well, giving a classification error that is approximately equal to the data noise.

The experiments record poor classification during the gradual drift in the underlying rules (drift of category 2), shown at the point where the classification error is about 7.5% and the subsequent few points. Note that the noise in the data during the period of this drift is 1%. The justification for this is that, since the data from new learning rules are added into the old dataset gradually, with the first block of mixed data having only 10% of the new records, the model does not see enough records to learn the new concept. Instead, it considers the data generated from new learning
rules as an increase in the noise presented in the data. The classification error for
this block of data should therefore, be around 10%. Our algorithm performs better
than that, and records a classification error of 7.5%, which drops as the number of
records from the new concept increases until it reaches classification errors in the
range of $[1.65\%, 2.55\%]$. The period during which the classification errors are around
2% starts when the records that are generated from the new rules represent 70% of the mixed dataset.

The results also show that, whenever a change is detected, more data records are
used to update the model than in the case where no concept drift appears in the data.
This is visible in the graph by noticing the gap between any two consecutive points
showing how many records were used since the last update of the forest. The points
are close together when there is no change in the data, while the gap increases at
concept changes.

Figure 4.5 shows the difference between the classification errors achieved by the
algorithm and the actual noise in the data, represented by the gray area in the graph.
From this figure, it can be seen that the only significant difference occurs during the
period when data records from two different underlying rules are present. Some points
show classification errors that are less than the noise of the data, which confirms that
the algorithm performs well.

4.3.3 Dynamic adjustment of parameters

We illustrate the dynamic nature of the algorithm by examining the variation of the
parameter values during the execution of the algorithm. We consider $tree_{threshold}$,
$n_{min}$, $tree_{min}$, and the number of trees to replace for our testing. Figures 4.6, 4.7,
Figures 4.8, and 4.9 represent the values of the four parameters respectively.

The Figures show that, when a concept change is detected, the values of the parameters are reset to their initial values. As shown in Figures 4.6, 4.7, and 4.8, the values of \textit{tree threshold}, \textit{n}_{\text{min}}, and \textit{tree}_{\text{min}} decrease when the model is better representing the data records for the following reasons: 1) Since \textit{tree threshold} represents the average classification error of individual trees, the stronger the trees, the higher their classification accuracy, and therefore, the smaller the value of \textit{tree threshold}. 2) Small values of \textit{n}_{\text{min}} and \textit{tree}_{\text{min}} mean that the algorithm is performing well and does not require large numbers of data records in order to keep its classification model up-to-date. It is also visible in the figures that the values of the parameters are reset to their initial values at the points where changes were detected.

Figure 4.9 shows that the algorithm initially grows 50 trees, then it normally replaces 25\% of the trees (12 trees) unless a concept drift is detected, when the number of trees to replace increases based on the significance of the change. For the first six drift points that the algorithm detects, the number of trees to be replaced...
Figure 4.6: Values of tree threshold through execution

Figure 4.7: Values of $n_{min}$ through execution
4.3.4 Time for building and updating the forest

We consider the time for building and updating the forest. To better approximate the running time of the algorithm, ten runs were conducted with random selection of the attributes used to build each node. The average total time for running the algorithm on the dataset is approximately 32.82 minutes. The recorded average time for the initial forest building and testing (50 trees) is around 70 seconds per 285,300 records. This gives an average per-record time of $0.245 \times 10^{-3}$ seconds, which allows the algorithm to handle streams of rates up to $4.08 \times 10^3$ records/second. The time
Figure 4.9: Number of replaced trees through execution

for building the initial forest represents 3.55% of the total running time.

Note that, in our experiments, each data record requires an I/O operation from disk. Stream applications, in practice, receive the data records directly from their sources (i.e. sensors, networks, or satellites). We believe that receiving the records from their sources instead of reading them from a disk would significantly decrease the total execution time for our algorithm, which in turn allows the algorithm to handle streams of much higher rates that what our experiments record.

Figure 4.10 shows the time in seconds of one instance of running the algorithm on the dataset. The figure shows the initial building phase takes about 72 seconds. The update time for the forest when there is no concept changes is 36.4 seconds on average, which is about half the time of the initial forest building phase. It reaches a maximum of 55.8 seconds during the period that reflects the second category of concept changes, and a minimum of about 14.5 seconds. The variation of the update time when there is no recorded concept change depends on how hard it is to learn
the boundaries between classes and the level of noise of the data.

The update time in the presence of concept changes is 56.8 seconds on average (about 78% of the initial forest building time), reaching a maximum of 81 seconds in the period of the second category concept drift, and having a minimum value of 41.4 seconds. The measurements show that the average time overhead for updating the forest, whether a concept change is present or not, is usually less than the average time for building the entire forest.

There are cases, however, where the algorithm takes more time to update the forest than to build it initially. These cases usually appear at the beginning (or the end) of the period that represents a category 2 concept change. The reason behind this is, as mentioned before, the algorithm considers the data records that are generated from the new (or old) concept as noise, which makes the boundaries between classes harder to learn, and therefore, the trees need more records and time to learn. If the initial forest was built using the records with category 2 concept drift, then it would need

![Figure 4.10: Time for building/updating the forest](image-url)
CHAPTER 4. DYNAMIC STREAMING RANDOM FORESTS

more time to be built than to get updated. The figure shows an instance of the case where the algorithm takes more time to update the forest than to build it initially.

4.4 Summary

This chapter has presented the Dynamic Streaming Random Forests algorithm, a stream-classification algorithm that adjusts its parameters dynamically, and deals with evolving streams using an entropy-based change-detection technique. The algorithm extends the Basic Streaming Random Forests algorithm [8].

The algorithm is tested on a synthetic data stream that contains 7 million records with different concept-change points falling in the three different categories of concept change that were explained earlier. It gave the expected behavior when tested; the concept-drift points were detected; the parameters were automatically adjusted; and the classification accuracy throughout the execution was approximately equal to the expected rate determined by the noise in the data. The time cost of updating the forest varies with the significance of the concept changes. It is usually, however, less than the time cost for building the entire forest on the same data records.

Note that other synthetic datasets were used for testing our algorithm, and similar results were obtained. Therefore, we only include the results of our combined dataset since it shows the properties of our algorithm.

The most important limitations of the Dynamic Streaming Random Forest include

• It does not deal with Scenarios 2 and 3, which are the closest scenarios to real life. It always assumes that the arriving block of labelled data records is big enough to build/update the classifier.
- As for the basic version of the algorithm, the code of the Dynamic Streaming Random Forests is not optimized.
Chapter 5

Autonomic Streaming Random Forests

We further extend the Dynamic Streaming Random Forests algorithm to propose the Autonomic Streaming Random Forests algorithm, for handling Scenario 3 in Figure 1.2. Accommodating Scenarios 2 and 3 requires the algorithm to make use of blocks of labelled data records that are not necessarily large enough to build/update a robust classifier. Each time a block of labelled data records ends, the algorithm must be capable of deciding whether the forest under construction should be used for classifying upcoming unlabelled records or whether more labelled records must be seen before deploying the forest for classification. We consider Scenarios 2 and 3 together in this chapter because handling Scenario 3 implies that the algorithm is also able to handle streams for Scenario 2. As mentioned in Chapter 1, Scenarios 0, 1, and 2 are all special cases of Scenario 3.
5.1 Algorithm Details

The Autonomic Streaming Random Forests algorithm mimics the Dynamic Streaming Random Forests algorithm in building the forest. It takes the number of trees to grow as an input parameter, and grows the trees sequentially, before entering a test phase. It then updates the model upon the arrival of any sufficiently-large new block of labelled data records, replacing the worst \( w \% \) of the trees in the forest when there is no concept change. The parameter \( w \) is set to 25. The update is only critical when a concept change is detected in the stream, and then the algorithm replaces \( R \) trees (more than \( w \% \) of trees). The number of trees to replace in the case of concept change, \( R \) (defined in Subsection 4.1.3), depends on the significance of the change. The more significant the change, the more trees are replaced.

