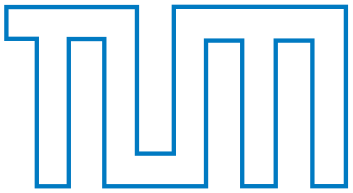
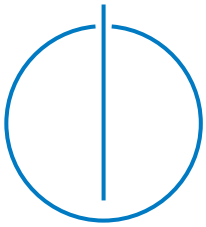


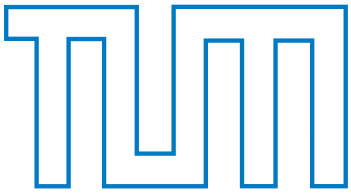
Department of Informatics
Technische Universität München

Master's Thesis in Informatics

Ensemble Learning in Data Streams

Hossain Mahmud





Department of Informatics

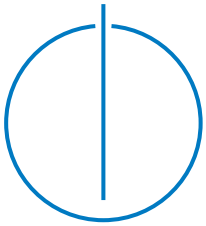
Technische Universität München

Master's Thesis in Informatics

Ensemble-Lernen in Datenströmen

Ensemble Learning in Data Streams

Author:	Hossain Mahmud
Supervisor:	Prof. Dr. Burkhard Rost
Advisors:	Dr. Eirini Ntoutsi
	Dr. Lothar Richter
Submission Date:	October 26, 2015



I confirm that this master's thesis is my own work and I have documented all sources and material used.

October 26, 2015
Munich, Germany

Hossain Mahmud

In the ecstasy of creation today
Laughs my face, smile my eyes
Glows my boiling blood
In the brook of my shuttered soul
The roaring tide brings the flood.

The Ecstasy of Creation
Kazi Nazrul Islam

Acknowledgments

I would like to give my regards and thanks to all the persons who have become a part of this thesis.

First of all, thanks to the Almighty for giving me the strength to finish this thesis.

To my advisor, Dr. Eirini Ntoutsi, for her constant guidance, teachings, directions, corrections, supports, advices, instructions, inputs, and time to this thesis.

To my advisor, Dr. Lothar Richter, for his guidance, directions, corrections, inputs, and time not just to this thesis, but also for teachings during Protein Prediction, and Data Mining courses.

To my supervisor, Professor Dr. Burkhard Rost, for his insights into the thesis and for being an inspiration. Most importantly, for agreeing to be my supervisor when I wanted to do my thesis at LMU.

To Professor Dr. Hans-Peter Kriegel, who is one of the reasons, I looked at TUM and LMU while choosing my graduate university. I am proud that I get to work in two of the world's finest research groups.

To my former supervisor, Dr. M. Eunus Ali, for his guidance in choosing a thesis topic.

To Christian Schulte zu Berge, for his support during my study at TUM.

To my parents, bothers, sister-in-laws, and my nephews, for sending the love and support being a continent away.

Lastly, to my friends here and abroad, especially Tanim, Ninoy, Charza, and Akram for, mostly, food and Euro trips.

Abstract

To handle the enormous amount of data being produced by various applications these days, stream mining algorithms have been developed where the assumption, made in batched approaches, of data being sampled from a stationary distribution is relaxed. However, most of these stream mining approaches assume that sources of these data within the stream contribute evenly to the stream. Many of the modern applications violate this assumption. Many streams are decomposable to a number of sub-streams with mutually exclusive sources. Some of these sources produce very high volumes of data in a short period of time, while other produces a significant amount of data over a longer period. Each source may produce data for all or some classes. As the learners try to update themselves with the most recent data, they often loose information about these slow but important sources. In this thesis, we investigate various aspects of such a setup for Hoeffding tree based learners. We also extend an existing algorithm to introduce our size restricted Hoeffding tree (SRHT), to be used with our carry-over bagging (CoBag) which is devised from online Oza bagging approach. With extensive experimental studies, we show that the new approach is more stable for learning streams that can be decomposed into speed-varied sub-streams.

Contents

List of Figures

List of Algorithms

List of Tables

List of Code Listings

Chapter 1

Introduction

Huge amounts of data are generated in an unprecedented rate now-a-days from different application domains like social networks, telecommunications, WWW, scientific experiments, e-commerce systems, health care systems, etc. These data flow in and out of systems continuously, but with varying update rates. Banking systems register each of their ATM and account transactions, telecommunication providers log each of their calls' information, popular websites maintain their user logging details, organizations like CERN produce petabytes of data during their experiments; these are known as stream data. The volume and the rate of incoming data make it nearly impossible to mine these data with traditional data mining approaches. As alternatives, mining a subsample of available data or to mine for models much simpler than what the data might support can be performed. This, however, drastically limits the ability to extract information from the data. Moreover, in some cases, accumulating and storing the data during runtime and performing a sampling to apply mining algorithms simultaneously is a challenge. For such reasons the notion of mining fixed-size database is slowly being replaced with the idea of open-ended data streams.

Compared to the classical data mining approaches stream mining is relatively a newer topic to be addressed in literature. Even though for batched approaches both classification and clustering problems have been vastly studied, their stream adaptations remain a challenge due the restrictions imposed by the stream data. Due to the volume and the nature of the stream data there are several restrictions that are needed to be considered while designing a stream mining algorithm. Most important limitations include not being able to store the complete data set in memory, and hence only being able to use an example once to train the model; evolving nature of the data with time changes, etc. Thus stream mining requires a different approach than the traditional batch learning. The possibility of temporal locality makes the classification problem even harder in a streaming environment. Algorithms need to address the evolution of the underlying data stream.

Traditional machine learning algorithms generally feature a single model or classifier such as Naïve Bayes or multilayer perceptron (MLP) learned from a training set and use it to classify a testing set. The free parameters of these learners (e.g. weights of a feed-forward neural network) are set by realizing the complete training set. These classifiers

provide a measurement of the generalized performance, i.e. how well the classifiers generalize the training set. Some of the assumptions these algorithms make are: (i) data are finite, (ii) underlying data regularities are stationary, (iii) stationary data sources generate independent and identically distributed data, (iv) data are invariant to the spatial or temporal changes, etc. None of these assumptions is valid for data streams. Stream data exhibit the following characteristics:

- Data come as a continuous flow of an unlimited stream, often at a very high speed.
- Underlying models in the data evolve over time.
- Data cannot be considered to be independent and identically distributed.
- Time and space can significantly influence data.

At the first glance, it may seem that some simple adaptation of current methods should be sufficient to address these changed conditions in data. In reality, these changes challenge most trivial learning methods of machine learning. For example, the computation of entropy of data. For batched learning, all the instances and their corresponding classes are known before-hand to compute entropy. However, for the streaming case, a data stream is no longer finite, the number of classes are not known a priori, and the domains of variables are not necessarily small in size. Hence, the computation of batched-like entropy function is not possible. Similarly, maintaining a frequent item set in a hundreds of gigabytes of database also cannot be easily derived from traditional machine learning algorithms.

Typically, adaptations of traditional algorithms need to address the continuous flow of data, the dynamic environment of generating sources, and the unavailability of a complete data set, etc. Following are some of the new requirements that are needed to be considered, while developing a knowledge discovery system for stream data:

- Algorithms should use limited resources in terms of power, memory, and processing time.
- Algorithms should only access the data a limited number of times, and may only use a limited bandwidth.
- Algorithms should be ready to predict *anytime*.
- Data gathering and processing might be distributed.

With traditional algorithms, a single model is learned, i.e. only one single generalization of the data is learned. However, given a finite set of training examples, it is rather reasonable to assume that the data might contain several generalizations. For example, a different initial setting of a neural network classifier (weights, node layers, node counts, etc.) might change the final network to some extent. For stream environment this assumption becomes primitive. Thus, choosing a single classifier is not always optimal. Using the best classifier among several classifiers, where each is trained with the same training set, would be an alternative; however, information is still being lost by discarding sub-optimal

options. A better alternative would be to build a classifier ensemble. Ensemble classifiers combine the predictions of multiple base level models built on traditional algorithms. A simple process for combining predictions would be to choose the decision based on majority voting. As demonstrated in several works [Breiman, 1994, Schapire, 1990] such ensemble methods (e.g. ensembles of neural networks) yield better performance.

Without proper selection and control over the training process of the base learners, ensemble classifiers could result in poorer performance. Simply choosing a base classifier and training it for several settings would surely produce highly correlated classifiers which would have adverse effect on the ensemble process. One solution of this issue is to train each classifier with its own training set generated by random sampling over the original one. However, with random sampling each classifier would receive a reduced number of training patterns, resulting in a reduction in the accuracy of the individual base classifiers. This reduction in the base classifiers' accuracy is generally not recovered by the gain of combining the classifiers unless measures are taken to make the base classifiers diverse.

Diversity is generally achieved by making the base classifiers minimally correlated. In recent years, various methods have been developed to address this issue. Among others bagging, boosting, or Hoeffding Tree based approaches are mentionable. Some of these approaches are suitable for label based classification, and some can be used to approximate the trend of data over a specific time granularity. It would, however, be nice if a method can approximate the trend of data over a set of time granularity. Moreover, investigating how ensemble methods perform where base classifiers are trained to predict one specific label should be interesting. For example, given a twitter stream an ensemble of classifiers that may contain base classifiers to classify sentiments for sports, politics, entertainment, etc. separately. It is expected that this ensemble setting would out-perform the generic ensemble approach for the complete stream altogether.

Ensemble methods build model outputs where abstract properties of the algorithms that produces the model are prioritized rather than the specifics of the algorithms. This allows a wide application across many fields of study. With precise use of ensemble methods, it would be possible to automatically exploit the strengths and weaknesses of different machine learning systems.

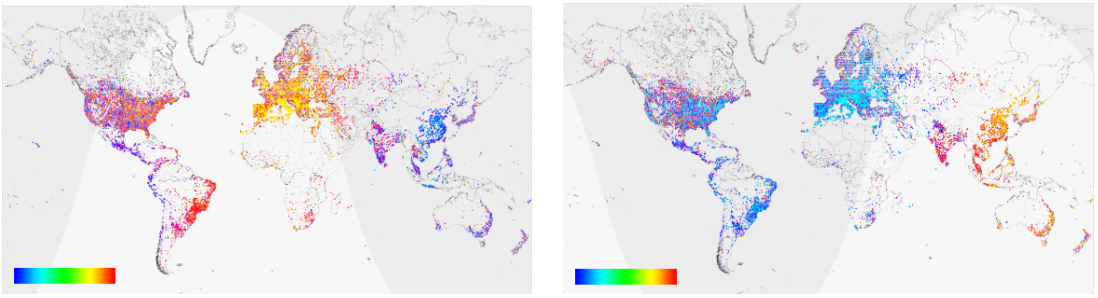
This thesis investigates such currently available ensemble approaches, and presents an empirical analysis of those methods. Moreover, this thesis presents a unique perspective in respect to the underlying setup that generates the stream, which applies to certain application domains such as social media. In following sections, we present this motivation and probable approaches to address such scenarios.

1.1 Motivation

One of the major challenges faced in stream mining is the lack of labeled data. There is no such problem in batched leaning. Data are finite and only, if any, an insignificant portion might be unlabeled. Streaming scenarios show a stark contrast. Most of the real world data are (mostly) unlabeled. It needs human intervention, resources, and time to properly

label a stream data set. Most often only a fraction of the data set is labeled by human experts or automated scripts are used. Because of such limitations, a large number of experimentations for stream mining algorithms are performed using synthesized data. It is much easier to control different parameters like concept drift, labeling, etc. in a generated data set. In most of the processes, this data generation process is mostly randomized, and the originating of data generating from a certain region remains the same for the entire hyperspace. Temporal locality is sometimes created by adding bias to certain region, however, the rates at which these regions produce data points mostly remain the same for all the regions in the hyperspace.

In reality many practical scenarios do not follow a uniform distribution for data generation. Some regions are more active than others. The term "more active" used here is in the sense that they produce more data. For example, Figure 1.1 shows the Internet traffic on an average day. A reference heat-map scale shows the blue color to be less than average and red to be higher than average. As the figure shows based on the time of the day, the amount of network traffic in each region changes drastically even though the number of active nodes remains almost the same.



(a) (b)

Figure 1.1: Worldwide Internet traffic [Botnet, 2012]. Day time traffic in (a) western (b) eastern hemisphere

In another example, let us consider profiling of cell phone users based on the phone usages. Typically, young people are more inclined towards using data services than voice services while the professional and elderly people mostly rely on voice services. Thus, the profiling algorithm should consider this differences in rates in which different user groups are using the services.

To get a clearer picture of the locality of data activation and rate in which data are generated, let's consider the social media statistics. As of the second quarter of 2015, Facebook had 1.49 billion monthly active users. In the third quarter of 2012, the number of active Facebook users had surpassed 1 billion. Active users are those who logged into Facebook during the last 30 days. It was only on the August 28, 2015, that Facebook had 1 billion users on a single day. Let us assume that we want to do a sentiment analysis over the data set collected from a week of data from Facebook. Even though the class distribution (positive and negative sentiments) may remain balanced, however, more active users will clearly overshadow the inputs given by the less active users. Similarly, about 300 million

users produce 500 million tweets per day on Twitter. Figure 1.2 shows tweets for 5 different hashtags namely #Trump, #Syria, #football, #Volkswagen, and #eclipse for the period of Aug 29, 2015 - Sept 28, 2015 ([Topsy, 2015]). As it can be seen, #football has a weekly

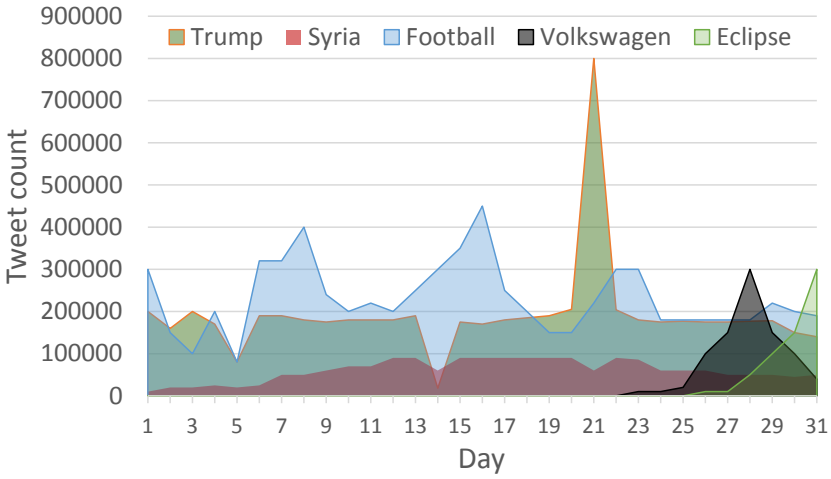


Figure 1.2: Number of tweets for different hashtags

repetitive trend, most certainly because of weekend games. #Trump (USA's presidential candidate for 2016), on the other hand, has a steady rate with occasional spikes depending on some highlighted events e.g. debate. Topic like #Syria has lower rate, however, has a steady flow. Owing to the news of carbon emission from Volkswagen cars and lunar eclipse of September 28, 2015, topics like #Volkswagen and #eclipse got trending; which can be expected to disappear soon. If each of these topics is to be treated equally, slower streams like #Syria would suffer from lack of data volume.

Based on the discussion above, it is evident that these streams do not necessarily follow a uniform distribution. Rather, these streams can be considered to be a collection of many sub-streams. Some properties of these sub-streams are follows:

- A set of slow sub-streams generating low but relatively consistent number of data.
- A set of alternating sub-streams generating moderate number of data. Activation of sub-streams within the set is dependent on each other.
- A set of sub-streams that produces a very large amount of data but only remain active for very short period of time.

Applying machine learning methods in such streams without considering the differences in the rate of data incoming from different sub-streams might lead to a set of decision rules dominated by the sub-streams with the highest number of instances. Moreover, most stream mining algorithms only keep track of most recent incoming instances and forget older data. Such algorithms would thus forget important decision rules learned over longer periods of time for slow but consistent sub-streams when a heavy burst of occasional sub-streams arrives. It would take a long time to learn the same rules again. Similarly, a burst could lead to the deletion of decision rules for recurring sub-streams too. This

could significantly affect the performance measures. Even for slow sub-streams overall performance measures may not reflect poor decision performances for those slow sub-stream, as the number of instances belonging to those streams are not high. But ideally it is expected that the mining algorithms should be equally effective for the entire data space.

In this thesis, we address this scenario. To the best of our knowledge no previous work has specifically addressed this issue. In most works, evaluation are always performed with randomized streams. Concept drift, concept evolution, concept recurrence, etc. are addressed within the randomized streams. We extend one such data generation algorithm to implement the scenario presented above. Empirical evaluations are performed to compare the performances of existing algorithms for such streams. Comparing the results with the results from general randomized streams, it is found that existing algorithms do not perform at their best for streams with high temporal locality. Thus, this thesis presents an ensemble algorithm derived from one of the state-of-the-art algorithms to improve its performance. Extensive evaluation shows that new adaptation achieves more stable outcomes in overcoming the challenges posed by the nature of the stream. It also retains all the positive factors of the original algorithm.

The next section discusses the basic idea of the algorithm without going into mathematical details. We presented the algorithm later in great detail once the related literature and the concepts are introduced in the next couple of chapters.

1.2 Intuition

The central idea of our method is based on the primal decision tree adaption by Domingos and Hulten [Domingos and Hulten, 2000] for streams named Very Fast Decision Tree (VFDT), also commonly known as Hoeffding Tree (HT). Their method is based on the assumption that to find the best attribute for a split decision at any node in a decision tree, it would be sufficient to consider only a certain amount of training data on that node. The Hoeffding bound [Hoeffding, 1963] is a measurement for the degree of certainty for such an approach, which gives an error bound for a decision taken after seeing a certain amount of instances. This bound essentially states that any decision taken after observing a certain amount of instances would remain the same after seeing an infinite number of instances and the error margin can be computed with this bound. A detailed discussion of this bound and the algorithm is presented in Chapter 2.

With such an approach, challenges posed by high volumes of data can be avoided. It does not require remembering all the observed data instances. Rather, the statistics of the instances are sufficient to effectively decide about the split attribute. One of the limitations of a Hoeffding Tree is that it grows linearly as the time passes or instances arrives at the system. Root is the node that is used for splitting the oldest set of examples. Similarly, decision rules at lower levels also become old, while the leaves contain the information from the most recent instances. To keep the rules updated numerous approaches such as resetting the tree, pruning bad performing nodes, etc., have been proposed. Detail

discussion of adaptations for concept drift, evolution, recurrence, etc. are discussed in Chapter 2.

With the reset and pruning facility a Hoeffding Tree exhibits a special property. It always adapts itself for the newer examples. The number of examples a HT's decision rules depend on is directly related to the size of the tree and the number of examples required to split a node. The number of decision or split nodes could be a measurement of the size of the tree. A smaller tree would adapt faster to the changes in the data. A larger tree would take longer time to adapt. Using this rationale a bagging method based on different sized trees is proposed by Bifet et al. [Bifet et al., 2009]. In this method, a fixed number of Hoeffding Trees with different size (number of decision nodes) limits are used. Each time a tree exceeds the size threshold, the tree gets reset. Trivially, smaller trees would reset more often than the larger trees. And thus decision rules from smaller trees will base on the most recent data. Larger trees, however, would also have decision rules from older data.

We combined all these concepts to address our problem. First, we introduced a size restricted variant of Hoeffding Tree named Size Restricted Hoeffding Tree (SRHT). Unlike its predecessor Adaptive Size Hoeffding Tree [Bifet et al., 2009], it does not get reset immediately after it reaches the size limit. Instead, it stops growing i.e. no further split occurs, however, the weights in the leaf nodes are updated with incoming instances. It also combines two different concepts introduced by its predecessors: (i) to reset once a size limit is reached and (ii) to maintain alternate trees or subtrees where necessary. Thus, even after reaching the size limit, a tree can switch to an alternate tree and start growing again till the limit is reached again.

We use this setup for a modified bagging scheme introduced in [Bifet et al., 2009]. Similar to the Hoeffding Tree, a smaller SRHT will have decision rules for most recent data, and a larger tree would contain rules for some older instances, too. In our problem's context, a smaller tree would have decisions for high speed sub-streams or burst of incoming streams. Whereas only the larger trees will have some decision rules for slow but consistent sub-streams, along with the decision rules for most recent data. This is because, the recent data are always dominated by high speed sub-streams. Smaller data portions do not contain enough information about the slow streams or enough examples from slow streams to decide on them. Keeping this in mind, we control the reset of larger trees to keep hard-learned decision rules. In doing so, we maintain an alternate pool of trees that are to be reset soon, and start maintain a new tree with same size limit from scratch. For the transition time we consider votes from all the trees, thus effectively increasing the weights for slower streams. In Chapter 3, we present more elaborate description of the algorithms.

1.3 Related Works

Compared to data mining, stream mining is a particularly young area. Most of the methods date back only a couple of decades. Similarly, the concept of ensemble learning was first

introduced in the traditional batched learning, however, ensemble adaptations for streams are fairly new concepts. In this chapter, we list the most influential methods developed so far with particular focus on tree based learners. We follow a rudimentary narrating style starting directly with methods for general stream learners for stationary streams, and then for evolving data. After that, we list the base methods for ensemble learning, and finally present ensemble based stream learners.

1.3.1 Stream Mining

Domingos and Hulten introduced a strict one-pass adaptation of decision tree [Breiman et al., 1984, Quinlan, 1993] approach in streams. Classic approaches like ID3 and C4.5 learners assume that all training examples can be stored in the main memory altogether. This is a significant limitation to the number of examples these algorithms can handle. Similarly, disk based decision tree learners (SLIQ [Mehta et al., 1996], SPRINT [Shafer et al., 1996], etc.) become very expensive when data sets are very large and the expected trees has many levels. Domingos and Hulten proposed *Very Fast Decision Trees (VFDT)* [Domingos and Hulten, 2000] that uses the Hoeffding bound [Hoeffding, 1963] to build an anytime decision tree for constant memory and time. The primary assumption in this approach is that to find the best attribute for a node in a decision tree, it may be sufficient to consider only a fraction of the training set that passes through that node. The Hoeffding bound provides a statistical measure to determine how much data is needed to ensure a level of degree of certainty, i.e. the error margin would be bounded by a given value [Catlett, 1991].

Like most statistical and machine leaning algorithms VFDT assumes that training data is randomly drawn from a stationary distribution. This assumption is not valid for large databases and data streams. Over time an underlying method or environment that generates data could change. The shift is referred to as *concept drift* in literature and can be abrupt as well as very slow. Data related to weather forecast, economic condition prediction, mis-calibrated sensors, etc. are examples of a concept drifting environment. A concept-adaptive variant of VFDT, *CVFDT* [Hulten et al., 2001], can handle such scenarios. CVFDT updates its decision rules, essentially the tree structure, by detecting the concept drift in the data. It maintains alternate subtrees whenever an old subtree becomes questionable, and replaces the old one with the alternative when it becomes more accurate. CVFDT uses a sliding window and updates sufficient statistics by increasing the count of newly arrived examples and decreasing the count of old examples in the window. Essentially CVFDT achieves same accuracy that would be achieved if VFDT had been run again with the new data. CVFDT does this in $O(1)$ with an additional space requirement as compared to the VFDT's $O(w)$ where w is the window size. Another extension of VFDT, VFDTc was proposed in [Gama et al., 2003] that improves VFDT by handling continuous numeric attributes and adding naïve Bayes prediction at the leaves.

1.3.2 Ensemble Learning

Traditional machine learning algorithms generally feature a single model or classifier such as *naïve Bayes* or *multilayer perceptron (MLP)*. The free parameters of these learners (e.g. weights of feed-forward neural network) are set by realizing the complete training set. These classifiers provide a measurement of the generalization performance i.e. how well the classifier generalizes the training set. However, given a finite set of training example, it is rather reasonable to assume that the data might contain several generalizations. For example, a different setting of the neural network classifier (weights, node layers, node counts, etc.) changes the final network to some extent. For stream environment, this assumption becomes trivial. Thus, choosing a single classifier is not always optimal. Using the best classifier among several classifiers where each is trained with the same training set would be an alternative, however, information is still being lost by discarding sub-optimal options. A better alternative would be to build a classifier ensemble. Ensemble classifiers combine the prediction of multiple base level models built on traditional algorithms. A simple process for combining the predictions could be to choose the decision based on majority voting [Parhami, 1996]. As demonstrated in several works [Breiman, 1993, Schapire, 1990, Wolpert, 1992], ensemble methods (e.g. ensembles of neural networks) [Hansen and Salamo, 1990, Tumer and Ghosh, 1999] yield better performance.

Without proper selection and control over the training process of the base learners, ensemble classifiers could result in poorer performance. Simply choosing a base classifier and training it for several settings would surely produce highly correlated classifiers which would have adverse effect on the overall ensemble process. One solution of this is to train each classifier with its own training set generated by random sampling of the original one. However, with random sampling each classifier would receive a reduced number of training patterns, resulting in a reduction of the accuracy of the individual base classifier. This reduction in the base classifier accuracy is generally not recovered by the gain of combining the classifiers unless measures are taken to make the base classifiers *diverse*. Classifiers with complementary information would give the lowest correlation [Breiman, 1993, Tumer and Ghosh, 1999]. Many methods have been proposed to promote diversity among the base classifier: *bagging* [Breiman, 1994], *boosting* [Drucker et al., 1994, Freund and Schapire, 1997, Tumer and Oza, 1999], *cross-validation partitioning* [Krogh and Vedelsby, 1995, Tumer and Ghosh, 1999], etc. These methods mainly process the entire training set repeatedly and require at least one pass for each base model. This is not suitable for streaming scenarios. Stream adaptation of bagging and boosting methods has been introduced by Oza et al. [Oza and Russell, 2001, Oza, 2001].

1.3.3 Ensemble Learning in Streams

Learning algorithms in data streams require maintenance of a set of hypotheses based on the training instances seen thus far with the need for storage and reprocessing. Facilitating

this requirement, Oza and Russell developed an online version [Oza and Russell, 2001, Oza, 2001] of traditional bagging and boosting. Bagging works by randomly sampling with replacement from the training set to form a given number of intermediate training sets which are used to train same number of classifiers. During testing a majority voting scheme is employed on the decisions of all classifiers to deduce the final decision. Boosting uses an iterative procedure to adaptively change the distribution of training data by focusing more precisely on misclassified instances. Initially all instances have equal weights, and at the end of a boosting round the weight of each instance is updated by increasing or decreasing if the instance was classified wrongly or correctly, respectively. For the *online* variant of these algorithms, not knowing the size of the training data poses a problem in determining the size of training sets to build the base models. In [Oza and Russell, 2001] the authors address this situation by training k models with each instances where k is a suitable Poisson random variable. Later on, [Pelossof et al., 2008] proposed the *Online Coordinate Boosting* algorithm where the number of weight updates of [Oza and Russell, 2001] is reduced using few simple alterations.

As mentioned in the previous section, a *Hoeffding Tree (HT)* i.e. VFDT [Domingos and Hulten, 2000], can be used to build classifiers for concept drifting streams. The Hoeffding Tree has the property that it adapts itself for the newer examples. The number of examples that a HT is built upon is determined by two numbers: (i) the size of the tree, and (ii) the number of examples used to create a node. Thus, smaller trees adapt faster to the changes in the data, while larger trees try to retain the rules that reflect longer time-frame, simply because they are built on more data. In other words a tree bounded by size n would be reset twice as often as tree bounded by size $2n$. *Adaptive Size Hoeffding Tree (ASHT)* bagging [Bifet et al., 2009], uses this intuition to build an ensemble of classifiers of different sized Hoeffding trees. ASHT bagging attempts to increase the diversity in the bagging approach. The maximum allowed size for the n-th tree is twice the size of the $(n-1)$-th, where the first tree has a size of 2. Additionally, the inverse of the squared errors have been used as the weights for the trees. Authors made an observation that bagging using 5 trees of different size might be sufficient to gain higher accuracy, as the error level for bagging with more trees does not improve much but takes longer time.

Same authors also proposed an adaptive window size bagging method– *ADWIN* [Bifet et al., 2009]. ADWIN automatically detects and adapts to the current rate of change. To do so ADWIN adapts its window size to maximize the statistically consistently length that conforms following hypothesis "there has been no change in the average value inside the window". The window is not maintained explicitly, rather using a variant of a exponential histogram technique that takes $O(log\ w)$ memory and $O(log\ w)$ processing time where w is the length of the window. Experimental evaluation showed that ADWIN bagging has better accuracy than ASHT bagging, however, requires more time and memory.

ADWIN has later been used in *leverage bagging* [Bifet et al., 2010c]. Leverage bagging improves randomization by increasing the re-sampling and using output de-

tection codes. Re-sampling with replacement is done in online Bagging using Poisson(1) [Oza and Russell, 2001]. Instead leverage bagging increases the weights of re-sampling using a larger value λ to compute the value of the Poisson distribution. The Poisson distribution is used to model the number of events occurring within a given time interval. Randomization is added at the output of the ensemble using output codes. This method works by assigning a binary string of length n to each class and building an ensemble of n binary classifiers. Each of the classifiers learns one bit for each position in the string. A new instance is classified to the class whose binary code is closest.

ADWIN has also been used in building ensembles of *Restricted Hoeffding Trees* [Bifet et al., 2010a]. A mechanism for setting the learning rate of perceptrons using ADWIN's change detection method is used to restrict the tree. Additionally, a mechanism for resetting the member Hoeffding trees is also been introduced when a particular member is no longer performing well. The method outperforms traditional bagging in terms of accuracy, but requires additional memory and time.

1.4 Thesis Objectives

The primary objective of this thesis is to devise an algorithm realizing the intuitions discussed earlier. In doing so, an extensive survey of existing literatures is performed. The most relevant portion of which has already been presented in the previous section. The remaining is attached at the end of this thesis (see Appendix B). This thesis presents a new way to look at the composition of various real-world streams e.g. social networks. Based on that, a new approach of data generation to facilitate slow, fast, burst, recurrent, etc. behavior in a randomized data stream is proposed. In an effort to combine the ideas of adaptive size Hoeffding trees and adaptive Hoeffding trees using ADWIN, a new ensemble method is devised aiming to improve performance for slower streams. Lastly, a comprehensive empirical comparison of current Hoeffding tree-based approaches is performed and justification of the new approach is performed using synthesized data.

1.5 Outline

The rest of the article is organized as follows: Chapter 2 presents the base concepts required for the algorithm. We first discuss the basic streaming adaptations of batched methods, and later their usages in ensemble approaches. Chapter 3 precisely defines the problem and discusses the development of our algorithm addressing the motivation and intuition of this chapter. In Chapter 4 existing data set generators are discussed first and a new data generation scheme is explained that fulfills the requirements discussed in the intuition section. Finally, before concluding the thesis in Chapter 6, in Chapter 5 we describe the experimental evaluation process and findings. Extensive evaluation is performed to compare existing methods, as well as, method devised in this thesis (Chapter 3).

Chapter 2

Background

This chapter discusses the primitives of data and stream classification. First, there is a section on data stream classification where challenges and approaches for stream classification are introduced. Then, this is followed by an overview of current state-of-the-art ensemble learning methods. These discussions lay the foundation of the approach introduced in this thesis.

2.1 Data Stream Classification

Traditional data mining algorithms work in a memory bounded environment and require multiple scans of the training data. In a stream environment, one of the major assumptions is that new data samples are introduced in the system with such a high rate that repetitive analysis becomes infeasible. Thus, for stream classification, algorithms should be able to look into an instance only once and decide upon that. A bounded memory buffer can be used to facilitate some level of repetition. However, which instances are to remember and which are to forget would then become a decision choice. An alternate choice is to maintain sufficient statistics to have a representation of the data. The process of deletion or aggregation of instances, however, means that some information is being lost over the time.

In this section, first, these challenges are discussed in detail. Then it presents the basis of some of the concepts arisen to handle these challenges. Finally, before moving on to the ensemble learning, it discusses current state-of-the-art algorithms for stream mining.

2.1.1 Challenges

Challenges posed by the streaming environment can be categorized into two groups: (i) relating to runtime and memory requirements and (ii) relating to underlying concept identification. Speed of incoming data, unbounded memory requirement, and single-pass learning fall into the first category. On the other hand, the lack of labeled data, concept drift, evolution and recurrence are examples of the latter category.

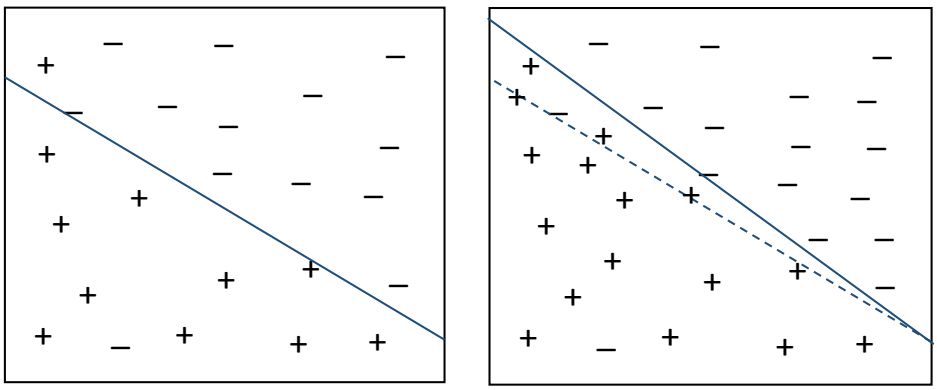
Speed of data arrival: As mentioned before, it is an inherent characteristic of data streams that data arrive with a very high speed. The algorithm should be able to adapt to the high speed nature of streaming information. The rate of building the classifier model should be higher than the data rate. This gives a very limited amount of available time for classification as compared to the traditional batch classification models.

Memory requirements: To apply traditional batched approaches in streaming data, an unbounded memory would be needed. This challenge is addressed using load shedding, sampling, aggregation, etc. Rather than storing all the instances, algorithms store a subset of the data set or some statistical values or a combination of both which represents the data seen thus far. New instances can be classified only by looking into these stored information.

Single-pass learning: The premise of this requirement is two-fold. First, as mentioned above, data would not be available in the memory after a short period of time due to the volume of data. Secondly, even if the data remain available, running a batched-like approach for millions of data points would highly increase the execution time of the algorithm. To attain shorter processing time with limited storage, the algorithm should access the data stream only once, or a small number of times. Mining models must possess the capability of learning the underlying nature of data in a single pass over the data.

Lack of labeled data: Unlike most data sets or settings of batched approaches, stream mining data sets are often poorly labeled. A large number of experimentations are done with generated data sets where the data generation process can easily be controlled to have proper labeling. However, data sets collected from the real world often lack this. For example, to set up a supervised learning experimentation using a data set collected from social media, e.g. Twitter, data need to be first categorized by human intervention. Manual labeling of such data is often costly, both in terms of resources and time. In practice, only a small fraction of data is labeled by human experts or automated scripts. A stream classification algorithm is thus required to be ready to predict after observing a small number of instances, i.e. to be ready to predict anytime.

Concept drift: Concept drift is a statistical property of data streams where the target variable drifts away from the model that is trying to predict it. In other words the underlying data distribution is changing over time. As a result, accuracy of the classifier model decreases over time. For example, the buying pattern of customers in a store changes over time, mostly due to the seasonality. Electrical and mechanical devices wear off over time, producing shifted results which would cause a drift in the observed data. Learning models should adapt to these changes quickly and accurately. Let us consider the example in Figure 2.1. As a new chunk arrives (Figure 2.1a), a new classifier is learned. The decision boundary is denoted by the straight line. The positive examples are represented by pluses while the negative examples are represented by minuses. Over time the concept or label of some of the examples may change. As shown in Figure 2.1b, due to concept

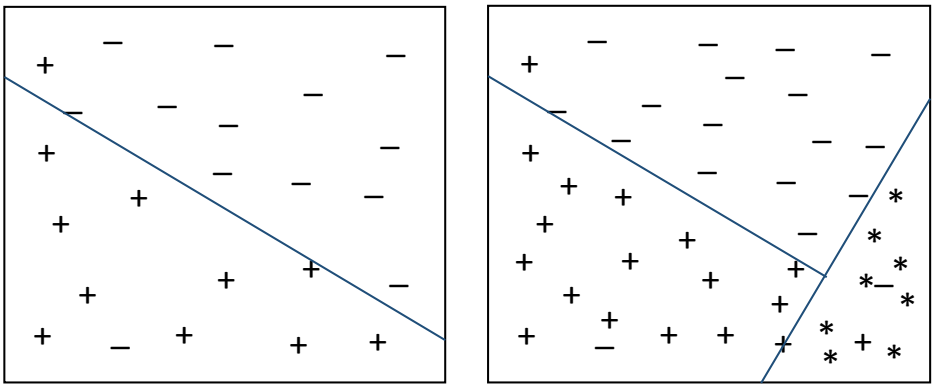


(a) (b)

Figure 2.1: Concept drift in data streams

drift newer positive examples may drift towards the decision boundary and into the negative concept's region. So, the previous decision boundary has become outdated and the model has to be updated. Formally, concept drift is the change in the joint probability $P(X, y) = P(y|X) \times P(X)$. Thus, observing the change in y for given X, i.e. $P(y|X)$ is the key for detecting concept drift.

One challenge posed here is to differentiate between noise in the data and the actual shift in the concept. In streaming environment data often contain both. The rate of the drift is also a factor in detection. Sudden drift, known as *concept shift*, is easier to detect than concept drift, which is considered to be gradual.



(a) (b)

Figure 2.2: Concept evolution in data streams.

Concept evolution: Concept evolution is referred to as the emergence of a new class or a set of classes in stream data as the time passes. Twitter stream is an ideal example where concept evolution is easily identifiable. Twitter reacts, seemingly, very fast upon important news around the globe. Looking into the different hash tag usages in such social media currently trending topics can be identified. To present a clearer picture let us consider the following example in Figure 2.2. For a certain point of time two classes and their

corresponding decision boundaries are shown in the Figure 2.2a. With more incoming data a novel class emerges, and for that, decision boundaries need updating. Emergence of a new class can affect any number of decision boundaries/rules, from one to all. Concept evolution is also prone to noise. Furthermore, clear distinction between drift and evolution might not always be possible, partially due to the lack of unlabeled data.

Class recurrence: Class recurrence is a special case of concept drift and evolution. In this case, the model forgets a class due to the drift or absence of data for some period, however, later the class reappears (evolution) in the stream. Seasonality could be one cause of this situation. An intrusion in network traffic may reappear after a long time. Forgetting the earlier intrusions is not desired in such a case. A fast recognition of the previously seen classes is desired in stream mining.

Following sections discuss the potential solutions to these challenges. First, how to address the limited resources and then the change detection schemes.

2.1.2 Maintaining Sufficient Statistics

In statistical evaluation, a statistic is sufficient for a family of probability distributions if the sample from which it is calculated gives no additional information than does the statistic, as to which of those probability distributions is that of the population from which the sample was taken [Fisher, 1922]. Mathematically, given a set $\mathbf{X}$ of independent identically distributed data conditioned on an unknown parameter θ, a sufficient statistic is a function $T(\mathbf{X})$ whose value contains all the information needed to compute any estimate of the parameters (e.g. maximum likelihood estimate). Using the factorization theorem (Theorem 2.1.1), for a sufficient statistic $T(\mathbf{X})$, the joint distribution can be written as $p(\mathbf{X}) = h(\mathbf{X})g(\theta, T(\mathbf{X}))$. From this factorization, it can easily be seen that the maximum likelihood estimate of θ will interact with $\mathbf{X}$ only through $T(\mathbf{X})$. Typically, the sufficient statistic is a set of functions or random variables of the data.

Theorem 2.1.1 (Factorization Theorem) *Let $X_1, X_2, \ldots, X_n$ be a random sample with a joint density $f(x_1, x_2, \ldots, x_n|\theta)$. A statistic $T = r(X_1, X_2, \ldots, X_n)$ is sufficient if and only if the joint density can be factored as follows:*

$$f(x_1, x_2, \ldots, x_n|\theta) = u(x_1, x_2, \ldots, x_n)v(r(x_1, x_2, \ldots, x_n), \theta)$$

where u and v are non-negative functions. The function u can depend on the full random sample $x_1, x_2, \ldots, x_n$, but not on the unknown parameter θ. The function v can depend on θ, but can depend on the random sample only through the value of $r(x_1, x_2, \ldots, x_n)$.

Bounds of a Random Variable

A random variable is a variable that can take a set or range of values, each with an associated probability, and is subjected to change due to the alteration or randomness of the data. Random variables are of two types: (i) discrete, and (ii) continuous. A discrete

random variable takes a set of possible values (e.g., the outcome of coin flipping), but a continuous random variable can take any value within a range (e.g. age of people in a randomly sampled group).

A function that is used to estimate a random variable is called an estimator. An estimator function is dependent on the observed sample data, and used for estimating unknown population within an interval with certain degree of confidence. For an interval of the true values of the parameter associated with a confidence of $1 - \delta$, the interval can be defined as follows:

- Absolute approximation: $\bar{X} - \epsilon \leq \mu \leq \bar{X} + \epsilon$, where ϵ is the absolute error.
- Relative approximation: $(1 - \delta)\bar{X} \leq \mu \leq (1 + \delta)\bar{X}$, where δ is the relative error.

where μ and $\bar{X}$ represent the actual and the estimated mean. There are a number of theorems that provide bounds on the estimation. The Chebyshev, the Chernoff, the Hoeffding bounds [Hoeffding, 1963], etc. are few of them.

Theorem 2.1.2 (Chebyshev Bound) *Let X be a random variable with standard deviation σ, the probability that the outcome of X is no less than $k\sigma$ away from its mean is no more that $1/k^2$:*

$$P(|X - \mu| \leq k\sigma) \leq \frac{1}{k^2}$$

In other words, it states that no more that $1/4$ of the values are more than 2 standard deviation away, no more than $1/9$ are more than 3 standard deviation away, and so on.

Theorem 2.1.3 (Chernoff Bound) *Let $X_1, X_2, \ldots, X_n$ be independent random variables from Bernoulli experiments. Assuming that $P(X_i = 1) = p_i$. Let $X_s = P_n \sum_{i=1} nX_i$ be a random variable with expected value $\mu_s = P \sum_{i=1} np_i$. Then for any $\delta > 0$:*

$$P[X_s > (1 + \delta)\mu_s] \leq (\frac{e^{\delta}}{(1 + \delta)^{1+\delta}})^{\mu_s}$$

and the absolute error is:

$$\epsilon \leq \sqrt{\frac{3\bar{\mu}}{n} \ln(2/\delta)}$$

Theorem 2.1.4 (Hoeffding Bound) *Let $X_1, X_2, \ldots, X_n$ be independent random variables. Assuming that each x_i is bounded, that is $P(X_i \in R = [a_i, b_i]) = 1$. Let $S = 1/n \sum_{i=1} nX_i$ whose expected value is $E[S]$. Then, for any $\epsilon > 0$:*

$$P[S - E[S] > \epsilon] \leq e^{\frac{2n^2\epsilon^2}{R^2}}$$

and the absolute error is:

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

Chernoff and Hoeffding bounds are independent of the underlying distribution of examples. They are more restrictive or conservative, and require more observations as compared to the distribution dependent bounds. The Chernoff bound is multiplicative and

the Hoeffding is additive. They are expressed as relative and absolute approximation, respectively.

These methods only take a finite number of values or a range. One of the well-known methods which supports infinity is a Poisson process. A random variable x is a Poisson random variable with parameter λ if x takes values $0, 1, 2, \ldots, \infty$ with:

$$p_k = P(x = k) = e^{-\lambda}\frac{\lambda^k}{k!} \tag{2.1}$$

where, λ is both mean and variance, i.e. $E(X) = Var(X) = \lambda$.

Recursive Mean, Variance, and Correlation

Fundamental equations of mean, variance, etc. are not usable for streams as past data points are lost as the time passes. However, their recursive versions can easily be derived. Equation 2.2, 2.3, and 2.4 can be used to recursively compute the mean, the variance, and the correlation respectively.

$$\bar{x}_i = \frac{(i-1) \times \bar{x}_{i-1} + x_i}{i} \tag{2.2}$$

$$\sigma_i = \sqrt{\frac{\sum x_i^2 - \frac{(\sum x_i)^2}{i}}{i-1}} \tag{2.3}$$

$$corr(a,b) = \frac{\sum(x_i \times y_i) - \frac{\sum x_i \times \sum y_i}{n}}{\sqrt{\sum x_i^2 - \frac{\sum x_i^2}{n}}\sqrt{\sum y_i^2 - \frac{\sum y_i^2}{n}}} \tag{2.4}$$

As it can be seen from the equations, maintaining (i) number of observations, n; (ii) $\sum x_i$, sum of i data points; (iii) $\sum x_i^2$, sum of squares of i data points; and (iv) $\sum(x_i \times y_i)$, sum of cross product of X and Y are enough to recursively compute these statistics.

Windowing

Windowing is the process of selecting a subset of the observed instances that would be remembered to be used in the computation of statistics. Where the data set is finite and of limited size, all the instances can be remembered. For streams, this is not possible. Furthermore, computing statistics over all the instances of the past, in streaming environment, would wrongly introduce information of classes that are not currently present. Thus, information of recent past is more important than the entire set. Windowing is categorized in two basic types: (i) sequence based windowing, and (ii) timestamp based windowing.

In sequence based windowing, the sequence is based on the number of observations seen; in the timestamp based approach, it is elapsed time. Landmark windowing and sliding windowing are the two most used sequence based windowing systems.

Landmark Windowing: All observations after a certain start point are remembered. Batched approaches can be thought of as examples of landmark windowing where every instance is remembered from the very first one. As new observations are seen, the size of the window increases. Landmark approach needs periodic/occasional updating to ensure that the statistics are recent.

Sliding Windowing: A fixed length window is moved through the observation set. As a new observation is seen, the oldest observation is forgotten, i.e., when the j-th instance is pushed into the window, the $(j - w)$-th instance is forgotten, where w is the size of the window. A limitation of sliding windowing is that it requires all elements within the window to be remembered, as it needs to forget the oldest observation.

Often it is more useful to learn about the most recent updates with fine granularity and the older ones in a summarized fashion. With this motivation, the concept of tilted-time windowing was introduced [Chen et al., 2002].

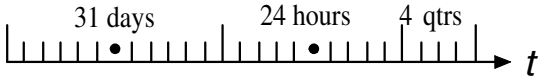


Figure 2.3: Natural Tilted Time Window

Natural Tilted Time Windowing: In natural tilted time windowing, the units of time are distributed non-uniformly. The most recent time gets more units. For example, Figure 2.3 shows a natural tilted time windowing scheme, where for the most recent hour, quarterly updates are stored in 4 units. Similarly, 24 units of time storing a day's, and 31 units storing a month's summary. That is, with 59 units of time information, this model stores about 32 days' information.

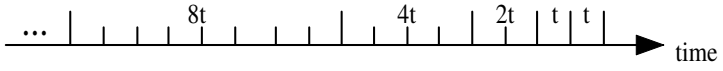


Figure 2.4: Logarithmic Tilted Time Window

Logarithmic Tilted Time Windowing: The concept for logarithmic tilted time windowing is the same as for the natural tilted time windowing. The only difference here is that the time scale grows in logarithmic order. Figure 2.4 shows an example.

2.1.3 Change Detection

An assumption of most machine learning methods is that data are generated from a stationary distribution. As discussed in the previous section, this assumption does not hold for a streaming scenario. Thus stream mining requires algorithms to facilitate drift detection methods. Differentiating between *noise* and *change* makes the problem challenging.

The difference between a new distribution and noise is *persistence*, where new examples consistently follows the new distribution rather than the old. This section discusses several methods to detect and adapt learning algorithms in presence of concept drift.

Detection methods can generally be classified into two categories. The first approach is to monitor the evolution of various performance indicators as done in [Klinkenberg and Renz, 1998, Zeira et al., 2004]. Another approach is to maintain two (or more) distributions varying the window length. Typically, one window would summarize past history while the other would summarize most recent information [Kifer et al., 2004].

Most methods follow the first approach. [Klinkenberg and Renz, 1998] monitors three performance indicators (accuracy, recall, and precision) over time, and uses their posterior comparison to a confidence interval of standard sample errors for a moving average value for each indicator.

A classical algorithm for change detection is *Cumulative Sum (CUSUM)* [Page, 1954]. CUSUM can detect that the mean of the input data is significantly different to zero. The test is as follows:

$$g_0 = 0$$

$$g_t = max(0, g_{t-1} + (r_t - v))$$

If $g_t > \lambda$, CUSUM triggers an alarm and set $g_t = 0$. This detects the changes in the positive direction. To detect the negative change min is used instead of max. CUSUM does not require any memory. Its accuracy depends on the choice of v and λ. Low values for v results in faster detection with more false positives.

For latter category, [Kifer et al., 2004] uses the Cherfnoff bound (Theorem 2.1.3) and examines examples drawn from two probability distributions and decides whether these distributions are different.

ADWIN Algorithm: ADWIN (ADaptive sliding WINdow) is another change detection algorithm of latter type. ADWIN keeps a variable length window of recent items. It ensures the property *there has been no change in the average value inside the window* for the maximally statistically consistent length. The core idea of ADWIN is that whenever two sufficiently large sub-windows of W exhibit distinct enough averages, it is assumed that their corresponding expected values are different, and the older portion of the window is dropped. Essentially, this means that when the difference of means of the two windows is greater than a certain threshold ϵ_{cut}, the older portion should be dropped. Equation to compute ϵ_{cut} is as follows:

$$m = \frac{2}{1/|W_0| + 1/|W_1|}$$

$$\epsilon_{cut} = \sqrt{\frac{1}{2m} \ln \frac{4|W|}{\delta}}$$

where $\delta \in (0, 1)$ is the confidence value, a user input.

ADWIN does not maintain the windows explicitly, but compresses it using a variant

Algorithm 1: ADWIN Algorithm

Data: Data Stream
Result: Window with most recent concept

```
1.1 begin
1.2     Initialize window W
1.3     foreach t > 0 do
1.4         W ← W ∪ {x_t}                    // add x_t to the head of W
1.5         repeat
1.6             Drop element from the tail of W
1.7         until |μ_{W_0} − μ_{W_1}| < ε_cut holds for every split of W
1.8         W = W_0.W_1
1.9     Output μ_W
```

of the exponential histogram technique. This means that it keeps a window of length w using only $O(\log_2\ w)$ memory and $O(\log_2\ w)$ processing time per item.

Adaptation to Change

To improve the accuracy of a decision model under concept drifting environment, the decision model needs adaptation to the changes. There are two types of approaches based on when to adapt:

- Blind or periodic methods: Models are updated in a regular interval whether any change has actually occurred or not.
- Informed methods: Models are only updated when there are sufficient reasons to believe that changes in the concept have occurred.

It could be a good idea to use periodic methods where the duration of the seasonality is known beforehand. Otherwise, the chosen duration could be too large or too small to respond to the changes. In such cases, an informed decision is more desired. However, informed methods require more resources (than blind methods) to be able to make such decisions.

2.1.4 Naïve Bayes Adaptation

The naïve Bayes algorithm is essentially a stream classification algorithm. One of the advantages of this classifier in the context of data streams is its low complexity for deployment. It only depends on the number of explanatory variables. Its memory consumption is also low since it requires only one conditional probability density estimation per variable.

2.1.5 Very Fast Decision Tree

Very Fast Decision Tree (VFDT), also known as Hoeffding Tree (HT), is a decision tree based adaptation for streams generating from a stationary distribution. It is an anytime-ready algorithm and uses the Hoeffding bound to ensure the performance in terms of

Algorithm 2: VFDT: The Hoeffding Tree Algorithm

Input : S: Stream of examples
X: Set of nominal attributes
Y: Set of class labels $Y = \{y_1, y_2, \ldots, y_k\}$
$G(.)$: Split evaluation function
N_{min}: Minimum number of examples
δ: is one minus the desired probability
τ: Constant to resolve ties
Output: HT: is a decision tree

2.1 **begin**
2.2 Let $HT \leftarrow$ Empty Leaf (Root) **foreach** $example(x, y_k) \in S$ **do**
2.3 Traverse the tree HT from root till a leaf l
2.4 **if** $y_k ==?$ **then** // Missing class label
2.5 Classify with majority class in the leaf l
2.6 **else**
2.7 Update sufficient statistics
2.8 **if** $Number\ of\ examples\ in\ l > N_{min}$ **then**
2.9 Compute $G_l(X_i)$ for all attributes
2.10 Let X_a be the attribute with highest G_l
2.11 Let X_b be the attribute with second highest G_l
2.12 Compute $\epsilon = \sqrt{\frac{R^2 \ln(2/\delta)}{2n}}$ // Hoeffding bound
2.13 **if** $G(X_a) - G(X_b) > \epsilon \mid\mid \epsilon < \tau$ **then**
2.14 Replace l with a splitting test based on attribute X_a
2.15 Add a new empty leaf for each branch of the split
2.16 Return HT

accuracy is asymptotically nearly identical to that of conventional tree based algorithms. VFDT runs on constant time and memory per examples and can serve thousands of examples on a typical consumer system.

VFDT constructs the tree by recursively replacing leaves with decision nodes. Each leave stores sufficient statistics that are required to evaluate the merit of the split-test and to decide the target class. For incoming instances, the tree is traversed from the root to a leaf (based on the incoming instance's values) and the statistics are updated accordingly. If a unanimous decision cannot be reached based on observed instances at any particular leaf node, the node is tested to check the sufficiency for a split. If there is enough statistical support in favor of a split, the node is splitted on the best attribute and stats are passed to the descendants (new leaves). The number of descendants of this new decision node is equal to the number of possible values of the chosen attribute. Thus, the tree is not necessarily a binary tree.

Deciding on whether to split a node or not is a unique contribution of VFDT. VFDT solves this difficult problem of deciding exactly how many examples are required to be observed by a leaf node before splitting by using the Hoeffding bound (Theorem 2.1.4). Let $G(.)$ be the heuristic measure of the attributes. This measure could be information gain as of C4.5 or Gini index of CART. For information gain, range R in Hoeffding bound

is lg(#*classes*). The goal is to find a n such that the attribute chosen for the split after observing n instances would, with high probability, be the same as it would be chosen after observing infinite instances. Assume that X_a is the attribute with the best $G(.)$, and X_b is the second best attribute after observing n instances. Then $\Delta G = G(X_a) - G(X_b)$ is the difference between their observed heuristics. From Hoeffding bound, we know if $\Delta G > \epsilon$, then with probability $1 - \delta$, X_a would be the attribute with the highest value in the evaluation function in the universe. Otherwise, if $\Delta G < \epsilon$, then the sample size is not enough to make a stable split decision. As the assumption is that the underlying generating distribution is stationary, thus as the sample size increases, ϵ decreases and the heuristic value for the most informative attribute goes up.

To fasten up the process, VFDT uses an extra parameter N_{min} to reduce the number of $G(.)$ computations. Computing $G(.)$ when there are too few instances is run-time inefficient. Thus, a user parameter N_{min} is used to indicate the minimum number of instances needed to be observed before the evaluation starts. This is known as grace period.

When multiple attributes continuously perform similar in heuristic evaluation after observing a large number of examples, ΔG might never be greater than ϵ. To break such tied cases, another user parameter τ is used, and when ϵ falls below τ (i.e. $\Delta G < \epsilon < \tau$), the algorithm splits on the best attribute. Algorithm 2 summarizes the pseudo-code of VFDT.

There are some properties of VFDT that are different from C4.5. In contrast to the C4.5 algorithm the number of examples that support a decision increases in VFDT. Typically, VFDT also results in a lower variance model than C4.5. However, there is no mechanism to avoid over-fitting in VFDT as there is no option of pruning.