An incoming block of labelled data records in a stream from Scenario 3 may not be large enough to grow the required number of trees for building/updating the forest. It may even be a single labelled record. The algorithm must evaluate the forest under construction each time a block of labelled records ends, and decides if the forest already built is accurate enough to deploy for classifying unlabelled data records. We call the phase during which the forest is evaluated for deployment the evaluation phase. The criterion that the algorithm uses to evaluate the forest during the evaluation phase is the margin function \( (mg) \) defined by Breiman [19].

The forest is evaluated based on a stored set of labelled data records, called the evaluation set. This set of records is a small percentage of randomly selected labelled records that have been processed since the previous evaluation phase. We set the selection percentage to 1\% since we aim to store as small a set as possible so that memory does not become an issue and the time taken to evaluate the forest
is minimized. We refer to the number of records in the evaluation set by eval cnt.

Table 5.1 describes the new parameters that are defined as part of the design of the Autonomic Streaming Random Forests algorithm.

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval cnt</td>
<td>the number of records in the evaluation set</td>
</tr>
<tr>
<td>mg</td>
<td>the margin function defined by Breiman [19]</td>
</tr>
<tr>
<td>$\bar{mg}$</td>
<td>average value of the margin function mg</td>
</tr>
<tr>
<td>conf</td>
<td>the confidence of the forest classification (linear method)</td>
</tr>
<tr>
<td>mgthreshold</td>
<td>simple threshold value to test whether the forest is ready for deployment or not</td>
</tr>
<tr>
<td>$v$</td>
<td>the excess of the number of correct votes over uniform random voting</td>
</tr>
<tr>
<td>conf$'$</td>
<td>alternative measure of the forest confidence (non-linear method)</td>
</tr>
<tr>
<td>conf$'$given</td>
<td>the defined accepted confidence of the algorithm</td>
</tr>
<tr>
<td>mg$'$threshold</td>
<td>alternative measure of mgthreshold based on conf$'$given</td>
</tr>
</tbody>
</table>

5.1.1 The margin function

The margin function defined by Breiman [19] for the original Random Forests algorithm measures the strength of an ensemble of classifiers at some time $t$. It measures how far the average number of correct votes for a specific record is from the average number of (incorrect) votes of the next candidate class. Given a Random Forest having $U$ trees $tree_1, tree_2, ..., tree_U$, recall that the margin function is defined as

$$mg(X, Y) = avg_U I(tree_u(X) = Y) - max_{j \neq Y} avg_U I(tree_u(X) = j)$$

where $1 \leq u \leq U$, $I(\cdot)$ is the indicator function, $X$ is a labelled record from class $Y$, and $tree_u(X)$ is the vote class of $tree_u$ for the record $X$. Clearly, the higher the value of $mg$, the stronger the forest.
5.1.2 Initial forest-building phase

Initially, the algorithm grows the defined number of trees based on the initial block of labelled data records of the stream. If the labelled records of the initial block are not sufficient to complete building the entire forest, then the algorithm enters an evaluation phase, but only if enough trees were grown for evaluation. We set the threshold of the number of trees that need to be built before the first evaluation phase takes place to 10% of the total required number of trees.

In the evaluation phase, the average of the margin value, $\overline{mg}$, for the current forest is calculated, ignoring the tree under construction. We define $\overline{mg}$ to be:

$$\overline{mg} = \frac{1}{\text{eval cnt}} \sum_{i=1}^{\text{eval cnt}} mg_i(X \subset \text{evaluation set}, Y).$$

If the calculated value of $\overline{mg}$ is greater than a threshold, $mg_{\text{threshold}}$, then the forest is robust enough and it is deployed for classifying unlabelled data records. In this case, the algorithm resets the evaluation set, enters a test phase to simulate the deployment classification error, and starts collecting new evaluation records from new incoming labelled data records. If $\overline{mg}$ is not greater than $mg_{\text{threshold}}$, then no classifications for unlabelled records are made and the algorithm waits for another block of labelled data records to resume building the forest to the specified number of trees. The testing phases that follow evaluation phases do not add any features for the algorithm. They are only to simulate the algorithm deployment accuracy.

At the end of the initial building phase, the algorithm records the classification error of the evaluated forest based on the evaluation set’s records. This calculated value does not affect the decision that the algorithm takes after evaluating the forest. It only shows how well (or poorly) the forest will perform if the algorithm decides to
deploy the forest (or not) on unlabelled records.

5.1.3 Deciding when the forest is good enough

A simple definition for the value of $mg_{threshold}$ is the number of correct votes if the forest were to vote uniformly randomly. This is the ratio of the current number of trees in the forest under construction, $U$, to the number of classes of the dataset, $C$; $mg_{threshold} = U/C$. One argument for choosing this ratio is that having more trees in the forest should lead to larger differences between correct and incorrect voting and, hence, larger values of $mg$. On the other hand, a dataset with more classes results in more choices for votes, and therefore smaller differences between correct and incorrect voting which leads to smaller values of $mg$.

Having this definition of $mg_{threshold}$ means that the algorithm does not consider a forest accurate enough for deployment unless the number of correct votes exceeds the number of votes for a uniform voting ($U/C$) by at least $v = \frac{C-1}{C^2}U$ for the worst case where the rest of incorrect votes are uniformly distributed among the other $C - 1$ classes. We derive the minimum increment of the number of correct votes ($v$) from an equation we have formulated for a forest with uniformly random voting:

$$v + U/C - (U/C - v/(C - 1)) \geq (mg_{threshold} = U/C)$$

The equation assumes that $v$ is taken equally from all other classes, hence, the incorrect votes are distributed evenly among the other $C - 1$ classes.

Positive values of $mg$ define how confident the forest is in its classification. We define the algorithm confidence, $conf$, for classifying unlabelled records when deployed, by relating $mg$ and $U$ linearly, so that
\[ conf = \begin{cases} \frac{mg}{U}, & \text{if } mg \geq 0; \\ 0, & \text{otherwise} \end{cases} \]

In practice, however, the expected confidence values might have looser boundaries and the forest confidence might be greater than a linear relation suggestion.

We therefore suggest another definition of the algorithm confidence \( conf' \), in terms of \( mg \), the number of classes \( C \), and the number of trees in the current forest \( U \), so that the following conditions are met:

- If \( mg \) is negative, then the forest confidence must be zero.
- If the value of \( mg \) is equal to 0, which means the forest has voted equally for all classes (uniform random classifier), then the forest must have \( \frac{100}{C} \)% confidence.
- If the value of \( mg \) is equal to number of trees, \( U \), which means all the trees have voted for the correct class, then the forest must have 100% confidence.
- For a constant value of \( C \), the greater the ratio of \( mg \) to \( U \), the greater the confidence of the algorithm should be.
- For a constant value of \( mg/U \), the greater the value of \( C \), the greater the confidence of the algorithm should be.
- The confidence function monotonically increases as the value of \( mg/U \) increases, and the rate of increase should be faster for larger numbers of classes.

A plausible definition of \( conf' \) that satisfies these conditions is:

\[ conf' = \begin{cases} \left( \frac{1}{C} \right)^{\left( 1 - \frac{mg}{U} \right) \frac{1}{C-1} }, & \text{if } mg \geq 0 ; \\ 0, & \text{otherwise} \end{cases} \]
Figure 5.1 shows the plots of $\text{conf}'$ versus positive values of $\text{mg}/U$ for $U = 50$ trees and different number of classes $C = 2, 3, 5, \text{ and } 10$.

![Figure 5.1: conf’ plots for C = 2, 3, 5 and 10](image)

It can be seen from the figure that starting from a confidence value of about 63%, the function performs as expected. The plots are not all in the correct order in the region below this. This is expected since the function starts at lower confidence values for larger number of classes. This region of incorrect ordering of plots can safely be ignored, since it is highly unlikely that an acceptable confidence level for a classification application is below 63%.