2.1.6 Concept-adapting Very Fast Decision Tree

Concept-adapting Very Fast Decision Tree (CVFDT) is the extension of CFDT that adds the ability to adapt the model by detecting the changes in the underlying distribution that generates the examples. CVFDT does not require the model to be recomputed, instead it updates the sufficient statistics stored in each node that the new instances affects by maintaining a sliding window. It increases the counter of new instances and decreases count of the oldest examples which now needs an instance to be forgotten. If the underlying distribution is stationary, this would not have any effect. But in case of a concept drifting distribution, some decision nodes that previously had passed the Hoeffding bound test will no longer pass, rather an alternate attribute would now have a higher or similar gain. CVFDT keeps statistics to detect such situation and starts maintaining an alternate subtree with the new best attribute as its root. When this alternate subtree becomes more accurate on new data, then the old subtree is replaced by the new one. Pseudo-code of CVFDT has been shown in Algorithm 3. The sub-routine *GROW* is essentially the Hoeffding Tree algorithm where instead of only keeping statistics in the leave, every node keeps track of the instances it has seen. Two other sub-routines *FORGET* and *VALIDATE_SPLIT* have been shown separately.

The *FORGET* function is used to remove the effect of instances of older concepts.

Algorithm 3: CVFDT: Concept-adapting VFDT

Input : S: Stream of examples
X: Set of nominal attributes
Y: Set of class labels $Y = \{y_1, y_2, \ldots, y_k\}$
$G(.)$: Split evaluation function
δ: is one minus the desired probability
τ: Constant to resolve ties
w: Size of the window
n_{min}: Number of examples between checks for growth
f: Number of examples between checks for drift
Output: HT: is a decision tree

3.1 **begin**
3.2 Let $HT \leftarrow$ Empty Leaf (Root) l_0
3.3 Let $Alt(l_0) \leftarrow \emptyset$ // Alternate trees for l_0
3.4 **foreach** *class* y_k **do**
3.5 **foreach** $x_{ij} \in X_i \in X$ **do**
3.6 Set $n_{ijk} = 0$
3.7 **foreach** $example(x, y) \in S$ **do**
3.8 Traverse the tree HT including $Alt(\{n : (x, y)\ passes\ through\ n\ in\ HT\})$ trees root till a set of leaves L
3.9 Set $id = \max(\{L.id\})$
3.10 Add $((x, y), id)$ to the beginning of W
3.11 **if** $|W| > w$ **then**
3.12 Let $((x_w, y_w), id_w)$ be the last element in W
3.13 FORGET$(HT, n, (x_w, y_w), id_w))$
3.14 $W = W - ((x_w, y_w), id_w)$
3.15 GROW$(HT, n, G, (x, y)$
3.16 **if** *examples seen since last checking* $> f$ **then**
3.17 VALIDATE_SPLIT(n)
3.18 Return HT

However, this is not straight-forward as HTs go through changes after the instance to be forgotten was initially added. Thus, CVFDT uses monotonically increasing ids for the nodes. When a new instance is added to the window (W), the maximum id of the leave it reaches in HT and all the alternate trees are recorded with it. To remove the effect of an old instance, counts are decremented at every node the example reaches in HT whose id is less than the stored id.

Lastly, there is $VALIDATE\_SPLIT$ sub-routine that periodically checks the internal nodes of HT where the current split attribute would no longer have the highest $G(.)$ or ΔG would be less than ϵ or ΔG would be less than $\tau/2$. This is a similar condition as to the original one with a more restrictive tie condition. The added restriction ensures the creation of less alternate trees.

Now that related concepts of basic stream mining are discussed, in the next section, the concepts of ensemble learning are presented.

Algorithm: FORGET($HT, n, (x_w, y_w), id_w$)
begin
3.13.1 Sort (x_w, y_w) through HT while it traverses leaves with $id \leq id_w$
3.13.2 Let P is the set of sorted nodes
3.13.3 **foreach** *node* $l \in P$ **do**
3.13.4 **foreach** $x_{ij} \in x : X_i \in X_l$ **do**
3.13.5 Decrement $n_{ijk}(l)$
3.13.6 **foreach** *tree* $t_{alt} \in Alt(l)$ **do**
3.13.7 FORGET($t_{alt}, n, (x_w, y_w), id_w$)

Algorithm: VALIDATE_SPLIT($HT, n, (x_w, y_w), id_w$)
begin
3.17.1 Let $HT \leftarrow$ Empty Leaf (Root) l_0
3.17.2 Let $Alt(l_0) \leftarrow \emptyset$ // Alternate trees for l_0
3.17.3 **foreach** *internal node* l *in* HT **do**
3.17.4 **foreach** $t_{alt} \in Alt(l)$ **do**
3.17.5 VALIDATE_SPLIT($t_{alt}, n, (x_w, y_w), id_w$)
3.17.6 **foreach** $x_{ij} \in X_i \in X$ **do**
3.17.7 Set $n_{ijk} = 0$
3.17.8 Let X_a be the current split attribute
3.17.9 Compute $\Delta(G) = G_l(X_n) - G_l(X_b)$ // X_n, X_b two current highest attributes
3.17.10 **if** $\Delta(G) \geq 0 \&\& X_n \notin \{rootsofAlt(l)\}$ **then**
3.17.11 Compute $\epsilon = \sqrt{\frac{R^2 \ln(2/\delta)}{2n}}$ // Hoeffding bound
3.17.12 **if** $\Delta(G) > \epsilon \;||\; \epsilon < \tau \&\& \Delta(G) \geq \tau/2$ **then**
3.17.13 Let l_{new} be an internal node that splits on X_n
3.17.14 $Alt(l) = Alt(l) + l_{new}$
3.17.15 **foreach** *branch of the split* **do**
3.17.16 Add a new leaf l_m to l_new
3.17.17 $X_m = X - X_n$
3.17.18 $Alt(l_m) =$
3.17.19 Compute $G_m(X_\emptyset)$ using most frequent class at l_m
3.17.20 **foreach** *node* $l \in P$ **do**
3.17.21 **foreach** $x_{ij} \in x : X_i \in X_l$ **do**
3.17.22 Decrement $n_{ijk}(l)$

2.2 Ensemble Learning

Ensemble learning, due to its intrinsic merits, focuses to get the best out of a collection of base learners. Ensemble learning can be thought of as a divide-and-conquer approach. However, it may not necessarily divide the tasks rather might do more tasks. The motivation behind was presented in the previous chapter. In short, ensemble methods are interesting because of following possibilities:

- Avoiding the worst classifiers by averaging several classifiers.

- Fusing multiple classifier to improve performance of the best classifier.
- Arranging the better classifiers (might include the best one) in a way to outperform the best one.
- Dividing the streams into chunks, learn separate models from each and then combine their results (divide and conquer).

A number of methods have been developed in the past couple of decades focusing on these motivations. They employ different approaches to improve the final classifier. Following is a summary of different approaches of generating ensembles of classifiers.

- Creating multiple training sets by re-sampling the original set and feeding those into different learners.
- Using different combination of features to learn multiple classifiers.
- Manipulating class labels. For example, transforming the classes into binary classification problem by partitioning the class labels into disjoint subsets.
- Manipulating the learning algorithms such that they result in different outcomes for the same training set, e.g., introducing certain randomness into a tree growing algorithm.

In this section, we present these basic concepts of building an ensemble of a collection of classifier. We start with primitive methods such as bagging and boosting. However, we are particularly interested in more sophisticated methods based on decision tree learning methods on stream data such as Adaptive Size Hoeffding Tree (ASHT) bagging and ADaptive WINdow (ADWIN) bagging.

2.2.1 Bagging

Bagging, also known as bootstrap aggregating, is a meta algorithm that improves accuracy and reduces variance and chance of over-fitting. Bagging is typically used with tree based learners. Given a training set D of size n, bagging generates m new training sets D_i of size n_{new} where $i = \{1, 2, \ldots, m\}$ by sampling with replacement. If $n = n_{new}$ approximately two-third of the instances of D_i is expected to be unique examples from D while the rest being duplicates. If K is the number of examples originating from the original training set then $P(K = k) = \binom{n}{k} \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k}$. These m sets are then used to learn m classifiers. The final decision is given by a majority voting scheme over the decisions of these m classifiers. Figure 2.5 shows an illustrative example of the bagging method.

For the online bagging approach $n \to \infty$, and the binomial distribution of $P(K = k)$ tends to a Poisson distribution with $P(K = k) = \exp(-1)/k!$. Using this justification, online bagging chooses a k from $Poisson(1)$ for each incoming training examples (x_i, y_i), and updates the base learning algorithms k times. The classification is done as the same way as the batched approach, by unweighted voting of m base classifiers. For a similar distribution of training examples, online bagging produces similar approximated base models. If (i) the

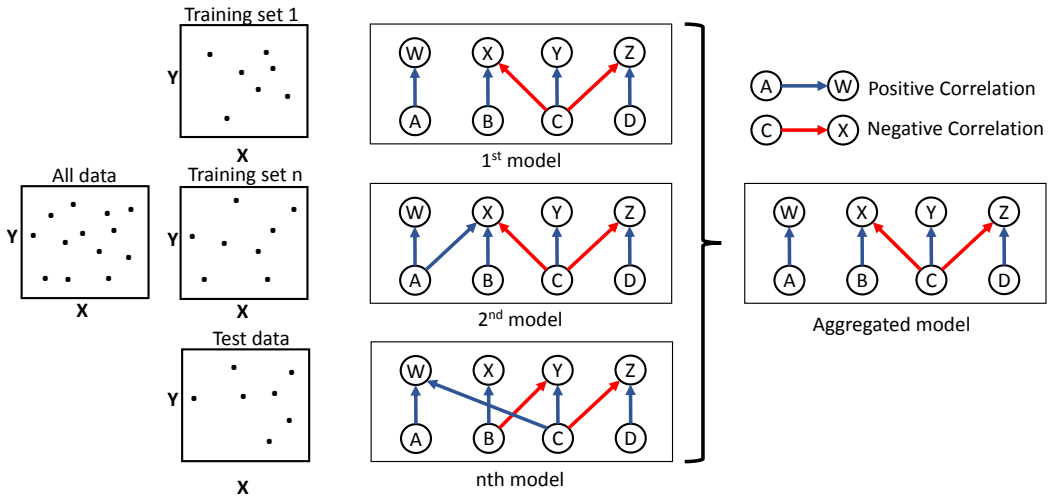


Figure 2.5: Bagging or bootstrap aggregation

used base learner converges to the same classifier with increasing training examples, and (ii) the base learner produces the same classifier given a fixed training set for both batched and online bagging methods, then online bagging will converge to the classifier obtained through batched learning methods.

2.2.2 Boosting

Boosting is another supervised learning algorithm to iteratively improve learning hypothesis. The motivation behind the boosting approach is that a set of weak classifiers could create a single strong learner. It forces a weak classifier to update or generate new rules that make less mistakes on previously misclassified records. Initially all records are assigned the same weights. At the end of a boosting round, weights might change due to the errors in classification. Weights of the instances are increased or decreased if they are classified wrongly or correctly, respectively. Thus successive classifiers depend upon their predecessors.

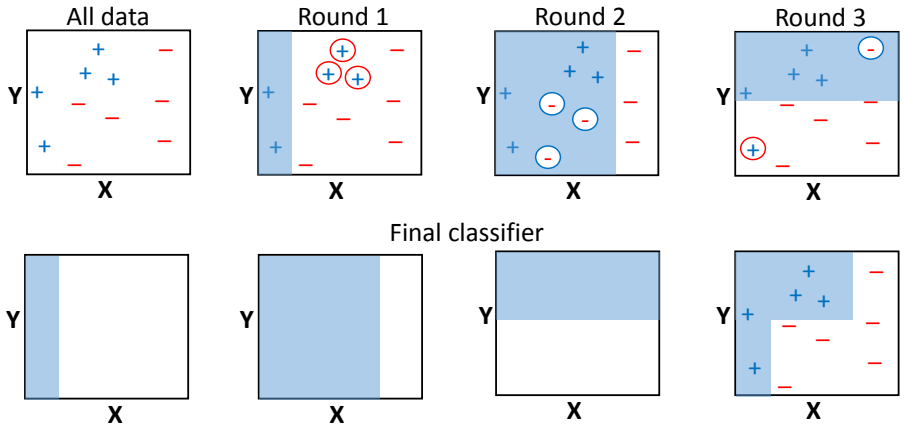


Figure 2.6: Boosting method example

Mathematically, let $H = \{h_1, h_2, \ldots, h_n\}$ be n weak hypotheses. The combined hypothesis $H(.)$ is a weighted majority vote of the n weak hypotheses where each hypothesis h_i has a weight of α_i for $i = \{1, 2, \ldots, n\}$:

$$H(.) = sign\left(\sum_{i=1}^{n} \alpha_i h_i(x)\right). \tag{2.5}$$

One of the earliest boosting approaches was proposed in [Schapire, 1990]. It calls weak learners three times on three modified distributions and achieves slight improvement in accuracy. Later, [Freund and Schapire, 1997] proposed AdaBoost, an adaptive boosting method, that works with a principle of minimizing upper bound of empirical error. Let, $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ are given examples where $x_i \in X, y_i \in Y = [-1, +1]$. Let, $D_t(i)$ be the weight of i-th example at t-th round. Then, AdaBoost initializes $D_1(i) = 1/n$. Iterative steps are as follows:

For each iteration $t = 1, 2, \ldots, T$:

- Train a weak learner using distribution D_t
- Get a weak hypothesis $h_t : X \to \{-1, +1\}$ with error

$$\epsilon_t = Pr_{i\ D_t}[h_t(x_i) \neq y_i]$$

- Choose $\alpha_t = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
- Update $D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha y_i h_t(x_i))}{Z_t}$ where Z_t is a normalization factor

Finally, the output is calculated with the Equation 2.5 given above.

An online variant of AdaBoost uses a similar approach as the online bagging method, simulating sampling with replacement using a Poisson distribution. However, here the Poisson parameter λ (Equation 2.1) associated with an example is incremented if that particular example is misclassified. In case of correct classification λ is decremented. Similar to the AdaBoost, the online boosting approach assigns total weights equally to the correctly and the misclassified instances, i.e. half of the total weights each. Unlike AdaBoost, in online boosting weights are updated based on only the examples seen thus far rather than the complete training set. This is intuitively problematic as initial hypotheses are built on too few examples. Even with this limitation, online boosting shows a good performance. Online boosting with naïve Bayes base learner converges to the model achieved with AdaBoost as the number of training instances tends to infinity.

2.2.3 Adaptive-Size Hoeffding Tree (ASHT) Bagging

In a previous section (Section 2.1.5), we have introduced the Very Fast Decision Tree (VFDT) or Hoeffding Tree (HT). The Hoeffding Tree is inspired by the fact that a small sample size could be sufficient to effectively choose an optimal splitting attribute. An upper bound of the error introduced because of such a generalization is given using Hoeffding bound.

The Adaptive-Size Hoeffding Tree (ASHT) is, as the name suggests, extended from a Hoeffding tree with deletion of nodes or resetting of the tree capabilities. Following are two significant differences of adaptive-size Hoeffding tree with a Hoeffding tree:

- The maximum number of split nodes or the size of the tree is bounded in ASHT. There is no such limit on HT. HT grows indefinitely as the data and new concepts are introduced.
- When the number of split nodes exceeds the maximum value (essentially after a new split), some nodes (or entire tree) are deleted to retain the tree property (max size).

There are two different choices for the deletion of nodes for the second point. The first option is to delete the oldest node. In a Hoeffding tree, the root is always the oldest node. All children of the root except for the nodes that were further splitted are also deleted in this case. The new root would be the node from the children of the root which is not deleted. Another and cruder approach would be to delete the complete tree altogether i.e. resetting the tree.

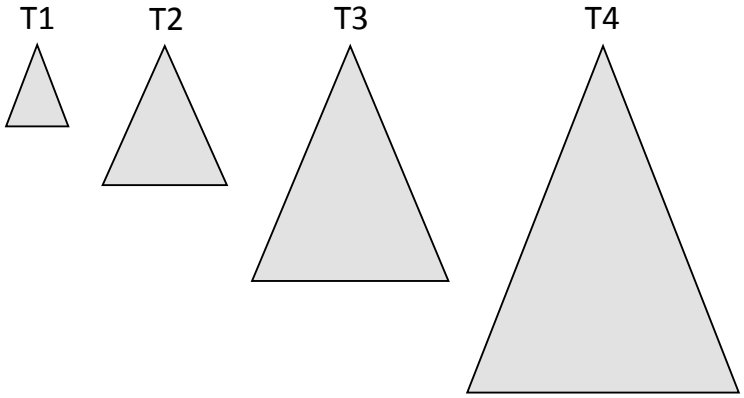


Figure 2.7: Adaptive Size Hoeffding Tree bagging concept

Based on this modified Hoeffding tree, a bagging method is developed in [Bifet et al., 2009]. The intuition of the method is as follows: smaller trees adapts to changes faster than the larger trees. However, larger trees perform better where data have only small changes or drifts because they are built with more data. A tree of size s would expect to be reset twice as often as a tree with size $2s$. Using an ensemble of different sizes would thus give a set of trees with different reset speeds. Smaller trees would be updated for small changes in the streams, while larger ones will maintain a history for a longer period. Polling over this set of classifiers would thus be expected to result in a classifier with finer granularity. It is to be noted that a reset will occur even for stationary data, however, this should not have any negative influence on the ensemble's predictive performance for stationary cases.

The proposed bagging method in [Bifet et al., 2009] uses n adaptive-size Hoeffding trees. The n-th ASHT has twice the maximum size of the $(n-1)$-th tree. Size of the smallest tree is 2. For the polling, each tree is associated with a weight, which is selected to be the inverse of the squared error.

2.2.4 ADWIN Bagging

ADWIN Bagging combines three different powerful concepts together: (i) bagging (Section 2.2.1), (ii) adaptive window change detection (Section 2.1.3), and (iii) Hoeffding tree (Section 2.1.5).

Bagging using ADWIN is implemented as ADWIN bagging where the bagging method is the online bagging method of Oza and Russell [Oza and Russell, 2001] with the addition of the ADWIN algorithm as a change detector. Hoeffding Trees are used as base classifier. When a change is detected, the worst classifier of the ensemble is removed and a new classifier is added to the ensemble.

Chapter 3

A New Hoeffding Tree Based Ensemble Approach

In the previous chapters, we have presented the motivations, and fundamental concepts for this thesis. Now, in this chapter, we present our algorithms, Size Restricted Hoeffding Tree (SRHT) and Carry-over bagging. The size restricted Hoeffding tree is extended from the original Hoeffding tree algorithm and incorporates the ADWIN method. Carry-over bagging is developed on top the concepts of the Oza online bagging approach. Before discussing the algorithms we define our problem mathematically.

3.1 Problem Statement

Various modern application areas produce huge amounts of data or so-called data streams. Most state-of-the-art stream mining methods generalize all application areas by similar streams. Assumptions are made that streams are generated from constant generators with concepts following some distribution. The concepts themselves may drift or evolve, but the overall nature of the generators remain the same.

A close examination of certain application domains such as social media, WWW, telecommunication, etc. suggests that streams are decomposable in smaller sub-streams, possibly in a mutually exclusive manner. These sub-streams can differ significantly in the way they generate or contribute to the overall stream. Some of these sub-streams are very short-lived but produce an extremely high number of instances, whereas some are always present and consistently produce data but with relatively very slow speed.

Generalizing such streams with a generator that does not produce these scenarios might not properly represent these applications. Performance measures such as overall accuracy, may fail to indicate limitations of learners in such environment.

In next chapter (Chapter 4), we demonstrate how to synthesize a data set with speed varying properties. With this type of data sets, current state-of-the-art methods face trouble in learning the slower streams. As it takes longer time to accumulate sufficient information for the concepts with slower speed, only those classifiers with larger capacity and longer lifetime would contain decision rules for slower concepts. Then again, we have the

usual challenges of concept drift, size of the classifier, over-fitting, etc.

With these motivations, we define our problem statement as follows:

Online classification of data streams, which are decomposable into speed-varied sub-streams, with Hoeffding tree based approaches.

Formally, given a stream S of very high volume, possibly infinite, having a set of m nominal or continuous attributes X, where each instance has a class label from a set $Y = \{y_1, y_2, \ldots, y_k\}$; we want to build a classifier that is able to predict the classes of incoming instances at any given time. Note that, this definition is very general, and can be used for any of the current learners. The uniqueness lies the in nature of stream S. We want to improve the performance when S is a composite stream of a set of sub-streams with huge contribution differences.

3.2 Solution Overview

Like all other stream learning algorithms, there are 3 (three) important aspects that need to be addressed in our potential learning algorithm. These are (i) the model has to be updated to be able to classify most recent data, (ii) the model has to detect concept drifts and has to be updated accordingly, and (iii) the model must not grow to be very complex to avoid over-fitting. Along with these requirements for our problem, additionally, (iv) it needs to perform equally good for different concepts within the classes. As the arguments were made before, fulfilling this new requirement is challenging because of the first two requirements. To keep the model up-to-date with the data and the concept, the effect of older concepts and data need to be removed from the model. In doing so, very easily we end up losing information of the slower sub-streams. As the faster sub-streams produce a high volume of data, the model is always dominated by those concepts. Slower sub-streams initially get classified wrongly, then over time when the model gathers enough information about these slower streams or concepts, new rules emerge for those concepts. However, these rules become 'old' soon with the incoming volume of faster concepts. As a result, in most of the current state-of-the-art stream mining methods, these rules will get pruned after a while to keep the model update to data and concepts.

To achieve the fourth goal, we modify a few of the existing methods and use them in a combination. From the methods discussed in Chapter 2, we learned approaches of learning a decision a tree and keep it updated to the newer concepts. Based on that, let us first analyze what are the possible approaches that can be taken to keep a model updated to the newer concepts and data, and how to get rid of the effect of older concepts.

3.2.1 Incrementally Growing Tree

Using methods such as a Hoeffding Tree (Section 2.1.5) would result in a growing tree with time. A Hoeffding tree keeps itself updated by spanning the entire feature space and updating the weights values within the leaves. Theoretically, with sufficiently large

learning time and diverse data instances, Hoeffding will span the entire feature space with all possible combinations.

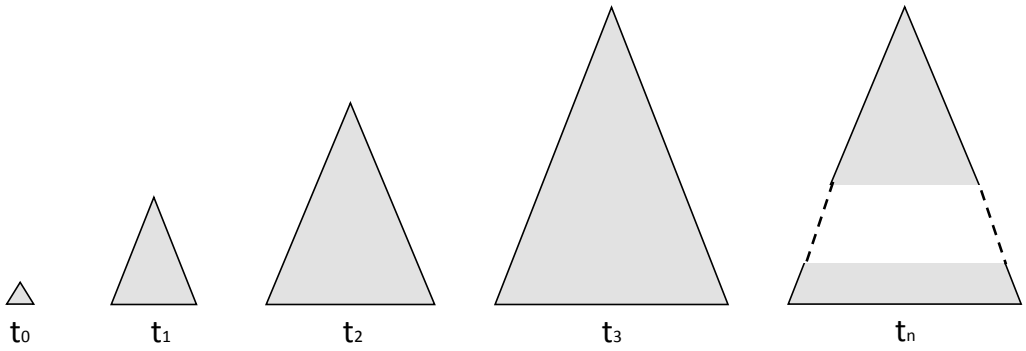


Figure 3.1: Incrementally expanding tree

Figure 3.1 shows how the size of the tree may change over time. At t_0, it starts with a single node, and keeps expanding as the time passes. Hypothetical corresponding decision boundaries in 2D space are shown in Figure 3.2. As the figure shows, the model become complex and prone to the slightest changes in data.

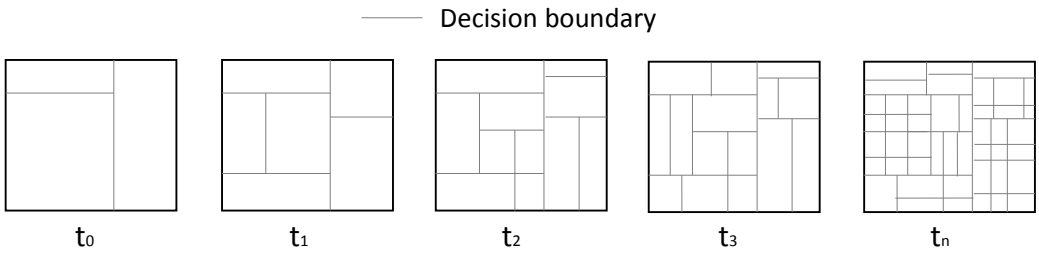


Figure 3.2: Decision boundaries for infinitely expanding tree

3.2.2 Resetting Tree

The scheme of an incrementally growing Hoeffding tree is simple and easy to implement. However, one of the major shortcomings of this approach is that, it leads to over-fitting. As there are abundance of training instances, after a period, the model starts following the instances rather than trying to realize the underlying relations. One way to stop that from happening is to reset the tree once in a while (Figure 3.3). This reset could be triggered by some time dependent event, or it could also be dependent on some properties of the tree. For example, depth of the tree, size or number of nodes in the tree, number of decision nodes in the tree, etc. can be used as a trigger with a maximum threshold bound.

Essentially, by resetting the entire tree, the model starts learning from scratch again. Every decision rule learned from observed data is lost (Figure 3.4). However, it does also get rid of all the biases for the incoming data that were present because of the observed data. In Adaptive Size Hoeffding Tree (ASHT) bagging, this reset idea has been used in association with an ensemble of different sized ASHTs (Section 2.2.3). As the size threshold

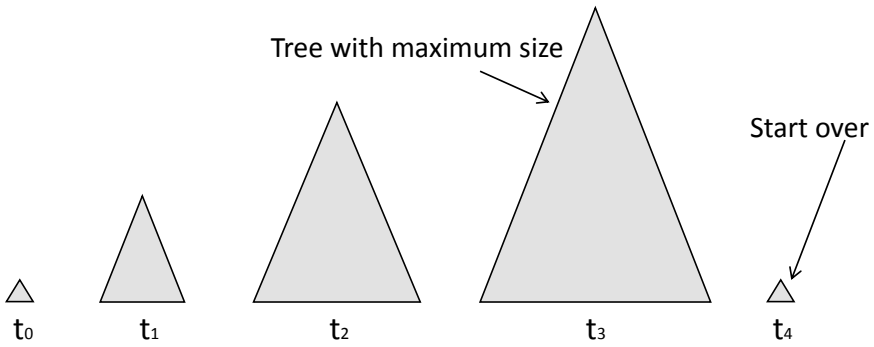


Figure 3.3: Resetting entire tree upon reaching threshold size

is different for the trees in the ensemble, the probability of all trees being reset at once is very small. Thus, by resetting a tree in an ensemble allows the model to retain important information upon a reset of a tree in the ensemble through other trees. However, it is important to understand that reset of smaller trees in ensembles have lesser effect than the reset of larger trees. Smaller trees in the ensemble target recent data, and there is abundance of that. On the other hand, larger trees contain information learned over a longer period, thus also containing small but important concepts. This is exactly what we want to retain.

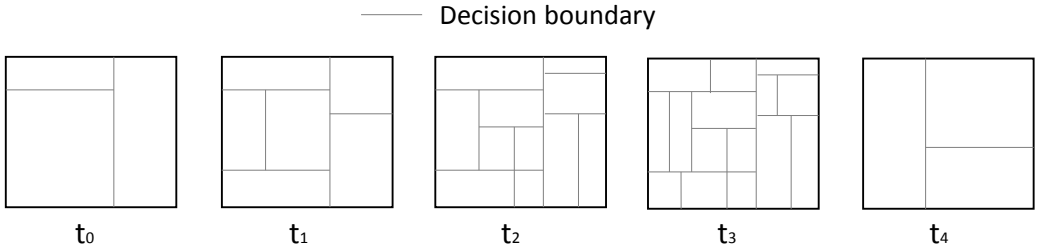


Figure 3.4: Decision boundaries after tree reset

3.2.3 Pruning by Deleting Older Nodes

An alternate choice to reset the entire tree is to delete the older decision nodes, e.g. root or top levels. As in the Hoeffding tree, the split attribute choice for each level depends on newer data from the previous level, the top level decision nodes are chosen based on oldest data segments (Figure 3.5). Thus, removing the root and all its children except for the one chosen as the next root would remove the effect of the oldest data. It also allows the tree to grow again, if there is any size restriction, as the threshold (number of decision nodes) is effectively decreased by at least 1.