Another definition of $m_{g_{theshold}}$ is required when using the $\text{conf}'$ equation as the algorithm confidence measure. We, therefore, derive $m_{g_{theshold}}$ from the equation of $\text{conf}'$, given the accepted confidence level that is expected from the algorithm, $\text{conf}'_{\text{given}}$ (perhaps defined by an analyst). The value of $m_{g_{theshold}}$ is defined as:

$$m_{g_{theshold}} = U \ast \left(1 - \log_{\frac{1}{C}} \text{conf}'_{\text{given}} \right)^{C-1}.$$
5.1.4 Subsequent building phases

As for the dynamic version of the algorithm, the autonomic algorithm enters a new building phase (update phase) to update the forest upon the arrival of a new block of labelled records. The algorithm’s behavior for dealing with a new block of labelled data records is affected by the state of the forest when the previous block of labelled data records has ended. Each update phase has a target number of trees to build, based on one of three conditions:

- If the number of trees of the forest is still less than the total number of trees to build, then the target number of trees for the upcoming update phase is the remaining number of trees.

- If the total number of trees has been grown, and there is no concept drift, then the target number of trees to replace is \( w\% = 25\% \) of the total number of trees.

- If the total number of trees has been grown, and there is a concept drift, then the target number of trees to replace is \( R \) (defined in Subsection 4.1.3).

The algorithm grows the target number of trees one by one. Each time the algorithm completes building a tree, it either adds the tree to the forest if the total number of trees has not yet been reached, or it replaces the existing tree with the worst performance by the new tree.

There are two main situations at which the algorithm may leave an update phase:

- The target number of trees to build is reached (the block of labelled records was big enough). In this case the algorithm acts like the dynamic version and enters a test phase to record the classification accuracy.
• The current block of labelled records ends before the algorithm has completed its update phase. In this case, the algorithm enters an evaluation phase only if two conditions are satisfied:

1. The number of trees already grown is at least 10% of the total number of trees.

2. The algorithm has at least seen $tree_{min}$ records since the previous evaluation phase when there is no concept drift, or since the last detected concept drift. This ensures that the initial structure of at least one additional tree has been built since the most recent event (evaluation phase or concept drift detection). The relation $eval\ cnt \geq 0.01 \ast tree_{min}$ must therefore hold.

Otherwise, the algorithm does not enter an evaluation phase. It waits until a new block of labelled data records arrives to resume the update phase by continue to build the tree under construction, and keep on building more trees in order to reach the target number of trees. Note that while a tree is under construction, the algorithm does not include it when evaluating the forest.

As in the initial building phase, when the algorithm completes an evaluation phase and the forest is ready for deployment, then the classification error for the evaluation set is calculated, the evaluation set is reset, and the algorithm enters a test phase to simulate the deployment classification error of the forest. The evaluation set is also reset when a concept change is detected. This ensures that the algorithm evaluates the forest on data with the same distribution of data records as the data on which the forest is deployed.

If the forest is not ready for deployment, the algorithm resumes the update phase
that was stopped upon the arrival of a new block of labelled data records.

If a block of labelled records ends during a test phase, then there is no harm since the update phase is over. The algorithm records the classification error for the test records seen so far, and it then enters a new update phase upon the arrival of a new block of labelled records.

The algorithm’s behavior for handling streams of Scenario 3 is shown in Figure 5.2. Refer to Appendix A for flow charts of the algorithm.

```plaintext
/*building/updating phase*/
while more trees to build
    call BuildTree
    if block of labelled records ends
        if current number of trees ≥ 10% of total number of trees
            and eval cnt ≥ 0.01 * tree_min
            call EvaluateForest to calculate $mg$
            if $mg$ ≥ ($mg_{threshold}$ or $mg'_{threshold}$)
                ignore tree under construction in case of initial building
                or restore old tree in case of update
                use current forest for classification
                reset evaluation set
            end if
        end if
    end if
    continue building trees when a new block of labelled records arrives
end if
if concept drift is detected
    reset evaluation set
end if
end while
```

Figure 5.2: Algorithm’s behavior when a block of labelled records ends
5.1.5 Time complexity

We perform an analysis to find the time complexity of our algorithm. The time the algorithm takes to complete a building phase depends on: 1) the time to build each tree, 2) the time to evaluate the forest, and 3) the time to test the forest.

The time to build each tree is the sum of the times for reading the training records, passing them down the trees, incrementing the counters, performing Hoeffding and Gini tests every \( n_{\text{min}} \) records, performing the entropy test for every entropy window of records, and testing the tree. This time is obviously bounded by the time for reading the records, passing them down the tree, and testing the tree, because other computations are only increments and simple comparisons. Each tree is trained from a maximum of \( 2 \times tree_{\text{min}} \) number of records, and tested using a maximum of \( 0.1 \times 2 \times tree_{\text{min}} \) records. Assuming the trees are balanced, during training and testing a record needs to pass through a maximum of \( \log_2 \text{tree}_{\text{size}} + 1 \) levels to reach a leaf node. The time to build a tree is therefore 
\[
(2 \times tree_{\text{min}} + 0.1 \times 2 \times tree_{\text{min}}) \times (\log_2 \text{tree}_{\text{size}} + 1),
\]
which gives a complexity of \( \mathcal{O}(U \times tree_{\text{min}} \times \log_2 \text{tree}_{\text{size}}) \) for building \( U \) trees.

The time to evaluate the forest depends on reading the records of the evaluation set and passing them down all the trees of the current forest. Since the evaluation set contains a maximum of 1% of the number of read records, the time for evaluating the forest under construction is 
\[
U \times 0.01 \times 2 \times tree_{\text{min}} \times (\log_2 \text{tree}_{\text{size}} + 1),
\]
which reduces to a complexity of \( \mathcal{O}(U \times tree_{\text{min}} \times \log_2 \text{tree}_{\text{size}}) \).

The time to test the forest depends on reading the test records and passing them down all the trees. The number of testing records is constant, so the time complexity for testing the forest under construction is \( \mathcal{O}(U \times \log_2 \text{tree}_{\text{size}}) \).

The time complexity of completing one building phase in our algorithm is therefore
\( O(U * \text{tree}_{\text{min}} * \log_2 \text{tree}_{\text{size}}) \). The complexity can be further reduced to \( O(U * \text{tree}_{\text{min}}) \), since \( \text{tree}_{\text{size}} \) can at most be \( (2 * \text{tree}_{\text{min}})/n_{\text{min}} \), which leads to the value of the term \( \log_2 \text{tree}_{\text{size}} \) to be much smaller than \( 2 * \text{tree}_{\text{min}} \).

### 5.2 Implementation and Experimental Settings

The Autonomic Streaming Random Forests algorithm handles Scenario 3, where the blocks of data have variable sizes and arrive in the system at random times. As for the previous versions of the algorithm, the number of attributes considered at each node for splitting is \( M = \text{int}(\log_2 m + 1) \).

#### 5.2.1 Testing criteria

The main criterion by which we evaluate the Autonomic Streaming Random Forests is the reaction of the algorithm each time a block of labelled data is not large enough to complete a training phase. A training phase could be the initial forest-building phase, or any later update phase of the forest regardless of the presence of concept changes or not.

We simulate the flow of blocks of labelled records through the data by randomly choosing a number of stop points such that each stop point defines the end point of a block of labelled records, and the start of the next block of labelled records. The number or records between any two successive points, therefore, defines the size of the block of labelled-records.

To show the behavior of the algorithm, we record the following:

- The values of \( m_g \), \( m_g^{\text{threshold}} \), and \( m_g^{'\text{threshold}} \), each time the forest is considered
for evaluation. When evaluating the forest based on \( mg_{threshold} \), we set the accepted confidence level for the algorithm, \( conf' \), to the minimum possible, which is 63%.

- The classification errors for records of the evaluation sets to support the decision of the algorithm. If the algorithm decides that the forest is not ready for deployment because \( mg < (mg_{threshold} \) or \( mg'_{threshold} \), then the corresponding evaluation-set classification error should be large to support the forest-unacceptability decision. Otherwise, the classification error for the evaluation set should be in the same range as real noise of the data.

- The confidence values, \( conf \) and \( conf' \), of the algorithm each time the forest is evaluated to support the decisions of the algorithm.

- The classification accuracy of the forest throughout the run of the algorithm by recording the classification errors each time the forest enters a testing phase. We simulate the forest classification error when deployed for classifying unlabelled data by forcing the forest into a testing phase when it is ready for deployment after evaluation. The recorded classification errors are, therefore, either the outputs of the ordinary testing phases (when the algorithm completes any build/update phase), or the testing phases for simulating the deployment of the algorithm. Note that the testing phases used to simulate the deployment phases are not part of the algorithm. They are only used to show the current forest’s behavior if it is deployed.

In all our experiments, we run each experiment twice, one run is based on \( mg_{threshold} \) and the other is based on \( mg'_{threshold} \). Both experiments are run using the same seed.
so that they both have the same stop points and use the same split attributes for each node. This ensures that the comparison of the behavior of the algorithm for the two cases is fair. Note that using $mg'_{threshold}$ results in $conf'$ and $mg'$. 

We also evaluate the Autonomic Streaming Random Forests algorithm by showing that it does not add much extra time for evaluating the forest each time a stop point appears. For this, we measure the average total time that the algorithm takes to complete a full run over a synthetic dataset, and compare it with the total time of running the Dynamic Streaming Random Forests on the same dataset.

### 5.2.2 Testing platform

As for the previous versions of the algorithm, the machine on which we conduct our experiments uses a Pentium 4 3.2 GHz processor and 512MB RAM. The autonomic version of the algorithm is implemented using FORTRAN 77, based on the code of the Dynamic Streaming Random Forests algorithm [9]. Table 5.2 summarizes the work done for extending the dynamic version to the autonomic one.

**Table 5.2: Implementation details of Autonomic Streaming Random Forests**

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Dynamic Streaming Random Forests</th>
<th>Autonomic Streaming Random Forests</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of code lines</td>
<td>2530 lines</td>
<td>3050 lines</td>
</tr>
<tr>
<td></td>
<td>used: 2330 ≡ 92.1% of dynamic version</td>
<td></td>
</tr>
<tr>
<td></td>
<td>altered: 200 ≡ 7.9% of dynamic version</td>
<td></td>
</tr>
<tr>
<td></td>
<td>deleted: 0 ≡ 0% of dynamic version</td>
<td></td>
</tr>
<tr>
<td></td>
<td>new added code lines: 520</td>
<td></td>
</tr>
<tr>
<td>No. of main subroutines</td>
<td>7 subroutines</td>
<td>8 subroutines</td>
</tr>
<tr>
<td></td>
<td>used: 3 ≡ 42.86% of dynamic version</td>
<td></td>
</tr>
<tr>
<td></td>
<td>altered: 4 ≡ 57.14% of dynamic version</td>
<td></td>
</tr>
<tr>
<td></td>
<td>deleted: 0 ≡ 0% of dynamic version</td>
<td></td>
</tr>
<tr>
<td></td>
<td>new subroutines : 1</td>
<td></td>
</tr>
</tbody>
</table>
5.2.3 Data sets

We test the Autonomic Streaming Random Forests algorithm on one synthetic and two real datasets. The number of stop points we use to conduct our experiments is 50 for the synthetic dataset, 15 for one real dataset, and 10 for the other real dataset.

**Synthetic data**

The synthetic dataset on which we base our testing is the same data we used to test the Dynamic Streaming Random Forests algorithm, which is generated using the DataGen data-generation tool by Melli [50]. The dataset contains 7 million records with concept changes. For a detailed explanation of the concept changes throughout the dataset please refer to Section 4.2.3 and Figure 4.2. Just as for the dynamic version of the algorithm, the testing set size is set to 5000 records, and the number of records used to test the tree under construction is $0.1 \times \text{tree}_{\text{min}}$.

**Real data**

We use two real astronomical datasets, extracted from the Sloan Digital Sky Survey (SDSS) [4]. SDSS is a huge survey that offers detailed astronomical information about more than one quarter of the sky, by capturing optical images. It also provides three-dimensional maps of about a million galaxies and quasars.

Our experiments are based on the sixth release of SDSS, DR6, that was made public in June 2007, and available online at [http://www.sdss.org/dr6/](http://www.sdss.org/dr6/). DR6 has more accurate measures than the previous releases of SDSS. The maximum recorded uncertainty for the attributes is about 2% [5].

We extract two subsets, each from a different view of the DR6 database, namely
the PhotoObj view and the SpecPhoto view. PhotoObj is a view that contains all the attributes of each photometric object, whereas the SpecPhoto is a view of joined Spectro and Photo objects that have the clean spectra. Each extracted dataset contains two classes of astronomical objects, namely galaxies and stars, and seven numerical attributes. The datasets are described as follows:

- **PhotoObj dataset.** The first dataset contains five attributes \((u, g, r, i, \text{ and } z)\) that represent original color measurements in the photometric color system \(ugriz\). The two other attributes in this dataset represent the \(RA\) (right ascension), and the \(DEC\) (declination) of each object.

The classes are selected based on the primTarget attribute, which represents the primary target category of the object. The records of the galaxies class (class 1) have a \(primTarget\) value of TARGET\_GALAXY. The records of the stars class represent red, white, and brown dwarfs, but all are marked as class 2. These records are selected with \(primTarget\) values of TARGET\_STAR\_RED\_DWARF, TARGET\_STAR\_WHITE\_DWARF, and TARGET\_STAR\_BROWN\_DWARF.

The extracted dataset contains 500,000 records of class 1 (galaxies), and 41,116 records of class 2 (stars). The ratio between the two classes is obviously unbalanced. Approximately 92% records are of class 1 and only about 8% of the records are of class 2. We, therefore, merge the datasets into one dataset with balanced classes ratios in order to be able to test the algorithm and prove our points. We do this by randomly merging the two datasets by selecting records from class 1 with probability 0.9 and records from class 2 with probability 0.1. Then, we replicate the records from class 2 ten times. The resulting dataset contains 782,730 records, with 371,517 records (48%) of class 1, and 411,160
records (52%) of class 2.

- *SpecPhoto dataset.* The second dataset contains the same five color attributes but corrected to include the effect of the redshift astronomical measure (\textit{dered\_u}, \textit{dered\_g}, \textit{dered\_r}, \textit{dered\_i}, \textit{dered\_z}). It also contains the redshift measure, \(z\), the \(DEC\) attribute, and the \(RA\) attribute.

The dataset is extracted in a way such that we know the point of execution at which a concept drift should be expected. Knowing that the \(RA\) and \(DEC\) values of the Galactic Center of the Milky Way are 266.25° and −28.14° respectively, we extract our dataset based on two main conditions:

1. Objects located in the area with right ascension (\(RA\)) \(\pm 45^\circ\) from the Milky Way Galactic Center. This means that the selected values of \(RA\) are in the range \([221.25^\circ, 311.25^\circ]\).

We specifically select this area to maximize the number of stars in our data, since it is the segment facing in towards the galaxy. See Figure 5.3 for more clarification.

2. Objects located in the area with declination (\(DEC\)) in the ranges \([0^\circ, 30^\circ]\) and \([60^\circ, 90^\circ]\) from the Galactic Center. This results in actual \(DEC\) values in the ranges \([-28.14^\circ, 2.14^\circ]\) and \([32.14^\circ, 62.14^\circ]\). The minimum \(DEC\) value in the view SpecPhoto is, however, −11.25° (16.89° from Galactic Center). The lower range of \(DEC\) value is, therefore, set to \([16.89^\circ, 30^\circ]\) from the Galactic Center, which transforms to actual values of \([-11.25^\circ, 2.14^\circ]\).