Figure 3.6 shows the changes in the decision boundary upon deleting the root. As it shows, only the earliest boundaries are deleted in this process. If the model has already become too complex, then this does not improve the situation much. But for less complex models, this should create a chance for the tree to learn information within the newly

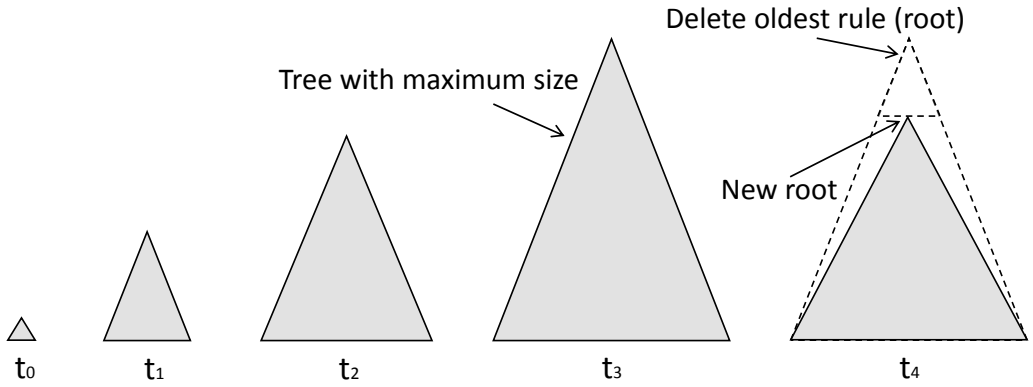


Figure 3.5: Deleting the oldest branch (root) and selecting new root from its children

merged regions. Moreover, because the order in which the checks are performed changes, the existing decision rules might change (decision class).

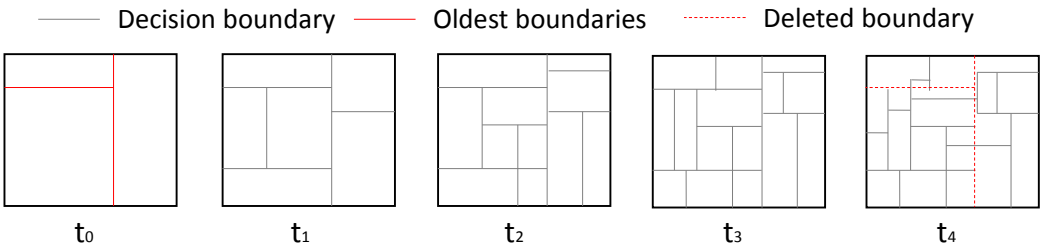


Figure 3.6: Decision boundaries after deleting root

3.2.4 Pruning by Maintaining Alternate Trees

In a different approach than deleting root or the entire tree, maintaining an alternate sub-tree wherever necessary can effectively maintain the model's relevance to the recent data. Adaptive Hoeffding Tree uses such an approach by maintaining ADWIN (Section 2.1.3) to check the error margin at each decision node. In case of an increase in error margin at any node, an alternate branch is initiated and learned. In the future, if there is enough evidence that an alternate branch performs better than the current one, the current sub-tree is replaced with the alternate sub-tree. Otherwise, the alternate sub-tree is deleted. Every time error margin exceeds the main tree and the alternate tree, a new alternate branch is created. Thus, one decision node might have multiple alternate sub-trees. Note that, an alternate branch might get started at the root itself, and if it performs better, would change the entire tree with something new.

Pruning using alternate trees can effectively detect changes in the concepts. As it employs ADWIN, the nodes are not immediately replaced if they start performing worse than before. Rather, it waits to confirm that an actual change in the data concepts has occurred. Otherwise, it deletes the alternate tree. This way, the model is saved from being prone to outliers and very small blocks of data from other concepts. In Figure 3.8,

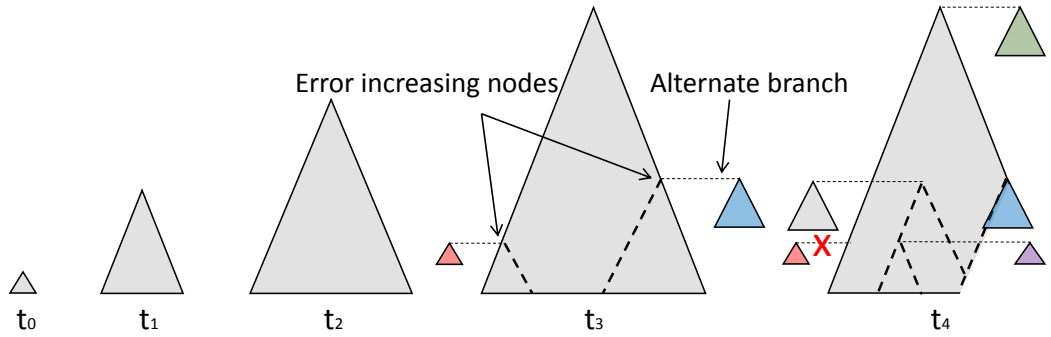


Figure 3.7: Maintaining alternate branches for pruning

we illustrated how the decision could change in terms of the trees shown in Figure 3.7. Generally, it is expected that a block of decision boundaries will be replaced by another block. It does not necessarily need be a simpler one. Alternate trees, theoretically, can be more complex than the block they would replace in the original tree.

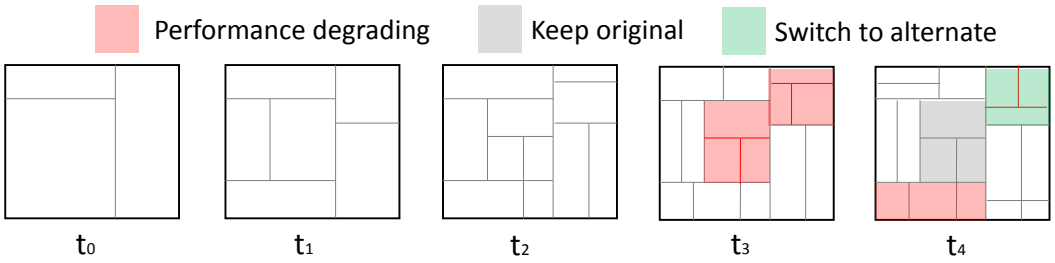


Figure 3.8: Decision boundaries after pruning using alternate branching

3.2.5 Combining the Ideas

To approach our problem, we combine these ideas effectively. Our solution is entirely motivated by assumptions made in Hoeffding tree approaches. To effectively decide on a split attribute in data streams with a certain degree of accuracy, a small portion of the data is sufficient. The bound of accuracy can be estimated using Hoeffding bound. Based on this fundamental assumption, we combined the ideas of restricting and resetting Hoeffding tree based on a threshold size in terms of number of decision nodes, and the concept of maintaining alternate branches for nodes where performance are degrading using ADWIN method. Restricting the size of the tree can be used to control the amount of data the tree would be built upon. Applying ADWIN within the size restricted tree would ensure that the tree model is updated to the drifted concepts.

As discussed earlier, immediate reset of trees is not desirable where retaining slower concepts is a concern. Thus, we developed a bagging scheme based on ASHT bagging to delay the reset of the trees upon reaching the limit. With this scheme we stop the tree growth immediately as the size limitation is reached. We also start building a new tree with the same size restriction. For the successive learning, both trees are learned with the

exception that the old tree cannot grow. It may, however, switch to alternate branches. In cases, this might reduce the tree size and allow it to grow again. Switching to an alternate tree that would increase the tree size is not allowed. For the bagging, we thus maintain two sets of learners. One is same as the one from ASHT bagging. A fixed number of trees each with double the size of the previous one, first one being of size 2, typically. The other is initially empty with a fixed capacity. Once a tree in the first set reaches its limit it is moved to the second set. The second set works as a fixed sized queue, as the newer trees are moved from the first set, older trees gets deleted. As the smaller trees reaches the size limit very fast, a threshold is set on the size of the trees that can be moved to the second set. The trees reaching their limit below this threshold are being reset immediately. In other words, we "carry-over" the larger the trees for a certain period of time even after reaching the max size limit. For the voting for a new instance, all classifier in both the sets contributes.

With this carrying-over concept, we essentially increase the weights of the decision rules in the larger trees, which directly influences the problem we are addressing. The larger trees contain decision rules for slower streams. By delaying the deletion of such trees, we create a buffer time for the new classifier to learn something from the newer data, while the older ones keep voting. The voting process remain more balanced between newer and older concepts during the transition period. Because in the transition time the new classifier with larger size limit acts as a smaller classifier just because it has not seen enough data yet. This basically increases the voting bias towards newer concepts. With our carry-over bagging, this becomes more balanced, as the larger trees would still keep voting for concepts present in older data. Algorithms devised based on the discussions above are explained in the following sections.

3.3 Size Restricted Hoeffding Tree (SRHT)

A Size Restricted Hoeffding Tree is an ADWIN variant of Adaptive Size Hoeffding Tree. It starts with an empty leaf or root. Every incoming instances is traversed to a leaf. The class label at the leaves are computed based on some base learners such as k-NN, naïve Bayes, majority voting, etc. A grace period is imposed to reduce the number of unnecessary computations. Once a leaf gets a predefined number of instances N_{min}, the checking for branching is started. If a leaf is not pure, its possibility for branching is checked. The Hoeffding bound is applied here. The best two attributes are chosen based on an information gain function. If the difference of gain between attributes is greater than the computed Hoeffding bound ϵ and the total number of decision nodes is less a given threshold n_d, the leaf node is replaced with a decision node, and the instances weights are distributed among the new node's children. Additionally, the ADWIN scheme is maintained at each decision node to check the error status. Whenever a decision node's performance starts falling, i.e. error increases, an alternate sub-tree for that node is created. Every instance reaching that node would be used to learn both the original and the alternate sub-trees. If the performance of any alternate subtree becomes better than the

Algorithm 4: SRHT: Size Restricted Hoeffding Tree

Input : S: Stream of examples
X: Set of nominal attributes
Y: Set of class labels $Y = \{y_1, y_2, \ldots, y_k\}$
$G(.)$: Split evaluation function
N_{min}: Minimum number of examples
δ: is one minus the desired probability
d: Alternate tree switching bound
n_d: Number of maximum decision nodes
τ: Constant to resolve ties
Output: HT: is a decision tree

```
4.1  begin
4.2      Let HT ← Empty Leaf (Root)
4.3      foreach example(x, y_k) ∈ S do
4.4          Traverse the tree HT from root till a leaf l
4.5          if y_k ==? then                                   // Missing class label
4.6              Classify with majority class in the leaf l
4.7          else
4.8              Update sufficient statistics
4.9              if Number of examples in l > N_min then
4.10                 Compute G_l(X_i) for all attributes
4.11                 Let X_a be the attribute with highest G_l
4.12                 Let X_b be the attribute with second highest G_l
4.13                 Start maintaining ADWIN
4.14                 if error increased from previous step then
4.15                     Start maintaining alternate tree
4.16                 else if oldError - alternateError > d then
4.17                     if ADWIN window sufficiently big then
4.18                         Switch to alternate tree
4.19                     else if alternateError - oldError > d then
4.20                         Delete alternate tree
4.21                 Compute ε = sqrt(R² ln(2/δ) / 2n)          // Hoeffding bound
4.22                 if G(X_a) − G(X_b) > ε || ε < τ then
4.23                     if decision node count < n_d then
4.24                         Replace l with a splitting test based on attribute X_a
4.25                         Add a new empty leaf for each branch of the split
4.26     Return HT
```

original sub-tree by a margin of d, the original sub-tree is replaced with the alternate one. Similarly, if the original sub-tree's performance improves again by a margin of d the alternate trees are deleted. The use of margin d ensures that, the classifier will not swing back and forth in case of similar and alternating concepts.

3.4 Carry-over Bagging with SRHT

SRHT is used in association with the Carry-over Bagging (CoBag) ensemble approach. As SRHT is limited in terms of size, it will get outdated time to time and would require to be replaced. This is done using the carry-over bagging method.

Algorithm 5: CoBag: Carry-Over Bagging with SRHT

Input : H: Set of base learners
n: Ensemble size
m: Additional ensemble size
b: Tree size limit to use additional ensembles
Output: A SRHT ensemble

```
5.1  begin
5.2      Create n SRHT H_i
5.3      Creating an empty ensemble with capacity m
5.4      foreach h ∈ H do
5.5          Set k according to Poisson(1)
5.6          for i : 1 − k do
5.7              Learn h with SRHT
5.8              if h.size > h.maxSize and h.size < b then
5.9                  Reset h
5.10             else if h.size > h.maxSize then
5.11                 Move h to Alternate ensemble
5.12                 Create new SRHT with size limit of h.maxSize
5.13                 if Size of additional ensemble > m then
5.14                     Delete the oldest h from additional ensemble
```

The carry-over bagging method is similar to the one used in ASHT bagging, which implements Oza online bagging approach. We use a similar approach with notable difference of using two different sets of classifiers for the ensemble. For the first set, a fixed number (n) of classifiers is initialized using different size limits. The size of the smallest classifier is a user given value. Successive classifiers have a limit of twice the size of its immediate predecessor. As explained previously, this ensemble performs similarly to the one from the ASHT bagging method. We introduced the second set with a capacity of m classifiers to store the classifiers from the first set when they reach their limit. Which classifiers are allowed to be stored in the second set is guarded by a threshold size, b. Hence, smaller classifiers being reset often do not affect the performance of the overall process. When the second set becomes full, and there is a new classifier to be added (that reached its limit), we deleted the oldest one from the set. For our experimentations, the size of the second set is kept equal to the size of first set, and the threshold is set to size of the median classifier.

3.5 Summary

Before we start presenting the data set preparation and the experimental evaluation results, here is a brief summary of the investigated methods:

Hoeffding Tree (HT): With enough instances on an impure node it checks for the best two attributes. If their information gain difference satisfies the Hoeffding bound, it splits the node with best attribute.

Adaptive Hoeffding Tree (AdaHT): It uses ADWIN monitor to replace sub-trees if the performance starts degrading.

Adaptive Size Hoeffding Tree (ASHT): Reimplementation of Hoeffding tree with complete resetting or just root deleting capability.

Bagging with Hoeffding Tree (BagHT): Oza online bagging with Hoeffding tree. All the trees grows with time.

Bagging with Adaptive Hoeffding Tree (BagHT): Oza online bagging with an adaptive Hoeffding tree. The trees grow indefinitely unless they are replaced by the ADWIN monitoring scheme.

Bagging with Adaptive Size Hoeffding Tree (BagASHT): Oza online bagging with ASHTs with different size limits.

Bagging with Size Restricted Hoeffding Tree (BagSRHT/CoBagSRHT): Carry-over bagging with SRHT with different size limits.

Boosting with Hoeffding Tree (BoostHT): Oza online boosting with Hoeffding Tree.

Boosting with Adaptive Hoeffding Tree (BoostAdaHT): Oza online boosting with adaptive Hoeffding Tree.

Boosting with ADWIN with Hoeffding Tree (BoostAdwin): Oza online boosting where the ensemble maintains ADWIN for classifiers (Hoeffding tree). Similar to BoostAdaHT.

Table 3.1: Comparison among various learners

	Size	**Reset**	**Pruning**	**Alternate trees**	**Stats stored**	**Reset by**
HT	increasing	no	no	no	leaf only	no
AdaHT	increasing	possible	yes	yes	all nodes	itself
ASHT	bounded	yes	no	no	leaf only	itself
BagHT	increasing	yes	no	no	leaf only	ensemble
BagAdaHT	increasing	yes	yes	yes	all nodes	ensemble
BagASHT	bounded	yes	no	no	leaf only	ensemble
BoostHT	increasing	no	no	no	leaf only	ensemble
BoostAdaHT	increasing	possible	yes	yes	all nodes	ensemble
BoostAdwin	bounded	yes	no	no	all nodes	ensemble
BagSRHT	bounded	yes	yes	yes	all nodes	ensemble

Chapter 4

Data Set Generation

Due to the lack of labeled real-world data streams, stream mining algorithms are most of the time tested with synthesized data. Synthesized data has several advantages over poorly labeled real-world data streams. First of all, it is rather easy to produce. It has significantly less storage overhead. It is easier to control synthesizing parameters to generate data sets with the desired concept drift, recurrence, etc. scenarios.

In this chapter, we briefly discuss some of such generators. For our experimentations we used random radial basis function (RBF) generator. Thus, we discuss the generation process and the generated stream's properties from a random RBF generator in greater details. We then describe the modifications in the generation process to introduce variable speed RBF streams.

4.1 Existing Stream Generators

A number of generators have been introduced in literature in the past decades: SEA concept generator, STAGGER concepts generator, rotating hyperplane generator, random RBF generator, waveform generator are some of the commonly used ones.

SEA Concepts Generator

The SEA concept generator is introduced in [Street and Kim, 2001] to generate a synthetic data set with abrupt concept drift. It uses three parameters valued between 0 and 10 inclusive. The points of the data set are divided into blocks of different concepts. Classification within the each block is controlled using the input parameters along with a threshold value.

STAGGER Concepts Generator

STAGGER is one of the early methods developed for stream generation by Schlimmer et al. [Schlimmer and Granger, 1986]. Binary concepts are generated from three attributes. Attributes are size, shape, and color. Each of the attributes has three possible values. Small, medium, and large are the sizes; circle, triangle, and rectangle are the shapes; and

red, blue, and green are the colors. 27 combinations are being mapped into a binary class by this generator.

Rotating Hyperplane Generator

A rotating hyperplane generator is introduced in the evaluation of CVFDT [Hulten et al., 2001]. Hyperplanes can effectively be used to simulate concept drifting environments. A hyperplane in a d-dimensional space is a set of points x that satisfies following equation:

$$\sum_{i=1}^{d} w_i x_i = w_0 = \sum_{i=1}^{d} w_i$$

where x_i is the ith coordinate of x. The hyperplane works as the boundary between two of the binary classes. By controlling the orientation and position of the hyperplane concept drift can be controlled.

Wavefront Generator

The wavefront generator is a data set available at the UCI machine learning repository [Asuncion and Newman, 2007]. Two or three base waves are used to generate a data set of three different classes of waveforms. The prediction task is to predict the classes of waveforms. The optimal Bayes classification rate is known to be 86%. There are two variations of this generator. Wave21 uses 21 numeric noisy attributes, and wave40 uses 19 more irrelevant attributes along with original 21.

LED Generator

LED generator is also available at the UCI machine learning repository [Asuncion and Newman, 2007]. The prediction task for this data set is to predict the digit displayed on a seven segment LED display. Each segment has 10% chance of being inverted. The optimal classification using naïve Bayes on this data set is 74%.

Random Radial Basis Function Generator

All the generators mentioned above are relatively simple and easy to use. However, it is harder to model a complex scenario with those generators. Hypothesis spaces for these generators are not very large, except for the rotating hyperplane generator. But with the rotating hyperplane generator it is harder to produce overlapping space and outliers. A random radial basis function (RBF) generator is therefore introduced to generate a complex concept type that is not straight forward to approximate, especially with a decision tree model.

RBF generator starts by selecting a fixed number of random centroids, each at a random position in the hyperspace with a standard deviation, class label and weight. Instances are generated by selecting a centroid at random and generating a random point

near that centroid by maintaining a Gaussian distribution with the previously selected standard deviation and the location of the centroid as mean. Direction of the deviation is selected at random. The class label of the instance is assigned to be the one of the selected centroid. The selection of centroid depends on the weights of the centroids, so that centroids with higher weights get higher chances to get selected.

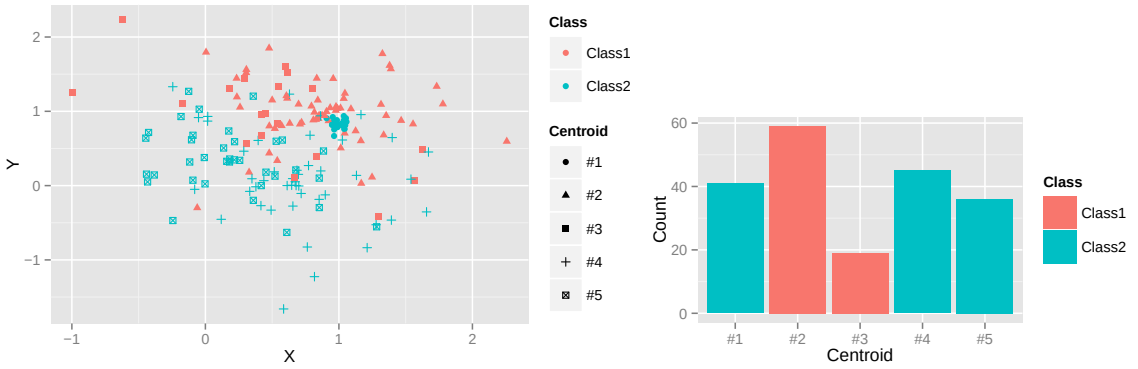


Figure 4.1: RBF data generation in 2D space (a) distribution (b) histogram

Figure 4.1 illustrates a generated data set on two dimensional space with no concept drift. Figure 4.1a shows the distribution of the instances, and Figure 4.1b presents the histogram of 5 centroids from which data are generated. The differences among the centroids' weights are clearly visible here. However, it is to notice here is that the overall data set is somewhat balanced in terms of class distribution.

Next, in Figure 4.2 we present 4 generated data sets with only one centroid, hence all instance belongs to one class. Figure 4.2a, 4.2b, 4.2c, and 4.2d have a drift coefficient of 0, 0.01, 0.1, and 1 respectively. As it can be seen, the instances are more compact when there is no concept drift. The introduction of a small concept drift gradually moves the concepts away. With a higher drift coefficient, instances become sparser.

We retain most of these properties of a RBF generator in creating our Variable Speed RBF Stream. We manipulate the generation process to achieve our desired properties in the data set. We want the data set to be balanced in terms of the binary class distribution. We also want some centroids to produce more data, possibly with higher drift while some centroids with very little drift produce fewer data in a certain time frame. With these goals, we present the data generation process for variable speed RBF Stream in the next chapter.

4.2 A New Approach to Generate Variable Speed RBF Streams

To generate a variable speed RBF stream, we replace the concept of centroids with pools of centroids. At the start of the generation process a sufficient number of centroids is

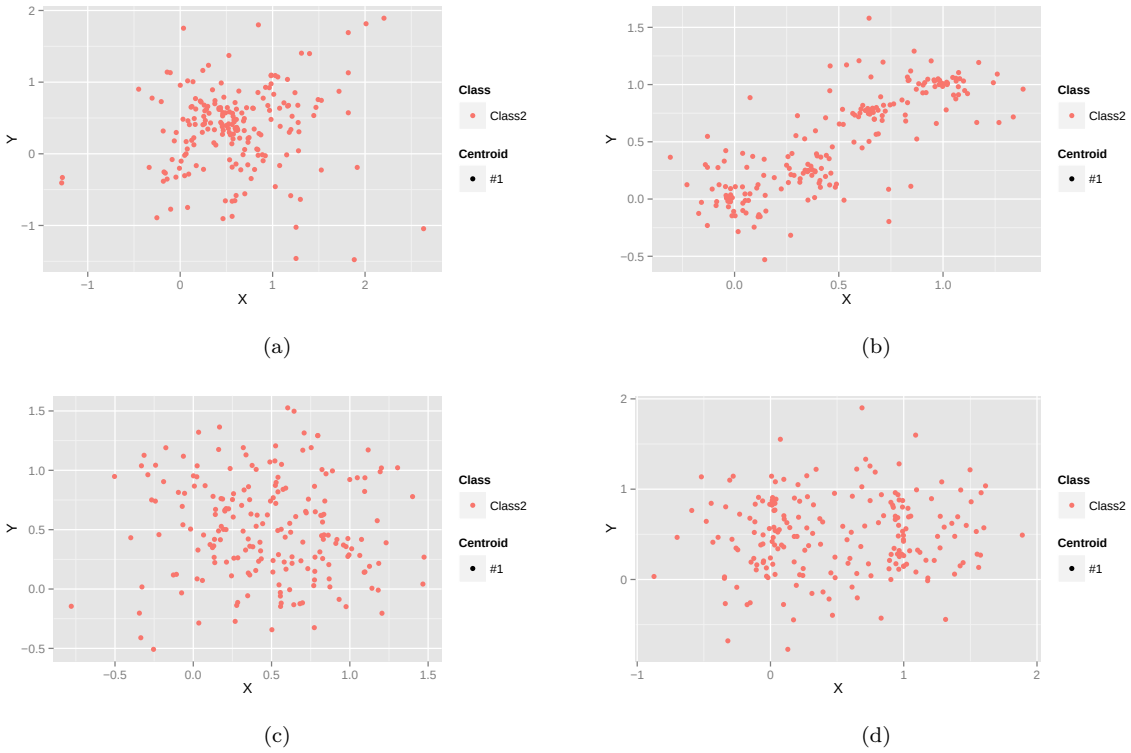


Figure 4.2: RBF data generation with concept drift. Drift coefficient: (a) 0, (b) 0.01, (c) 0.1, and (d) 1

generated and assigned randomly to a user specified number of pools. The generation process of centroids remain exactly the same as the RBF generator: a randomly chosen point in the hyper-space with a randomly assigned class label. Instances generating from each centroid have this point as the mean of the normal distribution with a randomly selected standard deviation. After assigning these centroids into pools, the pools are assigned relative weights. We have used linearly increasing and quadratically increasing weights for experiments. Drift coefficients associated with all the centroids within a pool are the same and also follow a quadratic rate among the pools. As we mentioned before, slower pools i.e. pools with lower weights have lower drift coefficients than the pools with higher weights.

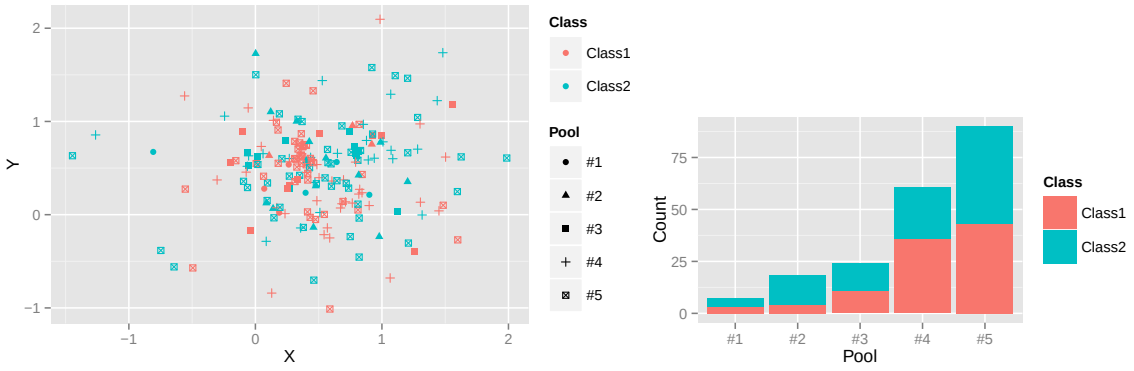
To mimic concept evolution and recurrence, we associate each pool with an activation percentage. The activation percentage determines the percentage of centroids within the pool that are activated. If a centroid is not activated, it will not produce any instance until it gets activated again. Slower pools has higher activation percentages. With these settings, the pool with the highest weight will produce a burst of instances belonging to only a few centroids while the pool with the lowest weight will produce small number of instances from a higher number of centroids. As mentioned before, these centroids are also associated with less drift. Thus, essentially we would get slow but consistent concepts.