We exclude the records with declination in the range \([30^\circ, 60^\circ]\) from the
Galactic Center to have a gap between the two selected areas, and so to ensure that the dataset contains a concept drift. We believe that the gap separates records of the dataset with different kinds of stars and galaxies based on the following reasons:

- Dimmer objects close to the galactic plane are obscured by dust, especially galaxies that lie behind the center of our galaxy. The higher the declination, the fewer dim objects will be obscured, since the angle of capturing the images is moving towards the vertical access of our solar system.

- Most stars lie close to the plane of the galaxy. The area that is close to the galactic center has therefore more stars and less galaxies. The ratio between the stars and galaxies switches gradually as the declination increases, so that the area that is close to the vertical access of our solar system has less stars and more galaxies. Since the two classes (starts and galaxies) do not have clear boundaries to separate them, changing the ratio of the classes might effect the boundaries between the two classes, and so becomes more difficult to learn.

Records are sorted in descending order according to the $DEC$ values. The $DEC$ attribute is then removed from the dataset. Figure 5.4 clarifies our suggested idea.

The classes of this dataset are extracted based on the attribute, $type$, which represents the morphological type classification of the object. It is equal to 3 for galaxies, and equal to 6 for stars. The extracted dataset contains 65621 records (72.3%) of class 1 (galaxies), and 20441 records (23.7%) of class 2 (stars). We,
Figure 5.3: Milky Way top view with selected $RA$ values. Picture taken from http://www.msnbc.msn.com

Figure 5.4: Milky Way side view with selected $DEC$ values. Picture taken from http://www.astrosurf.com
therefore, replicate the records from class 2 three times, resulting in a dataset that contains 61323 records of class 2. The percentages of the class 1 and class 2 of the new dataset are 51.7% and 48.3% respectively. We noticed that the data records in the lower range of \( DEC \) values are not frequent enough to train the classifier after the concept drift is detected. We, therefore, replicate the whole block of data 3 times to produce a final dataset with 158,128 records with the concept drift appearing at record 111,353.

5.3 Results and Discussion

Margin function test

Figure 5.5 shows the margin function values \( \overline{mg} \) and the margin threshold values \( mg_{\text{threshold}} \) and \( mg'_{\text{threshold}} \), throughout the two runs on the synthetic dataset. Note that both runs result in the same behavior since there were no points falling in the gap between the values of \( mg_{\text{threshold}} \) and \( mg'_{\text{threshold}} \), and so the decisions of the algorithm for both runs are the same. This figure, therefore, combines the plots of \( \overline{mg} \) and \( \overline{mg}' \) in one plot. Although the definition of \( mg_{\text{threshold}} \) has stricter bounds than \( mg'_{\text{threshold}} \), the graph of \( mg_{\text{threshold}} \) has lower values. This is because setting \( mg'_{\text{threshold}} \) to 10 for the case of 50 trees (the value of \( mg_{\text{threshold}} \)) results in a confidence, \( conf' \) of less than 50%, which lies in the ignored area of the \( conf' \) graph.

Each mark in all plots (asterisks and squares) represents a stop point of the labelled records flow. Five stop points do not allow the program into evaluation phases. These points appear at records 24872, 1892602, 2439275, 5025766, and 5074009, marked as A, B, C, D, and E respectively. The first point (A) appears at a stage where the
forest did not complete building 10% of the total numbers of trees. The forest was therefore not big enough to be evaluated. The points (B and C) at records 1892602 and 2439275 each appear during one of the test phases of the forest. The algorithm, hence, does not enter an evaluation phase since the forest has been completely updated as the building phase was not paused. It, however, records the classification error it calculated so far during these testing phases. The last two stop points (D and E) at records 5025766 and 5074009 do not cause the algorithm to enter evaluation phases since the sizes of the evaluation sets were insufficient to evaluate the forest.

Figure 5.5: Values of $m_g$, $m_g'$, $m_{g,\text{threshold}}$, and $m_{g',\text{threshold}}$ for synthetic data

The figure also shows that, for the remaining stop points at which the algorithm was evaluated, the value of $m_g$ always exceeds the values of $m_{g,\text{threshold}}$ and $m_{g',\text{threshold}}$, except for three stop points that appear at records 6577271, 6637870, and 6639975, marked as F, G, and H respectively. The algorithm, accordingly, reports that the forest is not ready for deployment for these three stop points. Otherwise, it reports
that the forest under construction is ready for deployment, that is for classifying unlabelled records.

Figure 5.6 shows the margin function values, $\overline{mg}$ and $\overline{mg'}$, and their associated threshold values, $mg_{\text{threshold}}$ and $mg'_{\text{threshold}}$, throughout the runs on the PhotoObj dataset. Seven stop points did not allow the algorithm to get into an evaluation phase since the evaluation sets were not big enough to evaluate the forest. These points are not shown in the graph. The points on the graphs only show the points at which the algorithm enters evaluation phases.

![Figure 5.6: Values of $\overline{mg}$, $\overline{mg'}$, $mg_{\text{threshold}}$, and $mg'_{\text{threshold}}$ for PhotoObj dataset](image)

The two runs give the same behavior until the execution reaches the record 518351, marked as F. At this point of execution, the margin value $\overline{mg}$ does not exceed its threshold $mg_{\text{threshold}}$, whereas $\overline{mg'}$ does exceed $mg'_{\text{threshold}}$. The algorithm, therefore, behaves differently starting from this point.

The run that is based on $mg_{\text{threshold}}$ does not allow the forest to get into an
evaluation phase at point F. In addition, the forest does not enter evaluation phases for the next two subsequent points (marked as G and H), because their $m_g$ values are less than the value of $m_g_{\text{threshold}}$ as well. The run that is based on $m_g^{'}_{\text{threshold}}$, on the other hand, decides that the forest is ready for deployment at the execution points F, G', and H', since the values of $m_g^{'}$ at these points exceed the value of $m_g^{'}_{\text{threshold}}$. Note that the labels F, G, G', H, and H' that appear in any of the later figures with regard to a specific dataset represent the same points of execution.

**Classification errors for evaluation sets**

Figures 5.7 and 5.8 show the classification errors that are calculated using the evaluation sets each time the forest is evaluated, for the synthetic and the PhotoObj datasets respectively. We present these error values to support the decisions that the algorithm takes after evaluating the forest whether the evaluation is based on $m_g_{\text{threshold}}$ or $m_g^{'}_{\text{threshold}}$.

Figure 5.7 shows that the points at which the algorithm decides that the forest is not ready for deployment (points marked as F, G, and H), have a corresponding classification-errors of 61.29%, 36.7%, and 34.95%, respectively, for data noise of only 5%. Obviously, the forest is not ready for deployment at these points since the errors are extremely high compared to the noise of the data.

The errors of the remaining stop points at which the algorithm decides that the forest is ready for deployment are approximately equal to the noise of the data. This demonstrates the appropriateness of the algorithm’s decisions. Some points at the start of the period where the drift of category 2 is simulated (the period from record 1,500,000 to record 6,500,000), however, record higher classification errors than
Figure 5.7: Classification errors for the evaluation sets (synthetic dataset)

Figure 5.8: Classification errors for the evaluation sets (PhotoObj dataset)
expected. The reason behind this behavior was previously explained in Subsection 4.3.2, where early gradual mixing of records from new and old concepts causes the algorithm to consider the small percentage of records from new concepts as noise.

Figure 5.8 shows the evaluation-set classification error for the PhotoObj dataset. As mentioned previously, the behavior of the algorithm for the two runs conducted on the PhotoObj dataset is the same until it reaches the point F. At this point, and its two subsequent points (G and H), the run that is based on \( m_{g_{\text{threshold}}} \) decides that the forest is not ready for deployment since \( \overline{m_g} < m_{g_{\text{threshold}}} \) (refer to Figure 5.6). The evaluation-set classification errors are, however, low. They are even lower than other points at which the algorithm decides that the forest is ready for deployment. This shows that \( m_{g_{\text{threshold}}} \) is too strict, and therefore some of its decisions waste chances to deploy a forest with high accuracy. This decreases the overall efficiency of the algorithm.