For the ease of implementation, we achieved this weight variation and activation among the pools by generating instances in batches. Each batch starts by configuring the pools. The activation of centroids is changed randomly based on the activation percentage at

Algorithm 6: Variable Speed RBF Generator

Input : c: Number of classes
d: Number of attributes
p: Number of pools
b: Number of batch size
n: Size of the stream
Output: D: A data stream

begin
  Create P pools
  Generate m centroids // Sufficiently high number, $n >> p$
  Distribute m centroids to p pools randomly
  Let w_i is the weight of ith pool
  $w_i = a * i^2$ // Associate each pool with a quadratic weight
  Let a_i is the activation percentage of ith pool
  while $n > 0$ **do**
    foreach $P_i \in pools$ **do**
      Randomly select a_i fraction of centroids in P_i
    foreach $P_i \in pools$ **do**
      $n_i = w_i * b/w$ where $w = \sum w_i$
      Generate n_i instances
    Randomly shuffle all generated instances
    Add instances to D
    $n = n - b$
  Return D



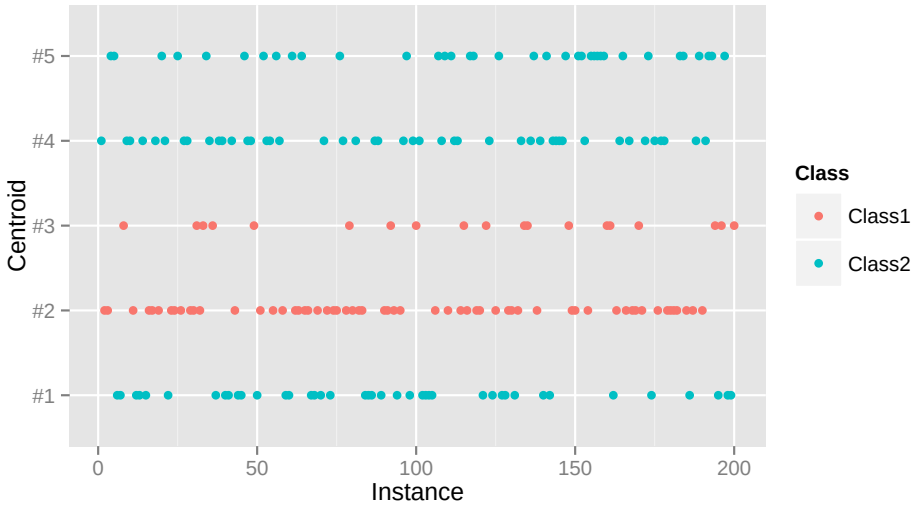
(a) (b)

Figure 4.3: Variable speed data generation in 2D space (a) distribution (b) histogram

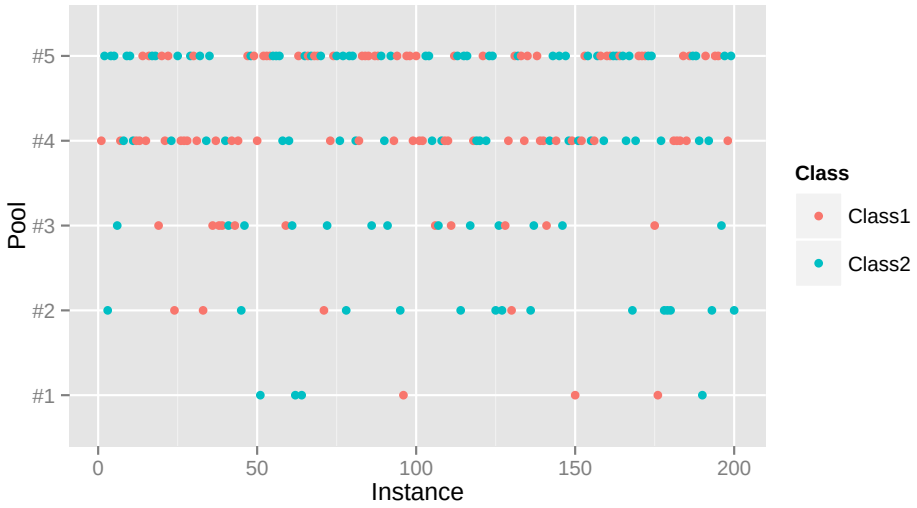
each configuration. It randomly activates the required number of centroids based on the associated activation percentage of the pools. For each batch, each pool generates a certain number of instances depending on its weight. Contributions from all the pools are then shuffled randomly. The batches serve as reservoirs. Once a reservoir gets depleted, the next batch fills the reservoir. Algorithm 6 presents pseudo-code of the generation process.

Figure 4.3 presents a generated data set with 5 pools. As the histogram indicates,

the 4^{th}, and 5^{th} pools produce significantly higher number of instances compared to the 1^{st}, 2^{nd}, or 3^{rd} pool. However, close inspection at Figure 4.3a would reveal that instances from these pools are more compact than the other two pools with a higher instance count. Another important property to notice here is that even though there is significant difference in the number of instances produced from each pool, the overall data set is somewhat balanced.



(a)



(b)

Figure 4.4: Timeline of instances (a) random RBF generator (b) variable-speed RBF generator

Lastly, it is to be noticed here that even though slower pools produce less instances than the faster ones, they are expected to be distributed in the entire time frame. Figure 4.4 confirms that expectation. In Figure 4.4a, it shows a timeline for a generated data set with the RBF generator. It shows that each centroid generates throughout the generation process. Similarly, in the data set generated by the variable speed RBF generator data

are generated by every pool, throughout the generation process (Figure 4.4b). The slowest pool produces less data, but it does not do so in a fraction of the time frame. Moreover, in Figure 4.5, it shows a timeline for the contributing centroids. Values are set such that at each timestamp there are about 75 instances, thus there are 3 different timestamps in the plot. Looking into the figure, it can be easily seen that for the 5th pool, at the beginning only Centroid10 is producing data. For the rest of the two timestamps Centroid30 is producing data. Similarly, for the 4th pool, at the beginning two centroids produces data: Centroid01 produces Class2 data and Centroid24 produces data from Class1. They also stop producing for the next two timestamps. For Pool3, Centroid29 is active all the time, while other change their activation. For another two pools most of the centroids are mostly active for all three timestamps.

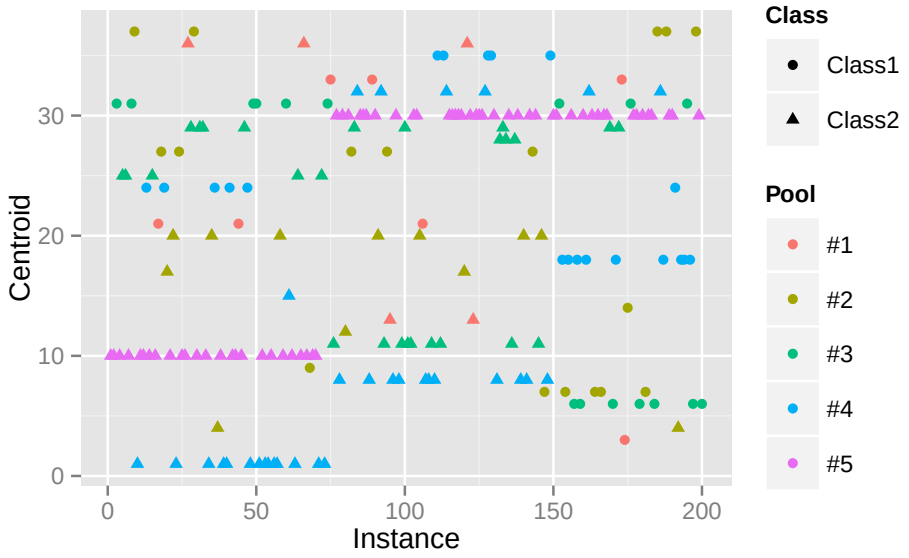


Figure 4.5: Timeline for contributing centroid in variable-speed RBF generator

The discussions so far in this chapter, demonstrate that the variable speed RBF generator is able to produce a stream that mimics the behaviors or properties of the streams introduced in Section 1.1, which is the basic motivation for our algorithm. In the next chapter, we evaluate the algorithms with these data generation schemes.

Chapter 5

Experimental Evaluation

In this chapter we present the empirical analysis of the discussed algorithms. We analyze the performances and properties of these algorithms varying a wide range of parameters. We compare our approach described in Chapter 3 with the current state-of-the-art methods presented in Chapter 2. Each of the analysis is performed simultaneously– for both data streams generated by random RBF generator and our newer approach that generates speed varied RBF streams (Chapter 4). A 64-bit Windows 7 machine with Intel Core i7 dual-core processor clocking up to 2.4 GHz and 8 GB of RAM is used to run the experiments. The implementation is derived from Java based MOA framework [Bifet et al., 2010b].

5.1 Evaluation Process

There are two different approaches in evaluating algorithms in a streaming environment: (i) holdout evaluation, and (ii) prequential evaluation. In holdout evaluation, two separate train and test sets are used. Once the training is done, typically using one single-pass, on the training set, the test set is used to evaluate the performance. This is ideal where there are large amounts of data, where it is expected that with splitted test set, chances of over-fitting would be very low. In prequential evaluation, each example is first used to test the model in hand, and then the instance is used to train (update) the model. This method, thus, continuously changes the observed performance metrics and the model. A prequential model is, particularly, a necessity when the classifier has to ensure that the model is updated for the newest instances. The well-known cross validation approach is trivially not applicable in the streaming environment. In our settings, prequential evaluation is most appropriate. Binary class data sets of 1,000,000 instances in a 10 dimensional hyperspace are generated using the random RBF generator and the variable speed RBF generator for the evaluation.

5.1.1 Performance Metrics

A number of metrics has been used for the evaluation criteria. Accuracy, processing time, memory usage, and kappa statistics are primarily used for the evaluation along with various tree structure related parameters such as tree depth, tree size, number of decision nodes,

etc. Moreover, we observed algorithm specific properties e.g. number of tree reset, number of pruned trees, number of maintained alternate trees, number of times of partial tree replacement, etc. These algorithm specific parameters are not always directly comparable. Parameters are comparable among HT, adaptive HT, and ASHT. Bagging and boosting approaches have a different setting. Thus, we used weighted averages for these methods when calculating the number of resets or pruning type of statistics. For example, in bagging methods, smaller trees reset more often, thus while calculating the number of resets, a lower relative weight is given to the smaller trees. In our experiments, weights are proportional to the maximum size limits, or equal where there is no size restriction.

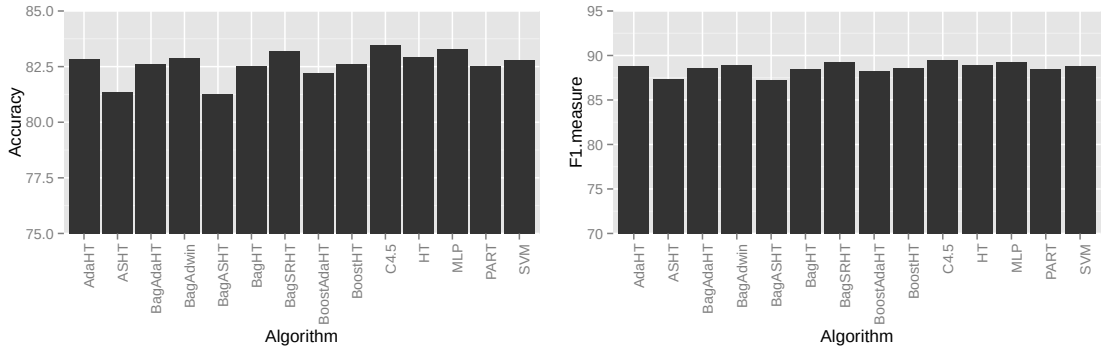
5.1.2 Generalization Error Bound

In batched leaning methods, the generalization error is a measurement that indicates how well a learning method generalizes to unseen data. It is the distance between the error on the training set and the test set, and is averaged over the entire set of possible training data for all iterations. In our stream mining environment, as we will be using prequential evaluation method, all samples would be first used as a test sample and then be used as a training sample. Moreover, as this will be performed only once, thus there would be no iteration over the training set, and the evaluation performance would be strongly tied with the sequence of the incoming data. Thus, the notion of generalization error does not apply here. Rather, we focus on the confidence of the model being build. For Hoeffding tree based approaches, δ (Algorithm 2, Algorithm 4) is used to indicate an allowable error in each split decision. For our experiments, we kept this value to 0.0000001. This is the maximum error to be accepted in a split decision. This is used to compute the Hoeffding bound, depending on which split decisions are taken.

5.2 Study case: Census Income Dataset

In the beginning, we check the performance of the stream algorithms in comparison to the classical batched data mining algorithms. For this evaluation, the census income data set [Kohavi, 1996] is used. This data set contains 48,842 instances of 14 attributes (6 continuous, 8 categorical) mapping to a binary "income" class (details in Appendix D). With experimental evaluation, it is found that most batched learning methods such as C4.5, SVM, MLP achieve an accuracy around 82-83% with 10 fold cross validation for the income class prediction task. As it can be seen from Figure 5.1, stream learning methods also achieve an accuracy within 1-2% percent difference than the batched learners. Measured F1 values also presents similar scores. Though differences are insignificant, however, ASHT and bagging with ASHT seem to perform relatively worse than other algorithms. It is because of the resetting of the trees. After each reset, a few instances will suffer because the new trees are still very small. On the other hand, with our proposed modifications, bagging with SRHT overcomes this problem, and performs nearly the same as C4.5.

In terms of the tree structure, the final classifiers from a stream learner are simpler than the C4.5 tree. While C4.5 has many decision rules with over 10 decision nodes, Hoeffding



(a) (b)

Figure 5.1: Comparison of classical and stream learning algorithms on census income data set with (a) accuracy and (b) F1-measure

tree based approaches are found to be performing similar with only 4 decision nodes. It is to note here that with a different ordering of the input data, a higher number of decision nodes is a possibility.

Lastly, we simulated a concept evolving scenario by controlled sampling of data from different clusters within the census income data. Five clusters were found using batched learning methods. While it had no significant effect on the performance of HT, however, methods that uses ADWIN are found to get reset often depending on the sampling sequence. When each clusters is used sequentially, they reset all 4 times when concepts were changed.

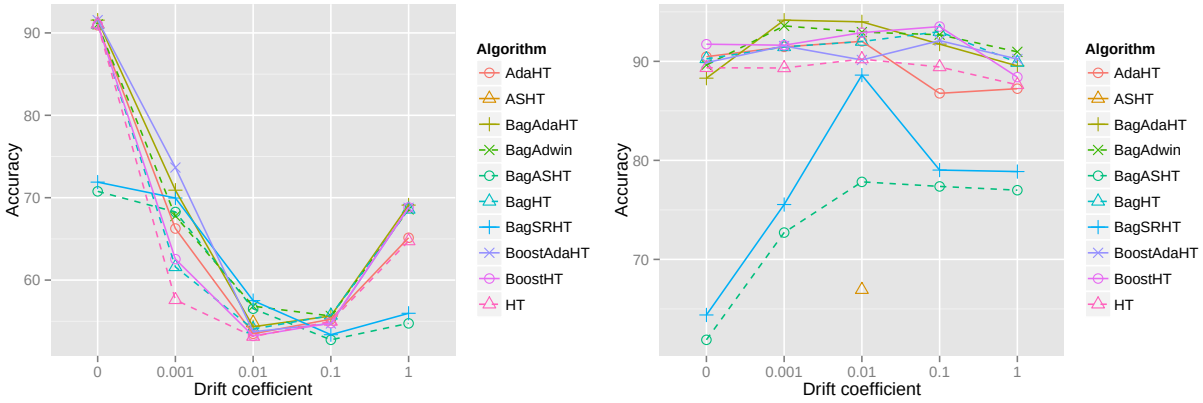
5.3 Impact of Parameters

The evaluated algorithms depends on a number of parameters. Grace period, tie threshold, and tree type (binary or non-binary) are the most important parameters for HT, adaptive HT, and ASHT. For bagging and boosting methods the ensemble size is important. Additionally, for ASHT and SRHT, we have first classifier size, choice between reset and pruning are also important. Similarly, there are parameters on which the data generation processes (Chapter 4) rely on. In this section, we analyze the effect of each of these parameters starting with the parameters for data generation. For the discussion here, we used the most relevant metrics in respect to parameter. A comprehensive list of plots is given in the Appendix A detailing all related metrics.

5.3.1 Effect of Drift Coefficient

Figure 5.2a suggests that the maximum deviation occurs within a value between 0.01 and 0.1. This is an implementation specific value, depending on the current MOA implementation. This is an indication that the drift coefficient changes the generation process periodically. That means after a threshold close to 0.05 drift reaches an amount that it starts forming clusters again. This agrees with previously seen cases. In Figure 4.2 it

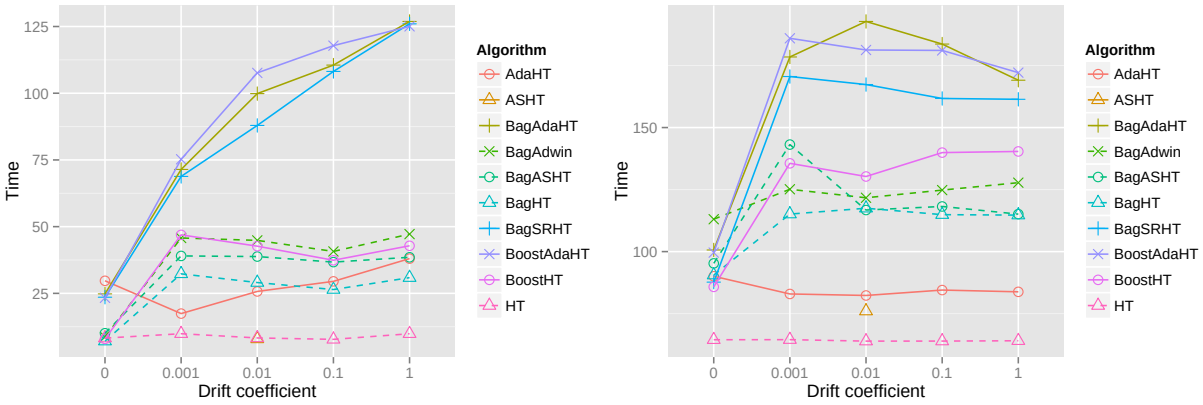
was shown how generated data is distributed on a 2D space. Compared to Figure 4.2c, Figure 4.2d is denser.



(a) (b)

Figure 5.2: Effect of drift coefficient on accuracy using (a) random RBF (b) VS RBF

For the random RBF stream, accuracy drops significantly for all algorithms with the introduction of drift in the system. In this range, bagging using ASHT and SRHT are the only two algorithms where no drift and very small drift have an insignificant (about 2%) change. The accuracy of adaptive HT based approaches drop the least, around 20%; while HT suffers greatly with more than 30% drop in accuracy. Performance improvement over bagging using ASHT with bagging using SRHT is insignificant (within 1-2%).



(a) (b)

Figure 5.3: Effect of drift coefficient on processing time using (a) random RBF (b) VS RBF

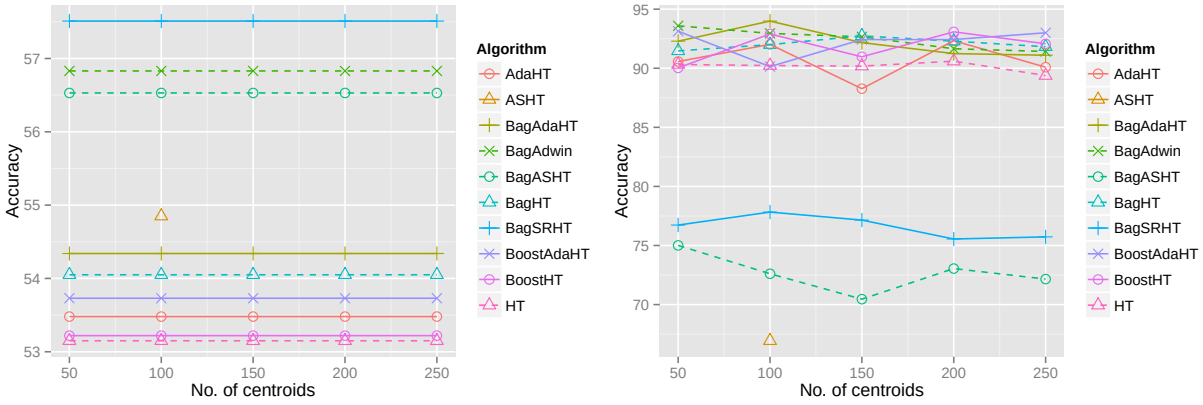
For a variable speed RBF stream, this has less importance as the highest number of data is produced by a few centroids. The performance relies more on those specific centroids than on all centroids' drift. Thus, accuracy remains on the similar level with varying drift. Importantly though, there is improvement in accuracy after drift introduction, contrary to its random RBF counterpart. One plausible explanation would be, with the fewer drift

centroids with higher data volume, it would cover more space, have higher decision rules, and thus achieve higher performance.

Next, on Figure 5.3 comparison of processing time for different drift coefficient is shown. As expected, classifiers that maintains higher number of trees, complete (SRHT bagging) or sub-trees (adaptive HTs), take longer time with drifting data. The reason is with drifting concepts, more alternate trees or bigger trees are needed to be maintained. For the same reason the tree depth, tree size, and memory show similar trends as the time plot (see Appendix A).

5.3.2 Effect of Number of Centroids

The umber of generating centroids is another parameter for data generation. Each centroid in some way simulates concepts. We varied this number from 50 to 200 with steps of 50. No changes in terms of performance metrics is observed for random RBF data sets (Figure 5.4a, 5.5a). For VS RBF streams a slight variation is observed, as depicted in Figure 5.4b, 5.5b.



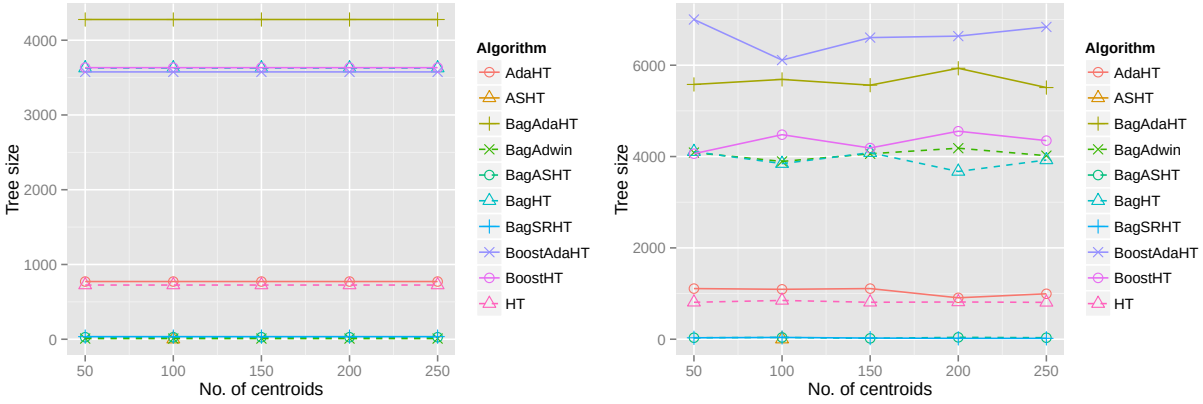
(a) (b)

Figure 5.4: Effect of number of centroids on accuracy using (a) random RBF (b) VS RBF

Tree sizes for all these algorithms fall into 3 general categories. Classifiers with size limitations (ASHT, SRHT) lies at the bottom of the plot, classifiers without any alternate tree maintenance scheme have higher tree sizes, however, those with alternate trees are about 5 times larger in sizes than these classifiers.

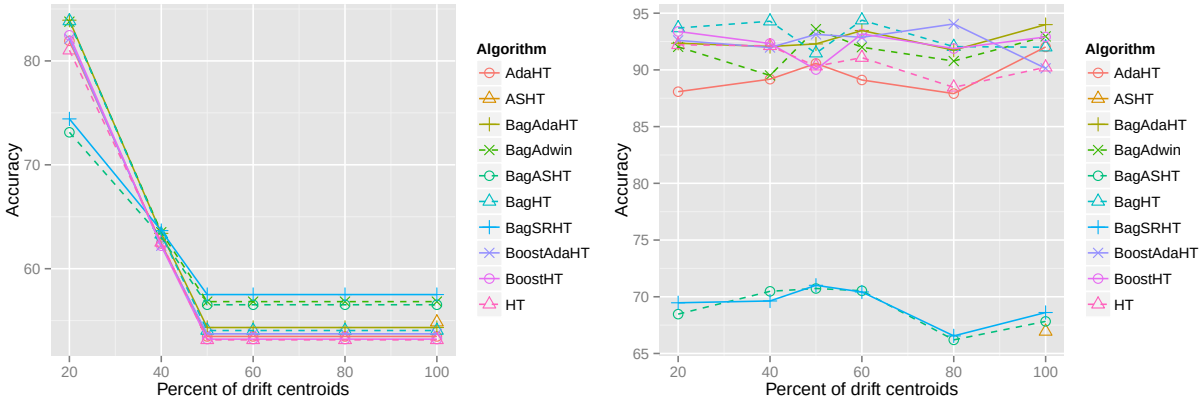
5.3.3 Effect of Percentage of Drifting Centroids

In this set of experiments, we checked the effect of number of drifting centroids. Previously, we have shown the performance for different drifting amounts, and have found that with a relatively higher drifting rate (e.g. 0.01), performances of the algorithms drops substantially. (Figure 5.2). For these experiments, we therefore, kept the drifting coefficient to 0.01 and varied percentage or drifting centroids from 20 to 100. As expected, accuracy and other metrics are found to be highly correlated with the percentage of drifting centroids



(a) (b)

Figure 5.5: Effect of number of centroids on tree size using (a) random RBF (b) VS RBF



(a) (b)

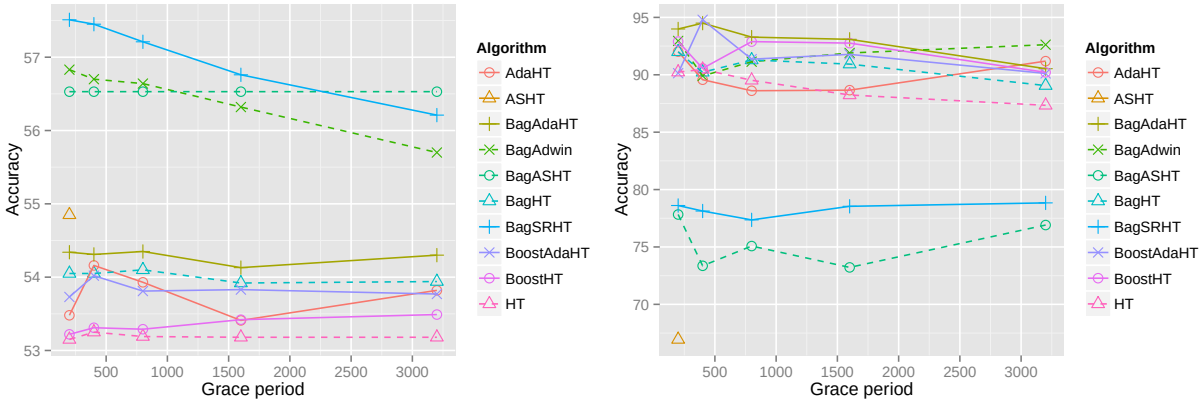
Figure 5.6: Effect of percentage of drift centroid on accuracy using (a) random RBF (b) VS RBF

till it reaches the random guess accuracy of 50% (binary class problem). With 50% of the centroids drifting with a coefficient of 0.01, performance reaches the minimum as shown in the Figure 5.6.

5.3.4 Effect of Grace Period

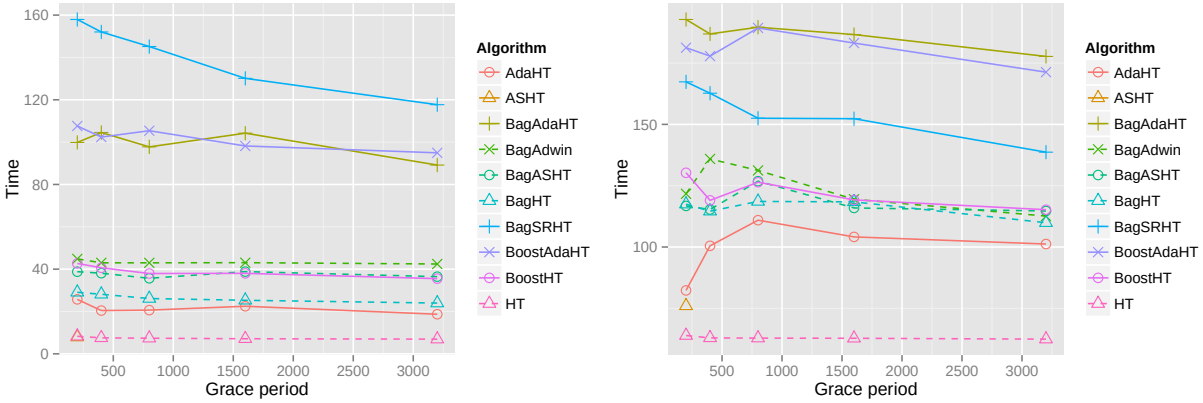
The grace period is the number of instances each node must receive before starting to check for split attribute. Thus, it is expected that with a higher grace period the algorithms will perform faster producing less decision nodes.

We varied the grace period from 200 (MOA default) to 3200 in a geometric rate of 2. Results for accuracy, processing time, and tree size are shown in Figure 5.7, 5.8, and 5.9 respectively. These figures show that for both random RBF and variable speed RBF, the effect of grace period is similar. Processing time and tree size drop about 25% to 33% while the effect on accuracy is negligibly within 0-2% only.



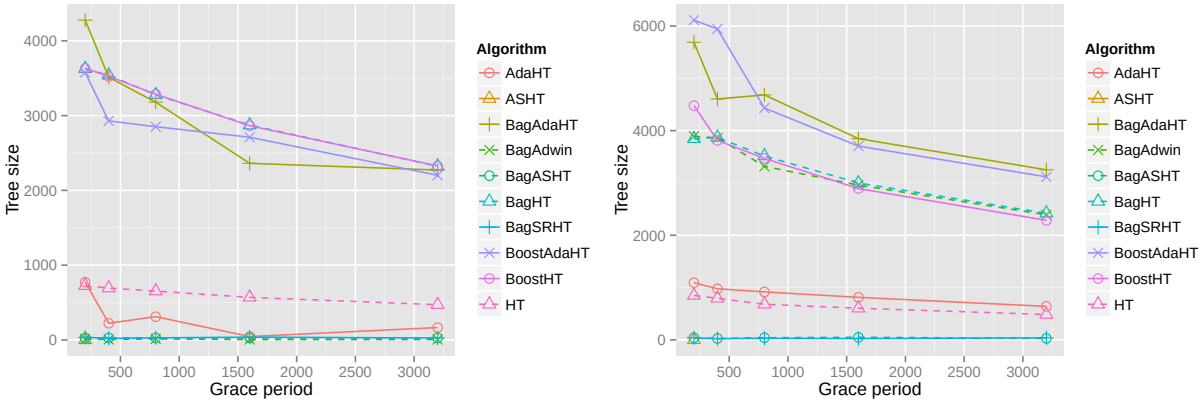
(a) (b)

Figure 5.7: Effect of grace period on accuracy using (a) random RBF (b) VS RBF



(a) (b)

Figure 5.8: Effect of grace period on processing time using (a) random RBF (b) VS RBF

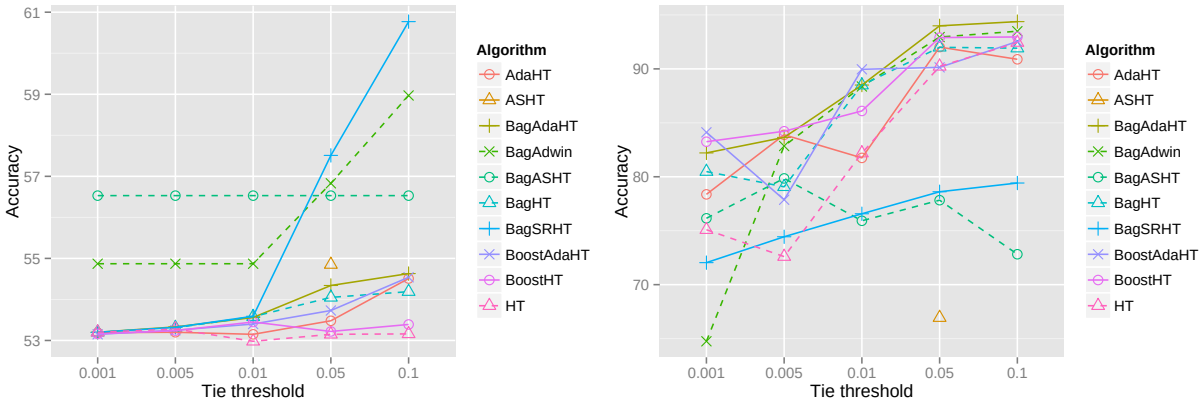


(a) (b)

Figure 5.9: Effect of grace period on tree size using (a) random RBF (b) VS RBF

5.3.5 Effect of Tie Threshold

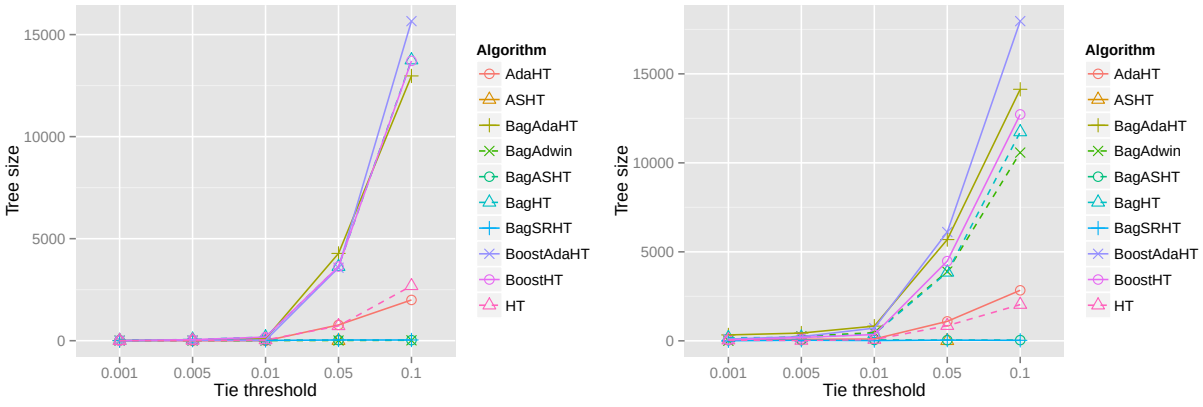
The tie threshold is used to break ties between two similarly good attributes for a split decision. If their gain difference in not greater than the Hoeffding margin, but the margin itself is less than tie threshold then a node is splitted on the best attribute at hand.



(a) (b)

Figure 5.10: Effect of tie threshold on accuracy using (a) random RBF (b) VS RBF

A larger value for the tie threshold ensures that the margin between the best two attributes is higher. As Figure 5.10 shows that the larger values yields better accuracy when we varied tie threshold from 0.001 to 0.1 with discrete intervals. However, in Figure 5.11, it can be seen that this also increases the tree size substantially for adaptive HT based methods. Both random RBF and VS RBF show similar trends, but classifiers perform better on VS RBF. The reason is explained in the section above about effect of the drift coefficient. We used 0.01 as drift coefficients for these experiments.

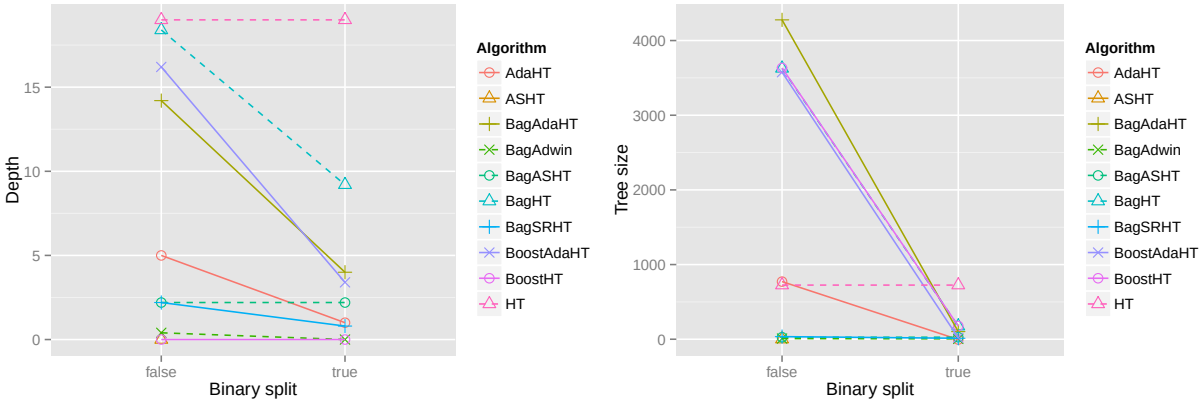


(a) (b)

Figure 5.11: Effect of tie threshold on tree size using (a) random RBF (b) VS RBF

5.3.6 Effect of Binary Split

This set of experiments explores the effect of a binary split. This attribute has more importance in terms of tree structure than on the overall performance. The overall accuracy difference is negligible (about 1-2%) for the non-binary and the binary tree. However, depth and number of decision nodes are greatly affected by this parameter. For example, tree sizes of adaptive HT based approaches drop down from around 5000 nodes to under 1000 nodes when a binary splitting is used. Similarly, depths fall from around 25 to just 10. In Figure 5.12 we have shown the depths and tree sizes for a binary and non-binary split scenario for the random RBF generator. The variable speed RBF generator also produces similar results (see Appendix A).



(a) (b)

Figure 5.12: Effect of binary split on (a) depth (b) tree size

5.3.7 Effect of Ensemble Size

The ensemble method has very little influence on static concepts. Figure 5.13 shows that for both random RBF and VS RBF generators increasing ensemble size achieves no better accuracy. But processing time increases linearly with increasing size.

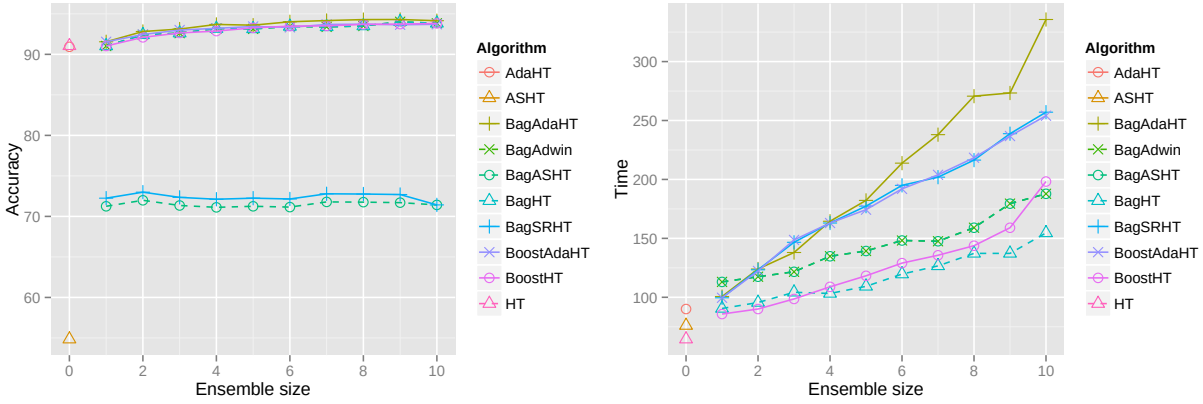
However, with a small drift coefficient of 0.001, algorithms demonstrate higher accuracy with the increasing ensemble size (Figure 5.14).

5.3.8 Other Parameters

Experiments are also performed for the maximum allowed size, the size of the first tree in an ensemble (other are multiples of the first one), and the reset vs pruning approaches. Their effect in our settings is found to be negligible. Evaluation plots are given in Appendix A.

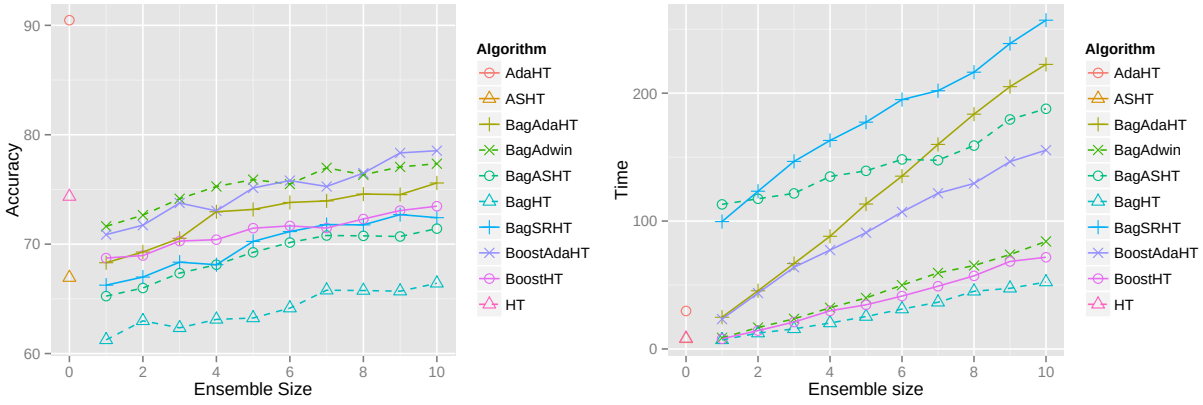
5.4 Comparative Analysis of Timeline

In the last section, we have presented performance metrics of the algorithms in terms of their parameters, and compared the algorithms based on the overall performance. However,



(a) (b)

Figure 5.13: Effect of ensemble size for static concepts on (a) accuracy (b) processing time



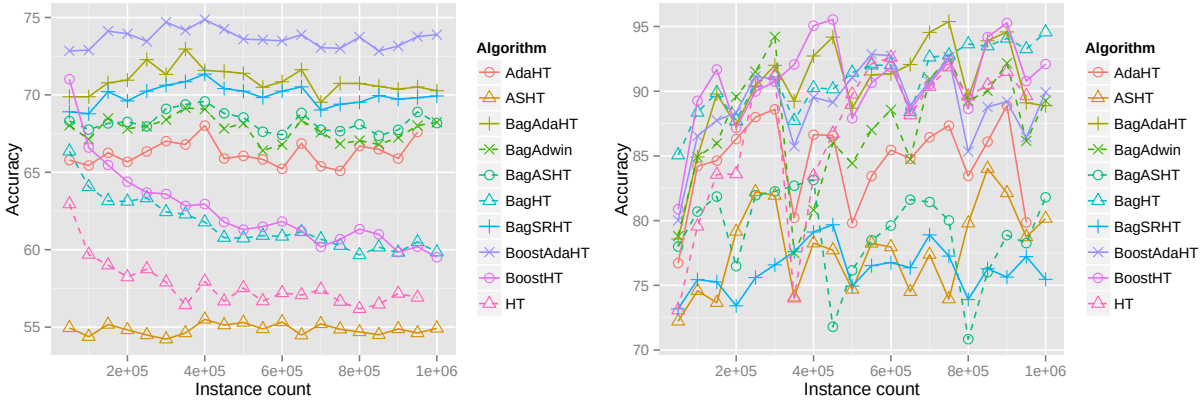
(a) (b)

Figure 5.14: Effect of ensemble size for drifting concepts on (a) accuracy (b) processing time

with the final output of performances, we do not get the idea of how a tree reset or pruning affects the classifiers. Moreover, it is unclear that how bagging with SRHT is an improvement over bagging with ASHT. Thus, in this section, we present the analysis based on a windowed evaluation in a timeline manner. For this, we have used a windowed evaluation approach with 50,000 as the size of the window. Performances are sampled at each 50,000 instances. For this set of experiments, the used parameter values are: drift coefficient 0.001, grace period 200, tie threshold 0.05, percentage of drifting coefficient 100, ensemble size 5, binary split, and resettable.

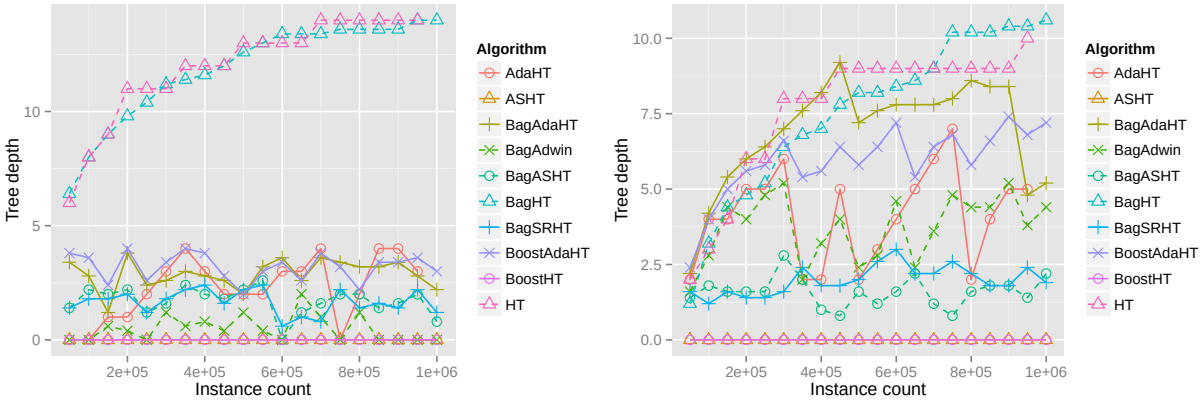
5.4.1 Performance Over Time

Figure 5.15 shows the accuracy of the algorithms in different time of the window. Figure 5.16 presents the depths of the trees at those corresponding times. For the random RBF generator, a lot of fluctuations in both accuracy (Figure 5.15a) and tree depth (Fig-



(a) (b)

Figure 5.15: Accuracy over time for (a) Random RBF (b) VS RBF



(a) (b)

Figure 5.16: Tree depth over time for (a) Random RBF (b) VS RBF

ure 5.16a) can be seen. For the VS RBF generator, these fluctuations are bigger. The reason is presumably the high number of instances being generated from small number of concepts in VS RBF, and those concepts are constantly disappearing. However, an important thing to note here is the relationship between the drop in the depth and the drop in the accuracy for the random RBF generator, especially the bagging with a ASHT classifier. On the other hand, bagging with SRHT achieves a bit higher accuracy with a more stable classification rate. For VS RBF generator, bagging with ASHT seems to perform a little better than SRHT bagging but SRHT bagging is more stable. Some other classifiers are performing better (Figure 5.15b), however, lead to an over-fitting scenario as the tree depth is constantly rising (Figure 5.16b).

This is essentially the goal of our new approach: to have a more stable ensemble such that deleting a classifier will not significantly affect accuracy for a certain period of time. Moreover, Figure 5.17 shows that bagging with SRHT improves performances for slower concepts (see Figure 4.3).

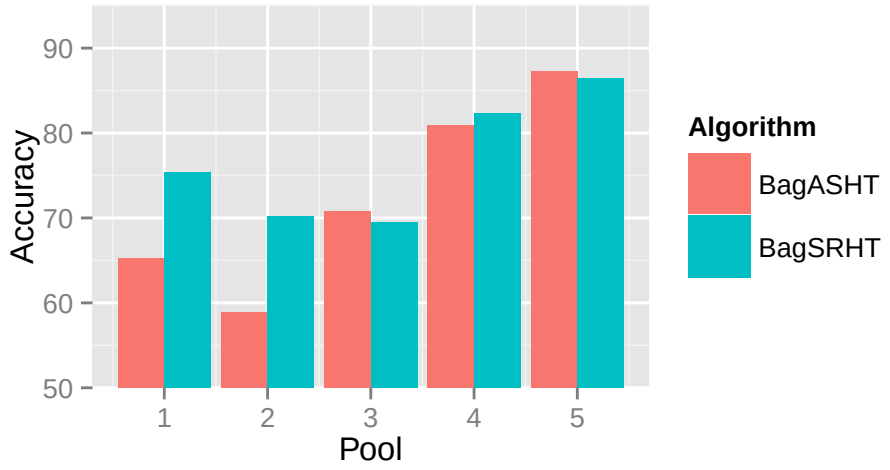
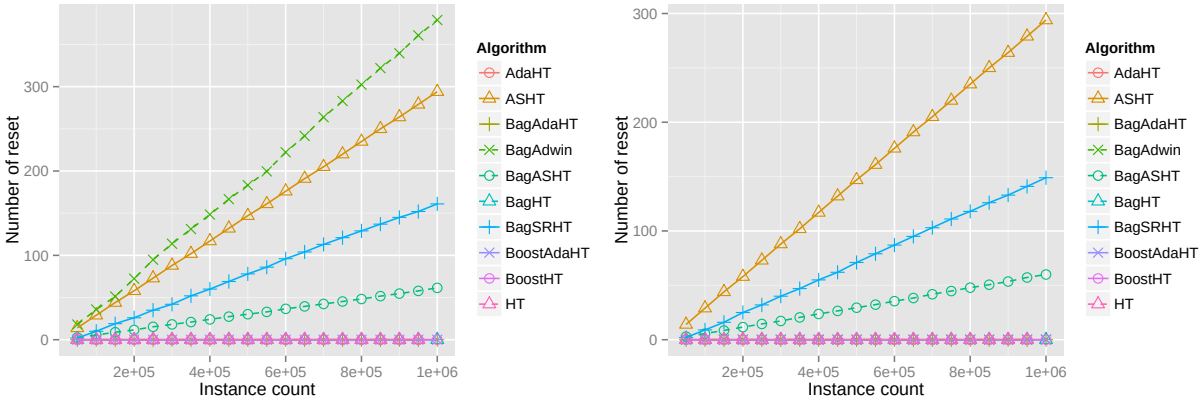


Figure 5.17: Accuracy within each data generating pool

Among other metrics, the number of reset count found to be linearly increasing with time for both generators (Figure 5.18). Similarly, the processing time shows a linearly increasing trend over time (Figure 5.19). Kappa statistics shows higher variations for VS RBF than for random RBF (Figure 5.20).



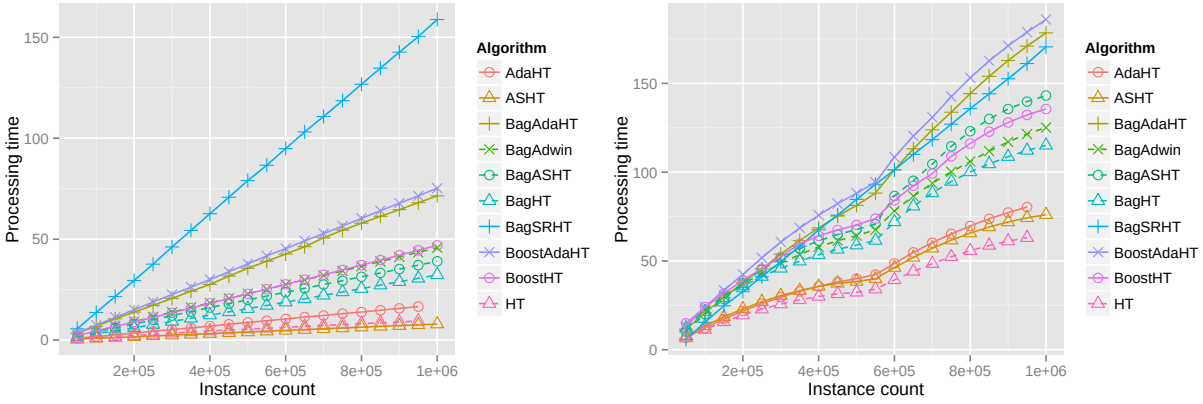
(a) (b)

Figure 5.18: Number of tree being reset over time for (a) Random RBF (b) VS RBF

5.5 Discussion on the Experimental Evaluation

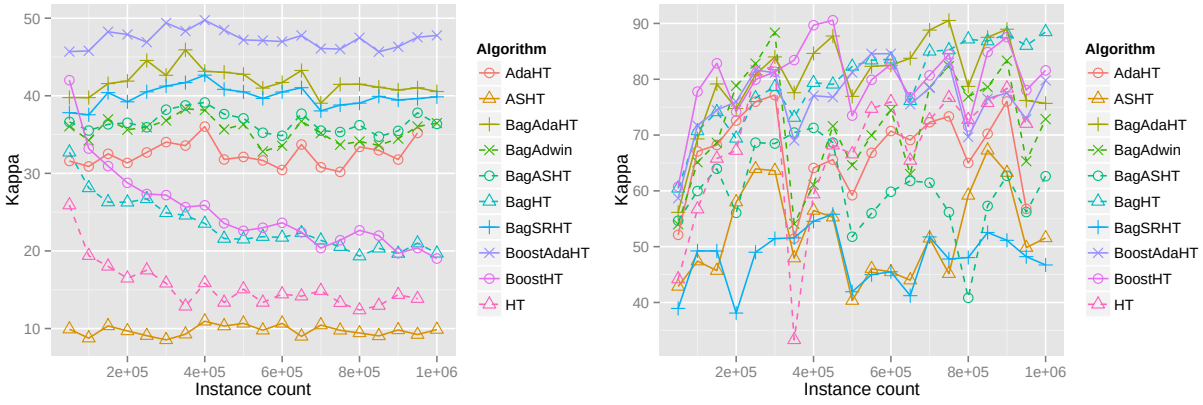
We started this chapter by comparing stream learners with state-of-the-art batched learners. We showed that for the census income data set [Kohavi, 1996] of about 49 thousand instances stream learners perform close to the best accuracies obtained by batched learners. This, however, cannot be used to conclude that stream learners would always show such a performance for other data sets.

Next, we have analyzed each of the algorithms including our new approach by varying



(a) (b)

Figure 5.19: Processing time over time for (a) Random RBF (b) VS RBF



(a) (b)

Figure 5.20: Kappa over time for (a) Random RBF (b) VS RBF

the parameters within a wide range. We also have presented findings on the effect of different parameters of the data generation processes. It was found that the drift coefficient is inversely related with accuracy for 0 to 0.01, after which accuracy starts to increase again. Hence, this indicates that its effect on the data generation process is periodic. Performances of the learners are found to be nearly independent of the number of centroids used for a fairly high drift of 0.01. On the other hand, the percentage of drift centroids is inversely related to performance, and can bring accuracy down to the 50% for a binary class problem with only 50% drifting centroids. After that, increasing number of drifting centroids cannot not affect the classifier performance.

From classifier parameters, a grace period is found to be very effective in reducing processing time and tree size, sacrificing a negligible amount of accuracy, which is essentially the motive to use a grace period. The use of higher threshold achieves higher accuracy while considerably increasing the tree size. It has also been shown that binary tree can achieve same accuracy but reducing tree size significantly. For ensemble based approaches,

the ensemble size is found to be effective in drifting conditions while for the static conditions it had no effect. Other parameters have very little impact in our experimental setup.

Finally, we have presented the windowed performance evaluation or the timeline of performances of algorithms. It is here, where we can clearly distinguish and effectively prove the effectiveness of our new approach. While analyzing parameters, we have seen that bagging SRHT mostly performs better than bagging with ASHT. However, with the windowed analysis, we demonstrated how in ASHT bagging ensemble performs poorly after the trees being reset. Experimental results showed that SRHT bagging overcomes this issue and keeps a stable classifier though-out the time. Additionally, a breakdown of performance for different pools evidentially proved that SRHT is more effective for slower streams. Thus SRHT fairly realizes the motivation we presented at the beginning.