For the run that is based on \( m_{g'_{\text{threshold}}} \), the points F, G', and H' have evaluation-set classification errors in the same range of other stop points, which demonstrates that the decision of deploying the forest at these points is appropriate. The behavior of the algorithm for the real dataset, therefore, shows that using \( m_{g'_{\text{threshold}}} \) as the base for the algorithm decisions is more efficient than basing the decisions on \( m_{g_{\text{threshold}}} \).

**Algorithm confidence**

Figure 5.9 presents the confidence values \( \text{conf} \) and \( \text{conf}' \) of the algorithm each time the forest passes through an evaluation phase for the synthetic dataset. It can be seen from the figure that the algorithm is always at least 70% confident when deployed on unlabelled records for both confidence values. The points at which the values of \( \text{conf} \)
and \( \text{conf}' \) are relatively poor (points marked as F, G, and H for \( \text{conf} \), and F, G', and H' for \( \text{conf}' \)) are the cases where the algorithm decided that the forest was not ready for deployment. One of the three points (point F) has a zero confidence of correct classification because the value of \( \overline{mg} \) for this specific point is a negative value (See Figure 5.5).

![Figure 5.9: Confidence values for synthetic dataset](image)

For the graph that represents \( \text{conf} \), the points G and H, have confidence values of 13.75%, and 13.90% respectively. This shows that the definition of \( mg_{\text{threshold}} \) guarantees high confidence, since even if the value of \( \overline{mg} \) is in the range \([0, mg_{\text{threshold}}]\) (which means the average of the correct class votes is greater than all of the other classes), the forest might still have poor confidence of correct classification.

For the graph that represents \( \text{conf}' \), the points G' and H', have confidence values of 53.29% and 53.44% respectively, which are less than \( \text{conf}'_{\text{given}} \) (63%). That shows that the measure, \( mg'_{\text{threshold}} \) does not allow the forest to be deployed unless it has
confidence values greater than the confidence defined by the analyst.

We believe that the definition of \( \text{conf}' \) is more representative of the behavior
of the algorithm. A uniformly random classifier still has the chance of getting the
classifications correct, which should lead to a confidence above zero when \( m_g \) is equal
to zero, and correspondingly, greater confidence when \( 0 < m_g < m_{g\text{threshold}} \). This is
only true when calculating the algorithm confidence using \( \text{conf}' \) instead of \( \text{conf} \).

Figure 5.10 shows the confidence values \( \text{conf} \) and \( \text{conf}' \) of the forest each time it
passes through an evaluation phase for the PhotoObj dataset. The values of \( \text{conf} \)
have a minimum of 58% confidence for the points at which the algorithm has decided
to deploy the forest. The points marked by F, G, and H are the points of execution at
which the algorithm has decided that the forest is not ready for deployment. These
points have \( \text{conf} \) values of 40%, 38%, and 46% respectively. The values here do not
reflect the evaluation-set classification errors shown in Figure 5.8, since the definition
of \( \text{conf} \) has strict bounds.

The values of \( \text{conf}' \) have a minimum confidence value of 66%, which is more
than the value of \( \text{conf}'_{\text{green}} \) (63%). This is why the algorithm decides to deploy the
forest at all evaluation points, which also have good evaluation-set classification errors
associated with them (see Figure 5.8). As for the synthetic dataset, the values of \( \text{conf}' \)
are more representative than the values of \( \text{conf} \) for this dataset.

**Classification accuracy of the forest**

Figures 5.11, 5.12, and 5.15 demonstrate the classification error of the forest each
time it passes through a testing phase for the synthetic, PhotoObj, and SpecPhoto
datasets respectively. The asterisks surrounded by circles in all figures record the
points at which the forest has passed through a test phase after it was evaluated and was ready for deployment (simulated deployment phases). The asterisks surrounded by diamonds in Figures 5.12 and 5.15 are the points of testing the forest after detecting concept-drift.

Figure 5.11 only plots one graph that shows the algorithm behavior for the two runs, since as mentioned earlier, the algorithm decisions for deploying the forest based on $m_g^{threshold}$ and $m_g^{'}_{threshold}$ are the same. Note that the test phases errors for the simulated deployment phases are approximately equal to the evaluation-set classification errors for the points at which the forest is ready for deployment shown previously in Figure 5.7. This confirms that the evaluation-set classification errors are good indicators of the algorithm’s behavior when deployed for unlabelled record classification.

The points that are represented only by asterisks are the ordinary testing phases
Figure 5.11: Classification errors of the forest for synthetic data

Figure 5.12: Classification errors of the forest for PhotoObj dataset
for the forest. The classification errors shown by the figure demonstrate that the algorithm behaves similar to the Dynamic Streaming Random Forests designed for streams of Scenario 1 (see Figure 4.4).

Figure 5.12 shows that the two runs for the PhotoObj dataset result in the same behavior until the point marked by F is reached (record 483,395). At this point, the two plots start behaving differently because of the different decisions of deploying the forest based on the values of $m_g^{threshold}$ and $m_g'^{threshold}$.

The two plots reach a maximum 1.8% classification error. The error reaches its maximum at the point marked by F, and decreases until it reaches its minimum towards the end of the dataset. It can be seen from the figure that there is a tradeoff when using $m_g'^{threshold}$ for deciding to deploy the forest. The forest is deployed with worse classification errors but less unlabelled records are wasted, since the forest is deployed more often than when basing the decision on $m_g^{threshold}$.

Although the algorithm detects many concept drift points (marked by diamonds), the classification accuracy is well maintained throughout the run such that the error rates do not exceed 1.8%. The error rates however fluctuate, reaching a maximum of 1.8% at point F for both plots, and a minimum of 0.05% at point G for the plot based on $m_g^{threshold}$ and 0.1% at point $G'$ for the plot based on $m_g'^{threshold}$. We believe that the error fluctuation is affected by the small drifts in the dataset, which the algorithm has handled.

Since we do not know the expected classification error of the PhotoObj dataset, we set up a small test to find the classification errors throughout the dataset, and compare them to the errors of the autonomic version of the algorithm. We select 1% of the records randomly, and divide the resulting dataset into 8 small sets. Each
small dataset contains 1000 records, except the final dataset, which contains 786 records. We run the standard Random Forests algorithm on each dataset 10 times, and record the classification error for each dataset. Figure 5.13 displays the averages of classification errors for the datasets.

![Classification errors of Random Forests for PhotoObj dataset](image)

The figure shows a graph with a maximum error of 1.2%, and a minimum of 0%. It has a similar structure of Figure 5.12; the peaks and bottoms of both figures appear at about the same points in the sequence. This shows that the recorded classification accuracy of the autonomic algorithm follows a similar trend as the expected classification error of the data.

To better show the algorithm behavior of detecting concept changes for real datasets without losing classification accuracy, we conduct an experiment using the SpecPhoto dataset that we extracted from the SpecPhoto view. The values of $\overline{mg}$ for this experiment exceed $mg_{threshold}$ and $mg'_{threshold}$, each time the forest in considered for evaluation. We, therefore, only consider one graph for the two runs. We believe that the reason of having large values of $\overline{mg}$ is because the dataset is a clean dataset
with corrected attribute values, and so the forest is well trained to classify the records.

Figure 5.14 shows the normalized entropy $H'$, the entropy accumulated average $H_{AVG}$, and the entropy threshold $H_{AVG} + \gamma$, for $\gamma = 0.25$. The sizes of entropy windows for this experiment is 500.