Chapter 6

Conclusion

In this thesis, we have dealt with a well-known problem of supervised classification of data instances in a streaming environment. Particularly, we wanted to observe the behavior of the current decision tree based state-of-the-art approaches for data streams that are composed of irregular sub-streams depending on the number of instances. The motivation was that often a large volume of data is produced by only few sources within a stream and they dominate the classifier's decision rules. This is especially problematic for the sources that produce few instances. We have analyzed common stream generators which are used for testing stream learners. It was found that most of these generators generate data in a randomized manner. Thus, we have devised a new stream generator where the volume of data produced by the source is related to its lifespan (time it remains active).

We then have tested current methods for both currently available and our new generators. The theoretical aspects are supported with the evidence from a large set of experimental data. It was found that some of the existing methods, e.g. Hoeffding tree, adaptive Hoeffding tree, and their boost variants, perform well, but lead to over-fitting. This problem is solved by using an ensemble approach with a maximum size limitation in bagging with adaptive size Hoeffding tree. This method, however, is found to have a limitation. Every time a larger tree is being reset in the ensemble, the classification accuracy for instances following immediately drops significantly.

With these observations, we developed a new bagging approach to solve the issue, which in principle works similarly to the bagging with ASHT approach. In the new approach we have delayed the reset process while starting to learn a new classifier. We have also used the ADWIN change detection approach within each model. In the ASHT bagging approach ADWIN is not used. With these few changes, we have managed to achieve smoother performance curves.

In summary, the major contributions of this thesis are as follows:

- A new look at the composition of various real-world streams e.g. social networks. Based on that we developed a new stream generation approach.
- A new algorithm, carry-over bagging with SRHT, to solve the limitations of ASHT bagging approach.

- An extensive empirical study to compare the discussed methods.
- A comprehensive survey of the existing literatures.

6.1 Future Works & Open Issues

The motivation of this thesis was taken from all text based examples. However, to keep the thesis within the scope, the experimental evaluations were performed using numeric data sets. We would like to extend this work and apply it in real world data streams.

In our current solution, we eventually used every instances to learn the model. An alternate option would be to balance the model by only training through a set of well-selected instances. The selection process would try to balance among data sources. That means, learning from a small subset of faster sources and using most instances from slower sources. We would like to investigate this approach in the future.

Bibliography

[Abdulsalam et al., 2008] Abdulsalam, H., Skillicorn, D. B., and Martin, P. (2008). Classifying evolving data streams using dynamic streaming random forests. pages 643–651.

[Abdulsalam et al., 2011] Abdulsalam, H., Skillicorn, D. B., and Martin, P. (2011). Classification using streaming random forests. 23(1):22–36.

[Aggarwal et al., 2003] Aggarwal, C., Han, J., Wang, J., and Yu, P. (2003). Clustream: A framework for clustering evolving data streams. In *VLDB*.

[Aggarwal et al., 2004] Aggarwal, C., Han, J., Wang, J., and Yu, P. (2004). On-demand classification of data stream. In *ACM KDD*, pages 503–508.

[Aggarwal, 2003] Aggarwal, C. C. (2003). A framework for diagnosing changes in evolving data streams. In *ACM SIDMOD*, pages 575–586.

[Asuncion and Newman, 2007] Asuncion, A. and Newman, D. (2007). Uci machine learning repository. `http://archive.ics.uci.edu/ml/`.

[Bifet et al., 2010a] Bifet, A., Frank, E., Holmes, G., and Pfahringer, B. (2010a). Accurate ensembles for data streams: Combining restricted hoeffding trees using stacking. 13:225–240.

[Bifet et al., 2010b] Bifet, A., Holmes, G., Kirkby, R., and Pfahringer, B. (2010b). Moa: Massive online analysis. 11:1601–1604.

[Bifet et al., 2010c] Bifet, A., Holmes, G., and Pfahringer, B. (2010c). Leveraging bagging for evolving data streams. In *ECML PKDD*, pages 135–150.

[Bifet et al., 2009] Bifet, A., Holmes, G., Pfahringer, B., Kirkby, R., and Gavaldà, R. (2009). New ensemble methods for evolving data streams. In *SIGKDD*, pages 139–148.

[Botnet, 2012] Botnet, C. (2012). Port scanning using insecure embedded devices. `http://www.huffingtonpost.co.uk/2013/05/15/global-internet-gif_n_3277404.html`.

[Breiman, 1993] Breiman, L. (1993). Stacked regression.

[Breiman, 1994] Breiman, L. (1994). Bagging prediction.

[Breiman, 1999] Breiman, L. (1999). Random forest.

[Breiman et al., 1984] Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). Classification and regression trees.

[Brzezinski and Stefanowski, 2011] Brzezinski, D. and Stefanowski, J. (2011). Accuracy updated ensemble for data streams with concept drift. In *HAIS*, pages 155–163.

[Castillo et al., 2003] Castillo, G., Gama, J., and Medas, P. (2003). Adaptation to drifting concepts. In *Progress in Artificial Intelligence*, pages 279–293.

[Catlett, 1991] Catlett, J. (1991). Megainduction: Machine learning on very large databases. In *PhD thesis, University of Sydney.*

[Chen et al., 2002] Chen, Y., Dong, G., Han, J., Wah, B. W., , and Wang, J. (2002). Multidimensional regression analysis of time-series data streams. In *VLDB*, pages 323–334.

[Dasu et al., 2004] Dasu, T., Krishnan, S., Venkatasubramanian, S., and Yi, K. (2004). An information-theoretic approach to detecting changes in multi-dimensional data streams. In *Duke University Technical Report CS-2005-06*, pages 180–191.

[Ding et al., 2002] Ding, Q., Ding, Q., and Perrizo, W. (2002). Decision tree classification of spatial data streams using peano count trees. In *ACM Symposium on Applied Computing*, pages 413–417.

[Domingos and Hulten, 2000] Domingos, P. and Hulten, G. (2000). Mining high-speed data streams. In *Proceedings of the ACM KDD.*

[Drucker et al., 1994] Drucker, H., Cortes, C., Jackel, L. D., LeCun, Y., and Vapnik, V. (1994). Boosting and other ensemble methods. 6(6):1289–1301.

[Fisher, 1922] Fisher, R. A. (1922). On the mathematical foundations of theoretical statistics. 222:309–368.

[Freund and Schapire, 1997] Freund, Y. and Schapire, R. (1997). A decision theoretic generalization of on-line learning and an application to boosting. 55(1):119–139.

[Gama et al., 2004a] Gama, J., Medas, P., Castillo, G., and Rodrigues, P. (2004a). Learning with drift detection. In *SBIA Brazilian Symposium on Artificial Intelligence*, pages 286–295.

[Gama et al., 2004b] Gama, J., Medas, P., and Rocha, R. (2004b). Forest trees for on-line data. In *Symposium on Applied computing*, pages 632–636.

[Gama et al., 2005] Gama, J., Medas, P., and Rodrigues, P. (2005). Learning decision trees from dynamic data streams. In *Symposium on Applied computing*, pages 573–577.

[Gama et al., 2003] Gama, J., Rocha, R., and Medas, P. (2003). Accurate decision trees for mining high-speed data streams. In *SIGKKD*, pages 523–528.

[Ganti et al., 2002] Ganti, V., Gehrke, J., and Ramakrishnan, R. (2002). Mining data streams under block evolution. 3(2):1–10.

[Hansen and Salamo, 1990] Hansen, L. K. and Salamo, P. (1990). Neural network ensembles. 12(10):993–1000.

[Hoeffding, 1963] Hoeffding, W. (1963). Probability inequalities for sums of bounded random variables. 58:13–30.

[Hulten et al., 2001] Hulten, G., Spencer, L., and Domingos, P. (2001). Mining time changing data stream. In *ACM KDD*, pages 97–106.

[Kifer et al., 2004] Kifer, D., Ben-David, S., and Gehrke, J. (2004). Detecting changes in data streams. In *VLDB*, pages 180–191.

[Klinkenberg and Renz, 1998] Klinkenberg, R. and Renz, I. (1998). Adaptive information filtering: Learning in the presence of concept drift. In *Learning for Text Categorization*, pages 33–40.

[Kohavi, 1996] Kohavi, R. (1996). Scaling up the accuracy of naive-bayes classifiers: a decision-tree hybrid. In *Knowledge Discovery and Data Mining*.

[Krogh and Vedelsby, 1995] Krogh, A. and Vedelsby, J. (1995). Neural network ensembles, cross validation and active learning. pages 231–238.

[Langley et al., 1992] Langley, P., Iba, W., and K.Thompson (1992). An analysis of bayesian classifiers. In *National Conference of Artificial Intelligence*, pages 223–228.

[Last, 2002] Last, M. (2002). Online classification of non-stationary data streams. 6(2):127–147.

[Matuszyk et al., 2013] Matuszyk, P., Krempl, G., and Spiliopoulou, M. (2013). Correcting the usage of the hoeffding inequality in stream mining. In *Advances in Intelligent Data Analysis*, pages 298–309.

[Mehta et al., 1996] Mehta, M., Agrawal, A., and Rissanen, J. (1996). Sliq: A fast scalable classifier for data mining. In *Extending Database Technology*, pages 18–32.

[Oza, 2001] Oza, N. C. (2001). Online ensemble learning. In *Ph.D. thesis, Department of EECS, UC Berkeley*.

[Oza and Russell, 2001] Oza, N. C. and Russell, S. (2001). Online bagging and boosting. In *Artificial Intelligence and Statistics*, pages 105–112.

[Page, 1954] Page, E. S. (1954). Continuous inspection scheme. 41(1/2):100–115.

[Parhami, 1996] Parhami, B. (1996). Voting algorithms. 43(4):617–629.

[Pelossof et al., 2008] Pelossof, R., Jones, M., Vovsha, I., and Rudin, C. (2008). Online coordinate boosting.

[Quinlan, 1993] Quinlan, J. R. (1993). C4.5: Programs for machine learning.

[Rojas, 1996] Rojas, R. (1996). *Neural Networks*.

[Rutkowski et al., 2013] Rutkowski, L., Pietruczuk, L., Duda, P., and Jaworski, M. (2013). Decision trees for mining data streams based on the mcdiarmid's bound. 25(6):1272–1279.

[Schapire, 1990] Schapire, R. (1990). Strength of weak learnability. 5(2):197–227.

[Schlimmer and Granger, 1986] Schlimmer, J. C. and Granger, R. H. (1986). Incremental learning from noisy data. 1(3):317–354.

[Shafer et al., 1996] Shafer, J. C., Agrawal, R., and Mehta, M. (1996). Sprint: A scalable parallel classifier for data mining. In *VLDB*, pages 544–555.

[Shannon, 2001] Shannon, C. E. (2001). A mathematical theory of communication. 5(1):3–55.

[Street and Kim, 2001] Street, W. N. and Kim, Y. (2001). A streaming ensemble algorithm for large-scale classification. In *KDD*, pages 377–382.

[Topsy, 2015] Topsy (2015). Topsy social analytics. `http://topsy.com/analytics`.

[Tumer and Ghosh, 1999] Tumer, K. and Ghosh, J. (1999). Linear and order statistics combiners for pattern classification. In *A. J. C. Sharkey, editor, Combining Artificial Neural Nets: Ensemble and Modular Multi-Net Systems*, pages 127–162.

[Tumer and Oza, 1999] Tumer, K. and Oza, N. C. (1999). Decimated input ensembles for improved generalization. In *IJCNN*, pages 105–112.

[Wang et al., 2003] Wang, H., Fan, W., Yu, P. S., and Han, J. (2003). Mining concept-drifting data streams using ensemble classifiers. In *SIGKDD*, pages 226–235.

[Wolpert, 1992] Wolpert, D. H. (1992). Stacked generalization. 5:244–259.

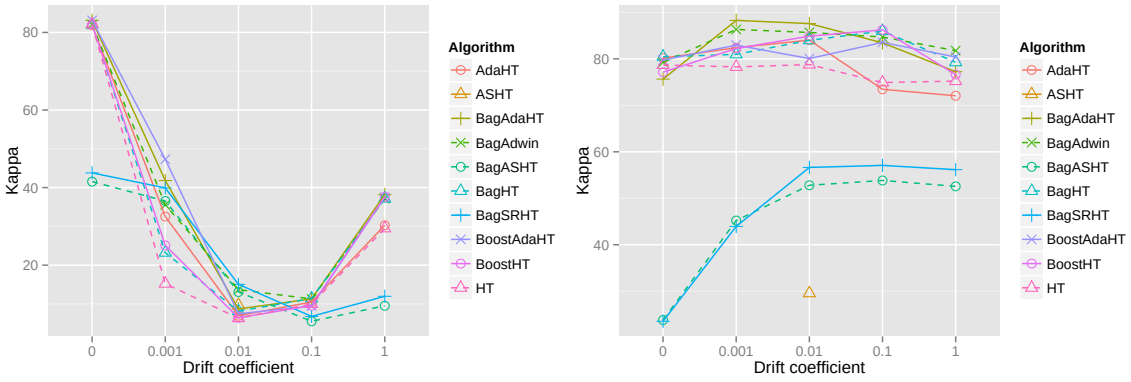
[Zeira et al., 2004] Zeira, G., Maimon, M., Last, M., and Rokach, L. (2004). Data mining in time series databases. 57:101–125.

Appendices

Appendix A

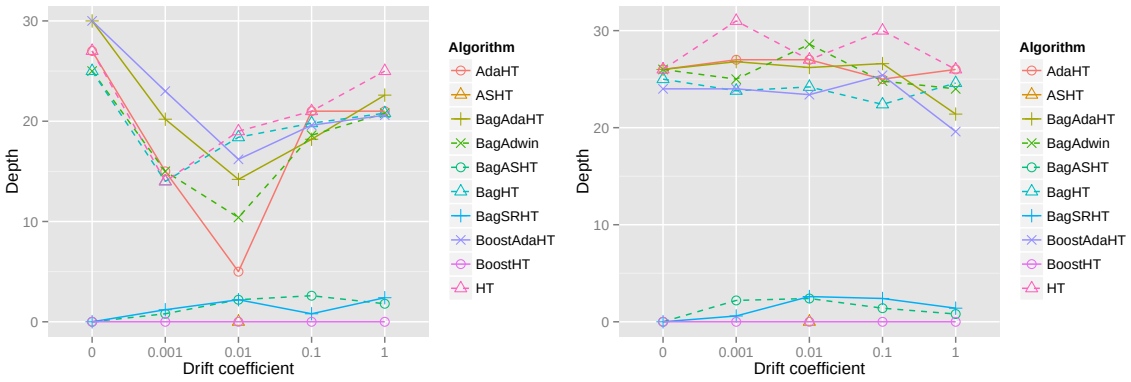
Additional Plots

Parameter Analysis



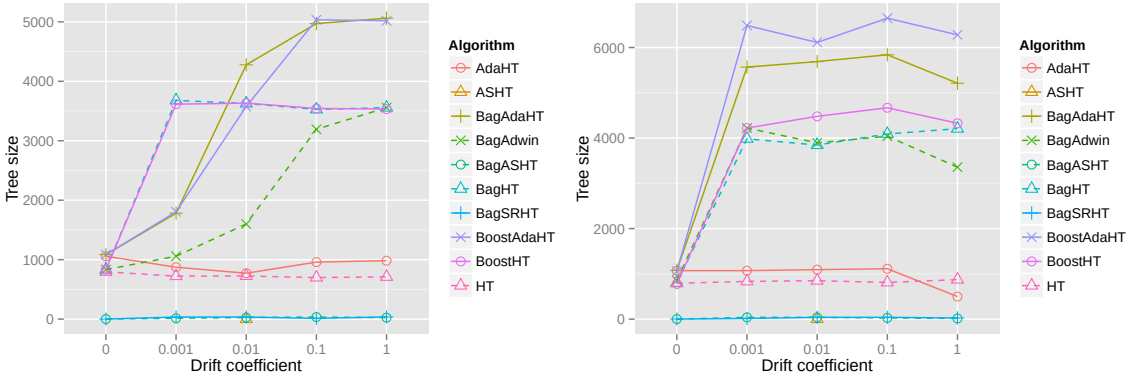
(a) (b)

Figure A.1: Effect of drift coefficient on Kappa statistics using (a) random RBF (b) VS RBF



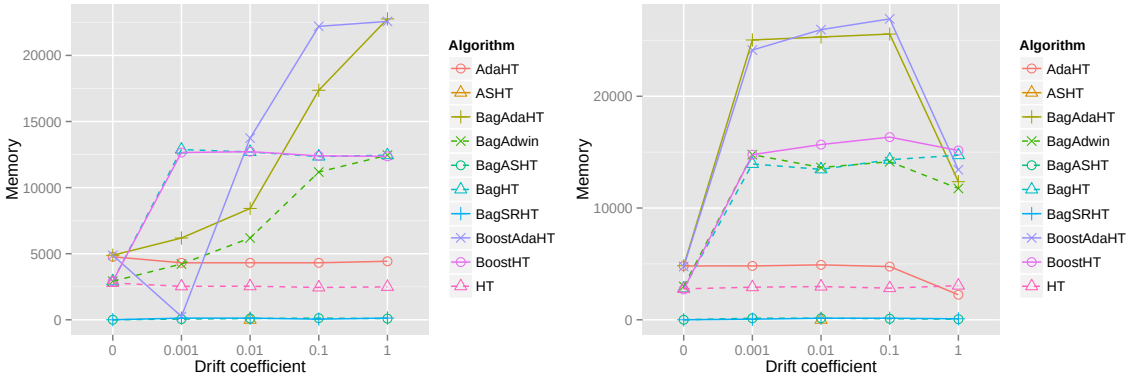
(a) (b)

Figure A.2: Effect of drift coefficient on tree depth using (a) random RBF (b) VS RBF



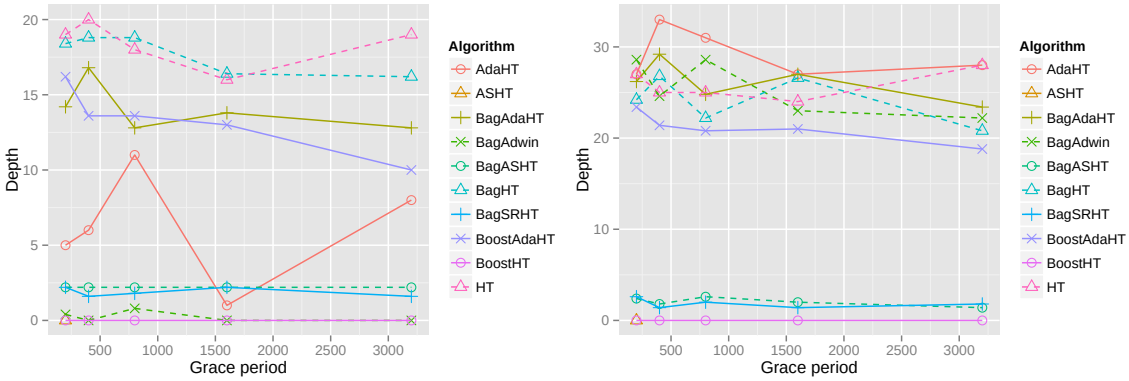
(a) (b)

Figure A.3: Effect of drift coefficient on tree size using (a) random RBF (b) VS RBF



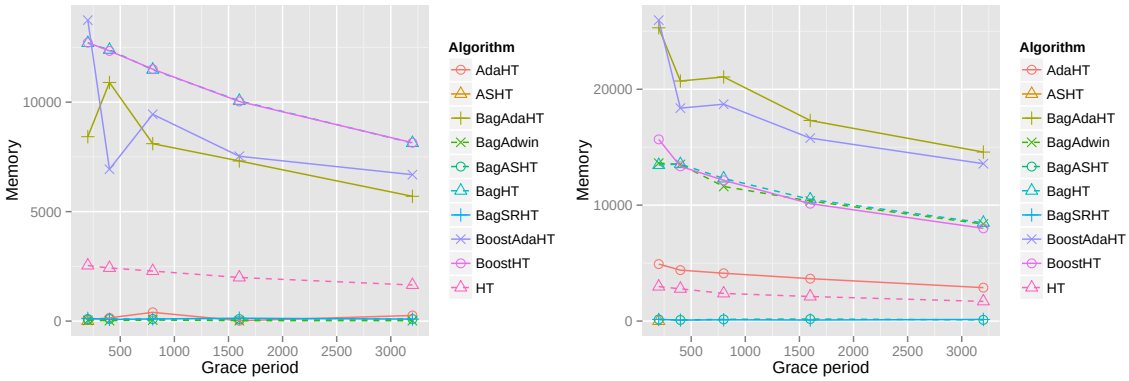
(a) (b)

Figure A.4: Effect of drift coefficient on used memory using (a) random RBF (b) VS RBF



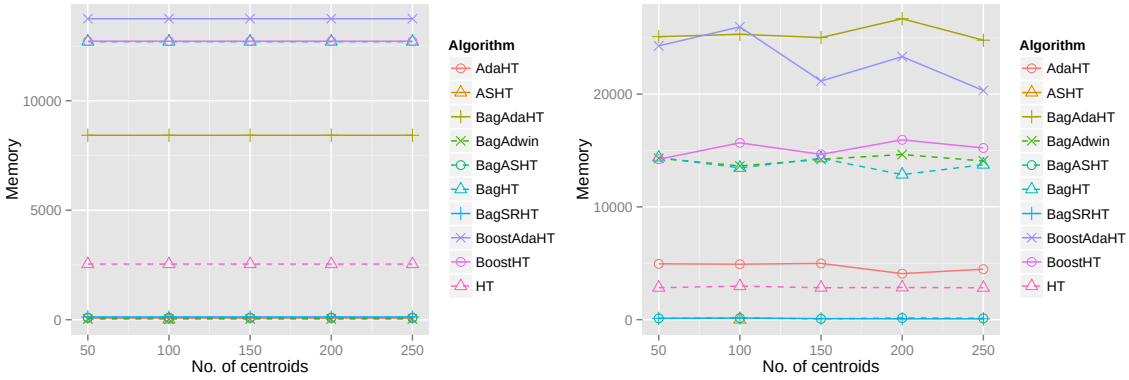
(a) (b)

Figure A.5: Effect of grace period on tree depth using (a) random RBF (b) VS RBF



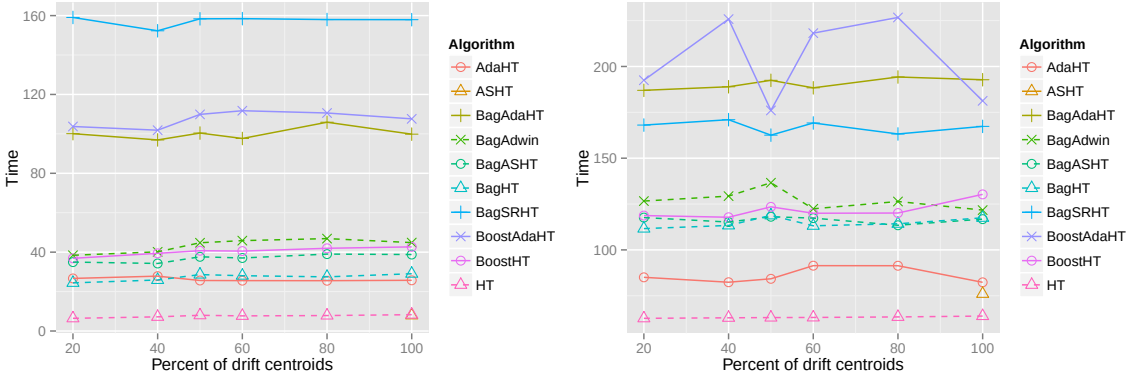
(a) (b)

Figure A.6: Effect of grace period on used memory using (a) random RBF (b) VS RBF



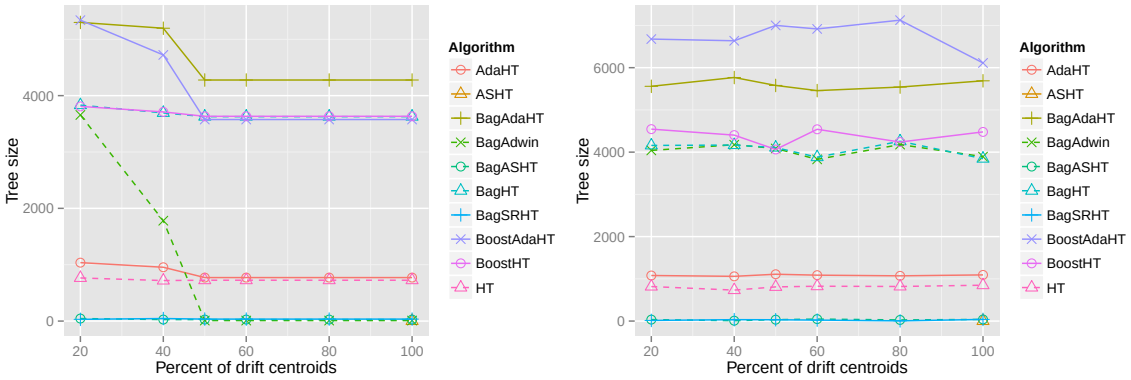
(a) (b)

Figure A.7: Effect of number of centroids on used memory using (a) random RBF (b) VS RBF



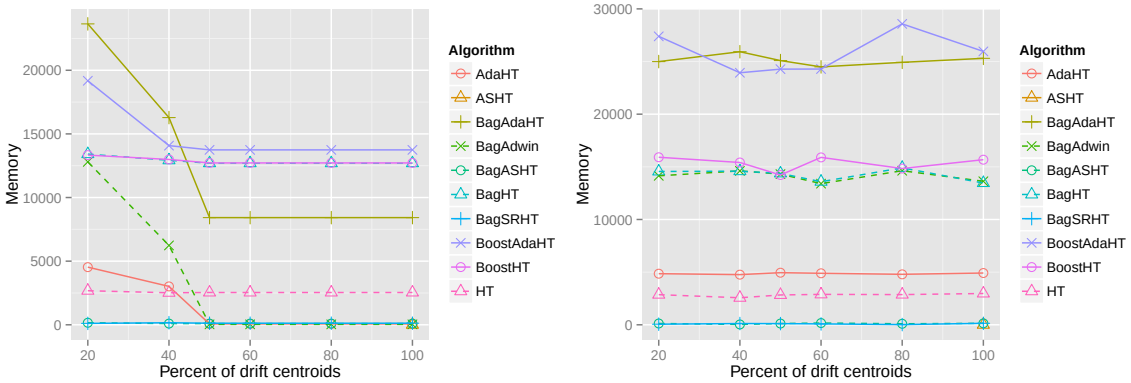
(a) (b)

Figure A.8: Effect of percentage of drift centroids on processing time using (a) random RBF (b) VS RBF



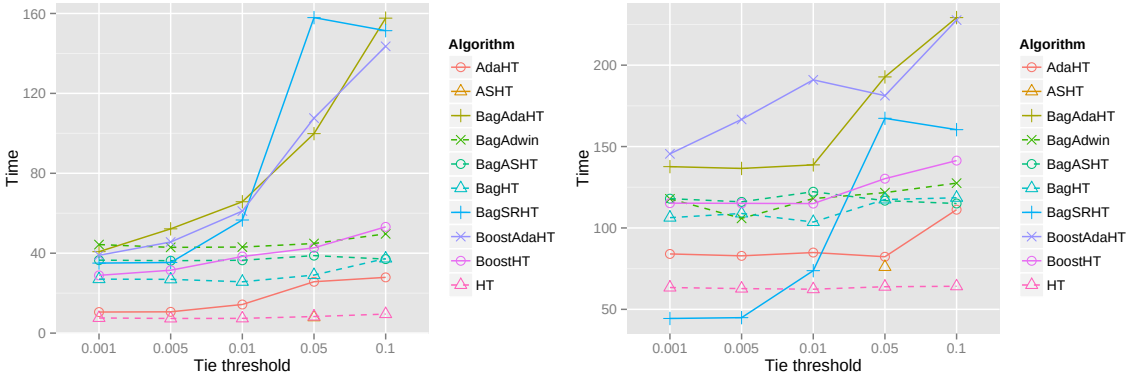
(a) (b)