![Entropy values for SpecPhoto dataset](image)

Figure 5.14: Entropy values for SpecPhoto dataset

The figure shows that the algorithm detects a concept drift with a significant entropy value of 0.756 at record 112,000 (point A), which represents the concept drift of the data. Recall that the point of concept drift appears at record 111,353. The figure also shows two other points of concept drift (points B and C). The algorithm does not, however, consider point B as a concept drift point since it appears very close to the drift at point A. We believe that the drift appears at point C as a result of the rapidly increasing dust when capturing the images of the objects close to the plane of the Milky Way.

Figure 5.15 demonstrates the classification errors for the SpecPhoto dataset, which
records a maximum error of 6.95%, and a minimum error of 4.10%. Other research work in classification algorithms have used extracted datasets from the SpecPhoto view of the SDSS [49], and recorded classification accuracies for subsets that include only galaxies and stars of about 95% (classification error of about 5%), for standard classification algorithms. Our incremental algorithm records an average classification error of 6%. This demonstrates that the algorithm is able to deal with real datasets having concept drifts without losing classification accuracy.

![Classification errors of Random Forests for SpecPhoto dataset](image)

**Figure 5.15: Classification errors of Random Forests for SpecPhoto dataset**

**Time for building and updating the forest**

We conduct ten experiments to estimate the total running time of the algorithm over the synthetic dataset having the same stop points for each experiment. Each experiment is conducted with a random selection of the attributes used to build each node. The average total time we record is 32.2 minutes, which is in the same range as the Dynamic Streaming Random Forests algorithm (32.82 minutes) to complete its
execution over the same dataset (see Subsection 4.3.4). This demonstrates that the evaluation phases during which the Autonomic Streaming Random Forests evaluates the forest under construction, do not add any extensive calculations, and the total execution time is not significantly affected.

5.4 Summary

This chapter has proposed the Autonomic Streaming Random Forests algorithm, a stream-classification algorithm that handles Scenario 3. It handles concept changes, and it is clever enough to decide whether or not the current forest should be deployed when a block of labelled records is not big enough to complete a build/update phase. The decision to deploy the forest is based on the average value of the margin function $m_g$ of the forest under construction. Only if the value of $m_g$ is above a threshold ($m_{g_{\text{threshold}}}$ or $m_{g'_{\text{threshold}}}$), is the forest deployed for classification. The values of $m_{g_{\text{threshold}}}$ and $m_{g'_{\text{threshold}}}$ are defined carefully to guarantee classification with high confidence. Each is defined depending on different criteria, with $m_{g_{\text{threshold}}}$ having stricter bounds.

We analyze the algorithm to find its time complexity. The execution time of the algorithm is bounded by the time that the algorithm needs to build/update the forest, since the time the algorithm takes to evaluate and test the forest is relatively small. The time complexity is, therefore, $O(U \times \text{tree}_{\text{min}} \times \log_2 \text{tree}_{\text{size}})$, assuming balanced trees. It can be further reduced to $O(U \times \text{tree}_{\text{min}})$, because $\log_2 \text{tree}_{\text{size}} \ll \text{tree}_{\text{min}}$.

The algorithm is tested on one synthetic and two real datasets. The synthetic dataset contains 7 million records, with simulated blocks of labelled records having random sizes, and different concept drift points. The real datasets are subsets
extracted from the Sloan Digital Sky Survey (SDSS). They represent a two-class problem.

Experimental results have shown the expected performance for synthetic and real datasets. The algorithm was able to decide if the forest was robust enough for classification each time a block of labelled records was not big enough to build/update the classification model. The results also show that using $mg'_{\text{threshold}}$ as a base for deciding to deploy the forest or not, and $conf'$ to calculate the confidence of algorithm decisions are more appropriate than the definitions of $mg_{threshold}$ and $conf$.

Time experiments illustrated that the evaluation phases do not add to the total time of the algorithm. The average total time of the autonomic version is similar to the total execution time of the dynamic version using the same dataset.

In addition, the general behavior of the autonomic algorithm with regard to detecting concept changes and dynamically adjusting the parameters is similar to the dynamic version of the algorithm. This shows that the autonomic version of the algorithm does not lose performance with regard to the previous features.

The main limitations of the Autonomic Streaming Random Forests are:

- The algorithm only deals with numerical or ordinal data; it does not handle categorical data.
- As for the previous versions of the algorithm, the code of the Autonomic Streaming Random Forests algorithm is not optimized.
- The algorithm assumes that the records of the dataset are uniformly distributed among the classes. It does not consider datasets with skewed distributions.
- The algorithm does not deal with missing data.
Chapter 6

Conclusions and Future Work

6.1 Conclusions

Data-stream mining has attracted a significant amount of recent research because many recent applications, such as sensor data analysis, telecommunication data management, and financial applications, produce endless amounts of data records (data streams) that need to be analyzed. It is not usually cost-effective to store all of this data. New algorithms are, therefore, required to deal with data streams.

Data-stream mining algorithms raise several challenges over ordinary mining algorithms. Algorithms must be able to extract all necessary information from the data records in only one pass. The main requirement of stream-mining algorithms is, therefore, the ability to handle streams with high arrival rates incrementally and in an online fashion. Stream-mining algorithms should, in addition, be able to cope with changes that might be caused by concept drift, and be able to produce results at any time after enough records have been seen.

Data-stream classification is one active subject in the area of data-stream mining.
Much of the existing work for stream classification algorithms is based on decision trees. The VFDT algorithm [27], and its extensions [37, 65, 66] are decision-tree based stream classification algorithms that use Hoeffding bounds to stop building each node in the tree. Trees that were constructed using Hoeffding bounds had good classification accuracy when tested. Tree ensembles have also been widely used for stream classification [24, 29, 33, 53, 57, 64, 69, 71]. They are, however, not totally incremental in learning their base classifiers. They learn their base classifiers using standard learning algorithms. Therefore, the overall efficiency of the algorithms might be reduced. In contrast to the standard classification algorithms that have three separate datasets to train, test, and deploy the model, the datasets used for stream classification algorithms might intermix, resulting in the scenarios shown in Figure 1.2.

In this work, we proposed a new incremental stream-classification ensemble algorithm that is designed to handle multi-class classification problems. To the best of our knowledge, our algorithm is the first tree stream ensemble algorithm that builds its base classifiers using an online and incremental learning algorithm. The algorithm assumes datasets that have approximately uniform distribution for data records among all classes. It also assumes that any random snapshot of the data stream represents the distribution of data among classes. We designed our algorithm by extending the Random Forests algorithm [19], a well-known standard classification algorithm, to introduce three data-stream classification algorithms, namely Basic Streaming Random Forests [8], Dynamic Streaming Random Forests [9], and Autonomic Streaming Random Forests, which handle Scenarios 0, 1, and (2 and 3) of Figure 1.2, respectively.

The Basic Streaming Random Forests algorithm transforms Breiman’s Random Forests algorithm to a streaming version. It grows its trees incrementally based on
the ideas of Hoeffding trees [27]. The basic algorithm also defines new parameters to
decide when to stop building a tree and move to building the next tree, and to decide
when to transform frontier nodes when the Hoeffding bounds are not met for a long
period of time.

Experimental results on a fairly small machine using synthetic and real data
showed that the Basic Streaming Random Forests algorithm builds its ensemble using
many fewer data records than other stream-classification algorithms, and performs
with an approximately similar accuracy to the standard Random Forests algorithm,
despite seeing the data records only once. The results also demonstrated that, un-
like other stream-classification algorithms that are designed and/or tested on only
two-class classification problems, our algorithm successfully handles multi-class clas-
sification problems. In addition, the results showed that the algorithm is capable of
processing streams rates of up to $2.8 \times 10^4$ records/sec, with per-record classification
time complexity of $O(U)$, where $U$ is the number of trees in the forest. Limitations
of the Basic Streaming Random Forests algorithm include its inability to deal with
the concept-drift problem, and its inability to alter parameters during a run.

The Dynamic Streaming Random Forests, therefore, extends the basic version in
two main directions: 1) it defines an entropy-based concept-drift detection technique
to be able to handle Scenario 1 with concept drifts, and 2) it dynamically adjusts pa-
rameter values based on the data seen so far. We incorporated ideas from Vorburger
and Bernstein [63] to propose our entropy-based technique for detecting drifts. The
technique incrementally maintains the entropies of a reference window that reflects
the most recent concept of the data, and a current window that reflects the cur-
rent concept. If the two entropies are significantly different (according to a defined
threshold), then a change has occurred.

New parameters are introduced in this version of the algorithm to dynamically control the procedures for building nodes and trees. The algorithm initially builds a defined number of trees based on initial values for the parameters. Trees are each built from a block of labelled records of a different size. The algorithm stops building a tree when its classification error is below a threshold parameter.

For each subsequent building phase, the algorithm dynamically adjusts the parameters, and uses them to replace the trees with the worst accuracy. If there is no concept drift detected, the algorithm replaces 25% of the trees. In the case of concept drift, the number of trees to replace increases according to the significance of the drift, and all other parameters are reset to their initial values.

The dynamic algorithm was tested on synthetic data with different examples of concept drift reflecting the three categories of concept drift explained in Subsection 4.1.3. The results showed that the algorithm behaves as expected. It detected the concept-drift points and it automatically adjusted the parameters, while maintaining a classification accuracy approximately equal to the defined noise of the dataset. The results also showed that on average, the time that the algorithm needs to update the forest, regardless of the presence of concept drift or not, is usually less than the initial forest-building time.

The dynamic algorithms is further extended to the Autonomic Streaming Random Forests algorithm, in order to handle Scenarios 2 and 3. The key feature of the autonomic version of the algorithm is that it is smart enough to decide if the forest under construction is robust enough for deployment when a block of labelled data records is not big enough for completely building/updating the forest.
In addition to the known phases of a typical classification algorithm (training, testing, and deployment phases), we define an evaluation phase, during which the algorithm evaluates the forest based on calculating the forest classification accuracy using a subset of data records (evaluation set), which we collected during the forest building. The algorithm then decides if the forest is ready for deployment or not. The algorithm does not enter an evaluation phase unless the sizes of the forest and the evaluation set are big enough to make a reasonable decision.

The forest evaluation is based on whether the average value of the margin function \[^{19}\] is greater than a defined threshold. We formulated two definitions of the threshold value and tested the algorithm based on both of them. We also formulated two confidence definitions, each associated with one threshold definition, to measure the algorithm confidence each time a decision to deploy the forest is taken. One confidence (and threshold) definition represents a linear relation with strict bounds, whereas the other is a monotonic relation with less conservative bounds.

The Autonomic Streaming Random Forests algorithm is a fast and incremental stream-classification algorithm. Assuming the trees are balanced, it completes a build/update phase with a time complexity of \( \mathcal{O}(U \times \text{tree}_{\text{min}}) \), where \( U \) is the current number of trees in the forest, and \( \text{tree}_{\text{min}} \) is the minimum number of records used for building a tree.

The algorithm was tested on real and synthetic datasets. The synthetic dataset is the same dataset used to test the dynamic version of the algorithm. Experimental results on both datasets demonstrated that the algorithm was able to take the right decision each time a block of labelled records was not big enough to build/update the forest.
The results also showed that the autonomic algorithm does not perform worse than the dynamic version. The algorithm behavior, in terms of detecting points of concept drift and dynamically adjusting parameters, is similar to that of the dynamic version. Moreover, the average execution time of the autonomic version over the synthetic dataset is within the same range of the average execution time of the dynamic version, which demonstrates that the evaluation phases do not add noticeable extra time to the total execution time.

6.2 Future Work

Despite of all the research issues that the Autonomic Streaming Random Forests algorithm covers, the algorithm has still some limitations that can be considered as opportunities for future research. We highlight the main directions in which we believe our work can be extended.

- The algorithm handles data with only numerical or ordinal attributes. Categorical data are, however, common for real datasets in the area of classification algorithms. It is, therefore, essential to consider data with categorical attributes. We believe that this can be easily achieved in two possible ways.

One way is to use the technique that the standard Random Forests algorithm follows to select the best split of categorical attributes. It finds all possible combinations that split the categories of a categorical attribute into two groups, so that data corresponding to each group flow down one branch of the node in the binary tree under construction. This technique might, however, be time consuming. It might also result in trees with large depth.
The other possible way of dealing with categorical attributes is to not restrict
the trees of the forest to be binary trees. The nodes that are associated with
categorical attributes can, instead, have as many branches as the number of
categories (attributes values).

- The algorithm only deals with datasets that satisfy the assumption of having
approximately uniformly-distributed records among the classes. It does not
consider datasets with skewed distributions, which are typical of many real-life
datasets. If the distribution of the skewed dataset is known, then it is straight
forward to extend our algorithm to handle such datasets by assigning different
weights to records from different classes, such that higher weights are assigned
to records from rare classes so they have higher effect for training the model.

If, on the other hand, the distribution of the skewed dataset is not known in
advance, then it is challenging to assign the weights. One possible solution for
this problem is to continuously monitor a window of records and calculate the
probabilities of occurrence of records from each class, and accordingly, assign
the weights to records from each class.

- Although our algorithm dynamically adjusts its critical parameters based on the
records seen so far, there are still some parameters that are assigned constant
values obtained empirically, such as \( w = 25 \) (the number of trees to replace when
there is no concept drift) and the minimum number of trees in the forest before
an evaluation phase can take place (10% of total number of trees). We could
not see a potential way to choose these numbers so we made a pragmatic choice.
We, however, believe that there is room for improvement in this direction so
that the algorithm can dynamically tune these parameters through execution.
• Our algorithm does not deal with missing data. Algorithms in the area of data mining should address this problem since it is a common problem. One way to replace missing values and it is used by standard Random Forests is to replace the missing value of an attribute with the median over all the values of this attribute within the same class. Other ways include replacing the missing values with the average value seen so far, or simply ignore the attributes that has missing values if the number of missing data is small.

• The algorithm assumes that the number of classes is given as an input parameter and it does not change through execution. An interesting problem for stream classification systems, however, is that classes might split (or join) as a result for concept drift, or even a new class is introduced to the classification problem. Our algorithm can be extended to handle this problem by dynamically adjusting the number of target classes through execution.

• The code we implemented for testing the different versions of the Streaming Random Forests algorithm is not optimized. We believe that using an optimized code would allow the code to handle streams with higher arrival rates. One technique to optimize the code is to use parallel programming so multiple trees can be built at the same time, and therefore, the execution time is less. Other techniques involve minimizing the function calls, loop unrolling, and minimizing the number of flow-control statements.
Bibliography


120


BIBLIOGRAPHY


Appendix A

Algorithm Flowcharts

This appendix presents the flowcharts of the Autonomic Streaming Random Forests algorithm. Figure A.1 shows the main flow of the algorithm passing through its major functions. Figures A.2, A.3, A.4, and A.5 demonstrate the flowcharts of the algorithm for building trees, testing the forest, evaluating the forest, and deriving the parameters respectively.
APPENDIX A. ALGORITHM FLOWCHARTS

Figure A.1: Main flow of the algorithm
Figure A.2: Build tree flow chart
Figure A.3: Testing phase flow chart
APPENDIX A. ALGORITHM FLOWCHARTS

### Figure A.4: Evaluation phase flow chart

1. Calculate $\overline{mg}$
2. If $\overline{mg} > mg_{\text{threshold}}$, then:
   - No: Forest not ready for deployment
   - Yes: Forest ready for deployment

### Figure A.5: Parameter derivation flow chart

1. Change flag = 1?
   - No: Calculate new $tree_{min}$, $n_{min}$, and $tree_{\text{threshold}}$
   - Yes: Reset $tree_{min}$, $n_{min}$, and $tree_{\text{threshold}}$ to initial values
2. Calculate number of trees to replace = 25% of total number of trees
3. Calculate number of trees to replace based on change significance

exit