Figure A.9: Effect of percentage of drift centroids on tree size using (a) random RBF (b) VS RBF



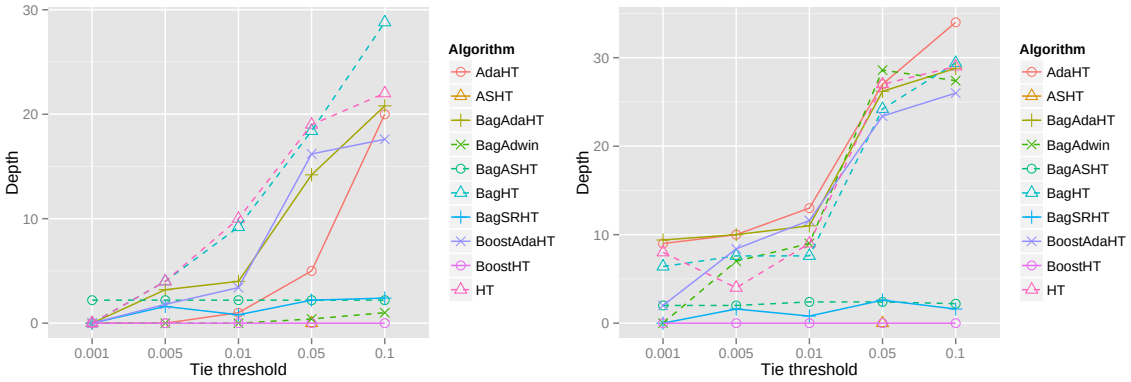
(a) (b)

Figure A.10: Effect of percentage of drift centroids on used memory using (a) random RBF (b) VS RBF



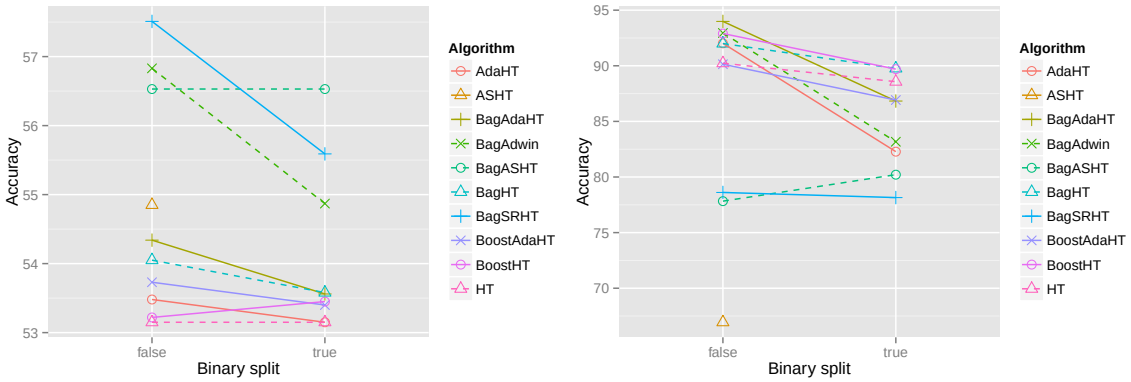
(a) (b)

Figure A.11: Effect of tie threshold on processing time using (a) random RBF (b) VS RBF



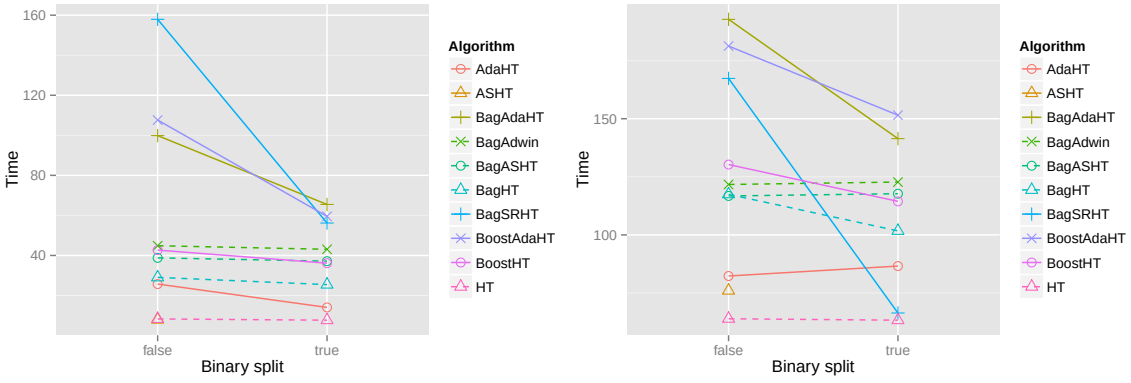
(a) (b)

Figure A.12: Effect of tie threshold on tree depth using (a) random RBF (b) VS RBF



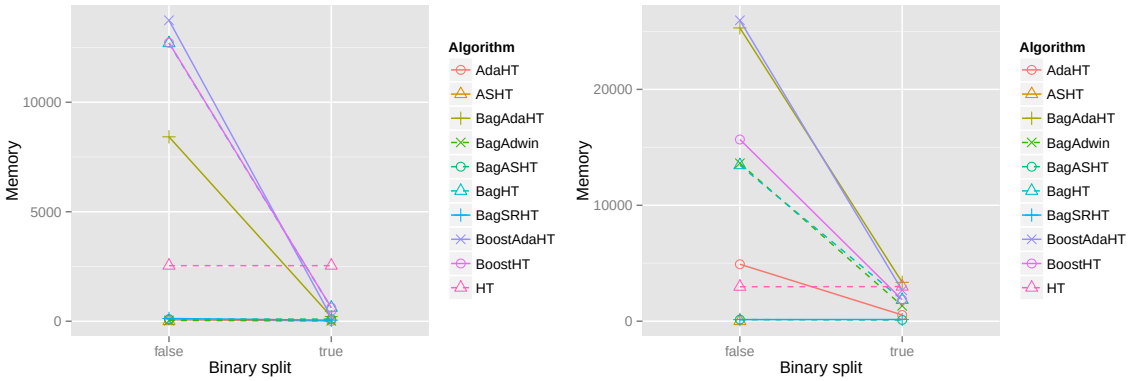
(a) (b)

Figure A.13: Effect of binary split centroids on accuracy using (a) random RBF (b) VS RBF



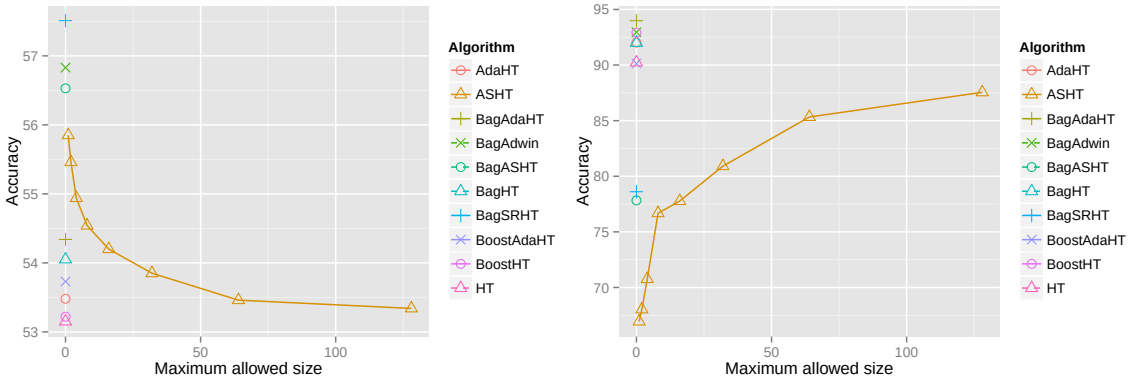
(a) (b)

Figure A.14: Effect of binary split centroids on processing time using (a) random RBF (b) VS RBF



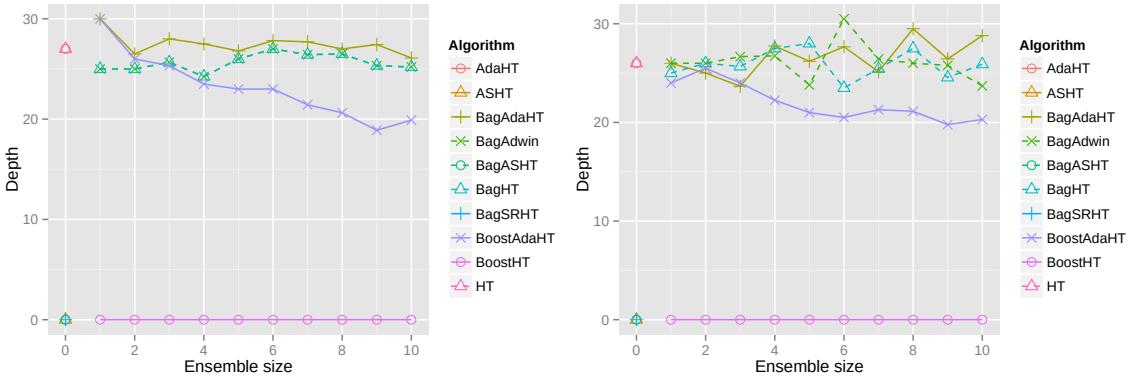
(a) (b)

Figure A.15: Effect of binary split on used memory using (a) random RBF (b) VS RBF



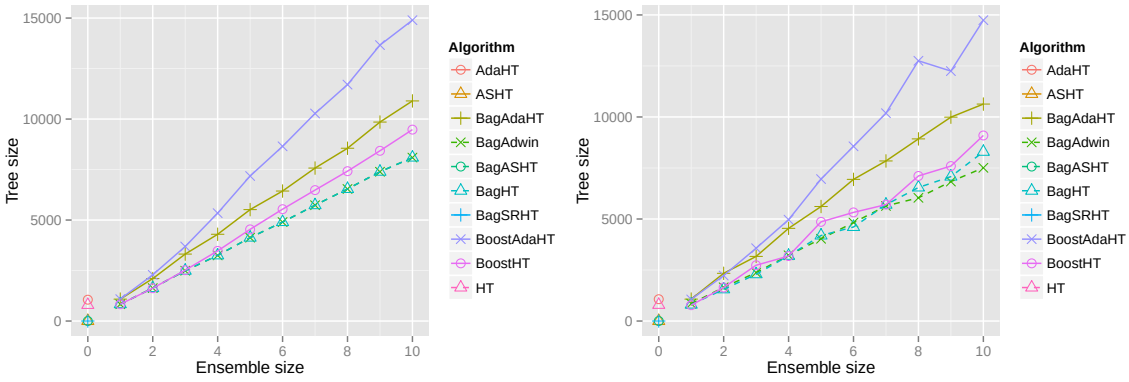
(a) (b)

Figure A.16: Effect of maximum allowed size of tree on accuracy using (a) random RBF (b) VS RBF



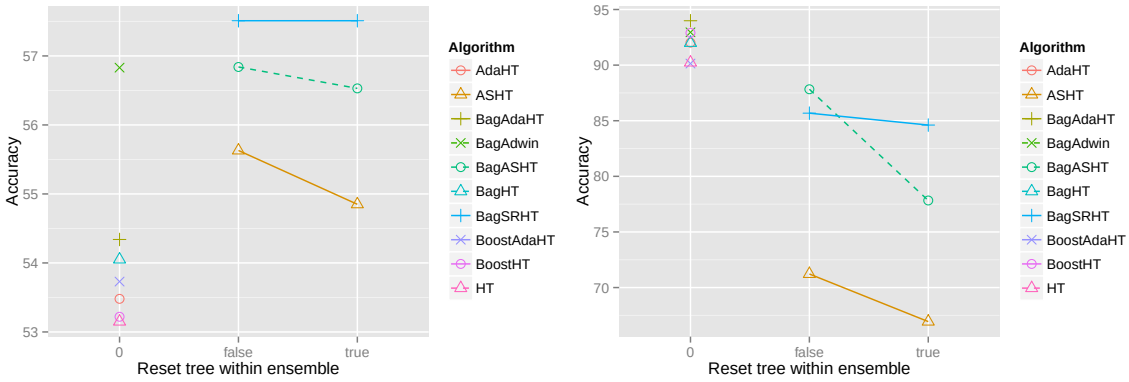
(a) (b)

Figure A.17: Effect of ensemble size on tree depth using (a) random RBF (b) VS RBF



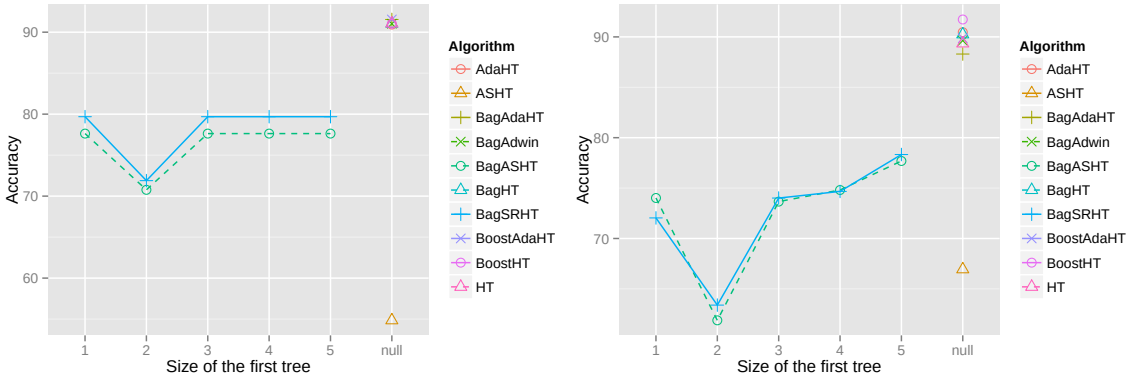
(a) (b)

Figure A.18: Effect of ensemble size on tree size using (a) random RBF (b) VS RBF



(a) (b)

Figure A.19: Effect of tree reset (within ensemble) on accuracy using (a) random RBF (b) VS RBF



(a) (b)

Figure A.20: Effect of first tree size centroids on accuracy using (a) random RBF (b) VS RBF

Comparative Analysis of Timeline

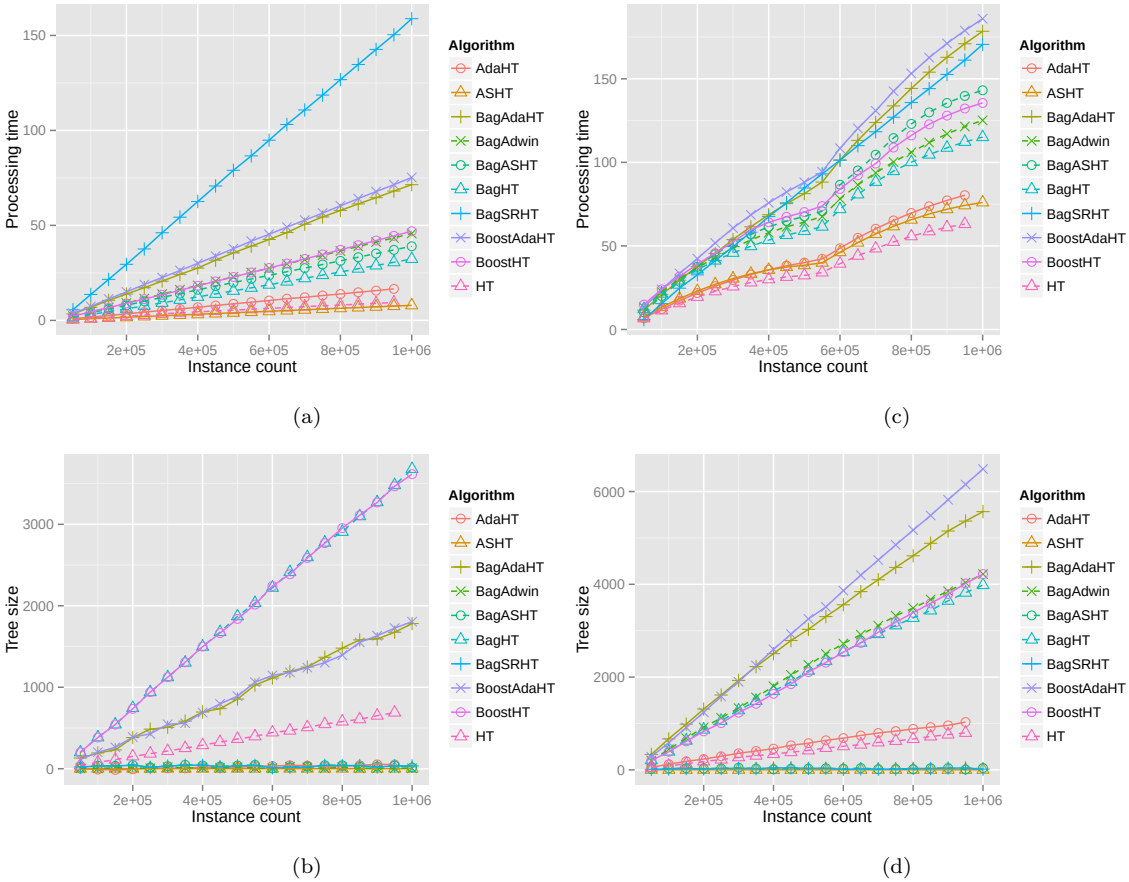


Figure A.21: (a,c) time, (b,d) tree size over time for (a,b) random RBF, (c,d) VS RBF

Appendix B

Extended Related Works

Stream Learning

In a recent approach, [Rutkowski et al., 2013] argued that using McDiarmid's bound instead of Hoeffding bound in VFDT is more appropriate for ensuring the approximation bound i.e. the split decisions made after seeing certain number of instances will, with high probability, remain the same for infinite number of instances. The authors also presented McDiarmid's bound for information gain of ID3 algorithm, and Gini index of CART algorithm. It is to be noted that Hoeffding bound is a special case of McDiarmid's bound. In another paper, [Matuszyk et al., 2013] showed usage of Hoeffding bound is mathematically flawed as (i) Hoeffding inequality only applies to arithmetic average, which information gain and Gini index are not, (ii) values obtained in sliding window methods are not independent, while Hoeffding inequality only applies to independent random variables. A revised bound showed that decision bound should be twice of the one given by Hoeffding bound, otherwise, error-likelihood should be updated accordingly.

A different decision tree based approach based on so-called *Peano Count Tree (P-tree)* has been developed by Ding et al. for spatial data streams [Ding et al., 2002]. The Peano Count Tree is a spatial data structure that facilitates a lossless compressed representation of spatial data. This structure is used for fast calculation of information gain for branching in decision trees.

Aggarwal et al. employs a slightly different idea in handling time-evolving data in their on demand classification approach of data streams [Aggarwal et al., 2004]. They used a modified *micro-clusters concept* introduced in [Aggarwal et al., 2003]. Micro-clusters are created from the training data stream only. Each micro-cluster corresponds to a set of points from the training data belonging to the same class. To maintain statistics over different time horizons and avoid storage of unnecessary data points a geometric time frame is used. In the classification task, the *k-nearest neighbor* based approach is taken, where micro-clusters are treated as node weighted by their instance counts.

In [Ganti et al., 2002] two algorithms named GEMM and FOCUS have been introduced for streams under block evolution. GEMM is used for model maintenance and FOCUS is for change detection between two data stream models. These algorithms have been tested

using decision trees and frequent item set models. FOCUS uses bootstrapping methods to compute the distribution of deviation values when data characteristics remain the same. This distribution is then used to check whether the observed deviation value indicates a significant deviation. In another approach to handle concept drift, Last [Last, 2002] proposed an online classification system OLIN that would dynamically adjust the size of the training window and the number of new examples between model re-constructions to the current rate of concept drift. OLIN uses constant resources to produce models, and achieves nearly the same accuracy as the ones that would be produced by periodically re-constructing the model from all accumulated instances.

In a different approach to handle concept drift Castillo et al. [Castillo et al., 2003] used Shewhart P-Charts in an online framework based on the idea of *Statistical Quality Control*. Two alternatives of P-charts were used to monitor the stability of one or more quality characteristics in a drifting stream. The two alternatives only differed in the methods they estimate the target value to set the center. The group later introduced another drift detection scheme that monitors probability distribution of examples and maintains an online error rate to detect any concept drift [Gama et al., 2004a]. When distribution changes, error rate will increase. For stationary concept, the error rate should always gradually decrease. A new concept is said to be started if the error rate exceeds some predefined warning or threshold level. This approach has been used in *Ultra Fast Forest Tree (UFFT)* [Gama et al., 2004b, Gama et al., 2005] stream classification method. UFFT maintains naive Bayes statistics for every node. If at any node the error rate starts increasing, the node is pruned for drifting concept. UFFT uses similar approach and Hoeffding bound to control the growth of the tree. For each pair of classes a tree is maintained, hence it is called forest-of-trees.

Later, Aggarwal proposed another concept drift technique based on velocity density estimation [Aggarwal, 2003]. Velocity density estimation is a technique to understand, visualize, and determine trends in the evolving data. The work presented a scheme to use velocity density estimation to create temporal velocity profiles and spatial velocity profiles at periodic instants in time. These profiles are then used to predict dissolution, coagulation, and shift in data. Proposed method could detect changes in trends in a single scan with linear order of number of data points. Additionally, a batch processing techniques to identify combinations of dimensions which results the greatest amount of global evolution are also introduced. In [Kifer et al., 2004] authors tried to formally define and quantify the change so that existing algorithms can precisely specify when and how the underlying distribution has changed. They employed a two fixed-length window model, where a current one is updated every time a new example arrives and a reference one is only updated when a change has been detected. To compare the distributions of the windows $L1$ distance has been used. Another method to compare two distribution has been presented in [Dasu et al., 2004] where authors used Kullback-Leibler (KL) distance to compare two distributions. KL distance is known to be related to the optimal error in determining the similarity of two distributions. In this non-parametric method no assumptions on the underlying distributions is required.

Ensemble Learning

Online bagging and boosting method do not particularly give attention to the concept drifting nature in the data. *Accuracy Weighted Ensemble (AWE)* [Wang et al., 2003] is one of the earliest work on concept-drifting stream data. AWE assumes that the stream is delivered in chunks of defined size. With each incoming chunk, AWE updates its k classifiers. Each classifier is associated a weight which is inversely proportional to the expected error of the respective classifier. To estimate this error, it is assumed that the distribution of test set is closest to the most recent chunks. Concept drift is adapted by effectively manipulating the number and the magnitude of the weights that are changing. An extension of AWE has been proposed in [Brzezinski and Stefanowski, 2011], namely *Accuracy Updated Ensemble (AUE)*. AUE takes the weighting motivation from AWE, but improves the limitation of AWE. In AWE each classifier learns from the incoming chunks in a "batched" fashion. AUE employs an online scheme instead. AUE also adapts the weighting function to reduce the adverse effect in AWE of sudden drift in data. AWE weighting function is prone to suffer by rapid change in the stream and most or even all classifiers assuming they are "risky". This limitation has been addressed in AUE. Result shows that AUE performs marginally better than AWE, however, also requires slightly longer time and larger space.

Another group of methods known as *random forests* uses bagging of decision trees. The concept of random forest is introduced by Breiman [Breiman, 1999]. Random forest works in a similar manner as bootstrap aggregation. However, for each split in the tree it only selects a subset of the features to be considered for splitting criterion. This *feature bagging* approach helps avoiding very strong predictors to get selected over and over. One method mentioned before, *Ultra Fast Forest Tree (UFFT)* [Gama et al., 2004b, Gama et al., 2005] uses concepts of tree bagging, however, works with all features the whole time. UFFT maintains statistical information on each node to detect drift and grows the tree in a similar approach to VFDT. Using the statistical information stored in nodes it detects concept drift with naive Bayes error-rate. Abdulsalam et al. proposed *Dynamic Streaming Random Forests (DSRF)* focusing on lowering the number of examples required to build up new model in a drifting environment [Abdulsalam et al., 2008, Abdulsalam et al., 2011]. Shannon entropy [Shannon, 2001] is used to detect concept drift. All these methods are empirically proven to be effective for generalized scenario and chosen data sets.

Appendix C

Classical Learning Algorithms

Learning Algorithms

The goal of data classification process is to predict an outcome based on some give data. In order to do so, data mining algorithms first processes a training set containing a set of attributes and corresponding outcome: a class or a value. Algorithms develop hypotheses that best describe the relationship among the attributes and the outcome for the total instance set. Formally, learning problems are defined as follows: Given a data set of m instances $D \equiv \{\{\vec{x}_1, y_1\}, \{\vec{x}_2, y_2\}, \{\vec{x}_3, y_3\}, \ldots, \{\vec{x}_m, y_m\}\}$ where $\vec{x} = \{x_1, x_2, x_3, \ldots, x_n\}$, learning algorithms try to approximate $H \equiv y = f(\vec{x})$. Additional to being typical linear, polynomial, etc. functions, H can also be a set of IF-THEN rules. These rules or functions are then used to predict unseen instances where outcome class is not known. These algorithms are known as learning algorithms, and in last few decades, have widely been researched in the field of machine learning. This section briefly discusses few of the basic algorithms upon which the foundation of ensemble methods and this thesis are laid. The detailed discussion of these algorithms are out of the scope of this chapter. Thus, we discuss only the core concepts here.

k-Nearest Neighbors

k-nearest neighbor algorithm, in short k-NN, is one the simplest learning available. It is a non-parametric instance-based lazy learning algorithm and works for both supervised and unsupervised learning. It uses similarity measure like distance functions to classify unknown instances. The decision is a majority voting of k neighbors where k is predefined. For example, if $k = 1$ then an unknown instance is assigned the class label of its nearest neighbor. There are several schemes to find the k nearest neighbors. For continuous data, some well-known distance functions are Euclidean distance, Manhattan distance, Minkowski distance, etc. As in Figure C.1 1-NN classification for the new instance is –, 2-NN is undefined, 3-NN is + using Euclidean distance.

- Euclidean distance $D_E = \sqrt{\sum_{ß=1}^{k}(x_i - y_i)^2}$
- Manhattan distance $D_M = \sum_{ß=1}^{k} |(x_i - y_i)|$

- Minkowski distance $D_{Mink} \sqrt[q]{\sum_{ß=1}^{k}(|x_i - y_i|)^q}$

For categorical data, Hamming distance can be used. Hamming distance is defined as $D_H = \sum_{ß=1}^{k} |(x_i - y_i)|$ where $|x_i - y_i|$ is 0 if $x = y$, 1 otherwise.

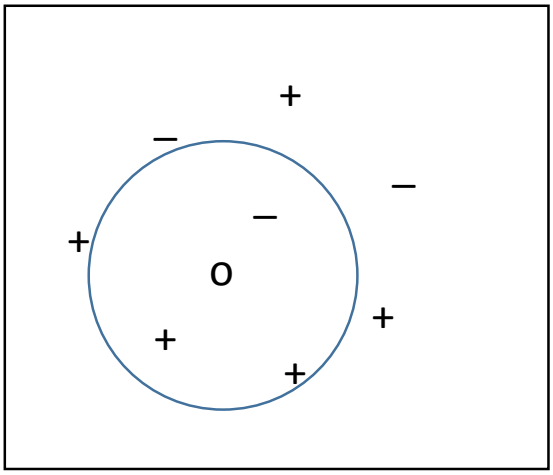


Figure C.1: Concept of k-NN

Choosing an optimal k is the prime challenge of k-NN algorithm, which is extremely training data dependent. Changing position of few training points could lead to a significant loss in performance. The method is particularly not stable in the class boundary. k should be large enough that k-NN could overcome the noises in data, and small enough that instances of other classes are not included. Generally, higher k reduces the overall noise and should be more precise. For most data set a value between 3-10 performs much better than 1-NN. Typically, cross-validation is used to determine a good k.

Naïve Bayes

Naive Bayes classification is based on Bayes rule. Bayes rule says that the probability of event x conditioned on knowing event y, i.e. the probability of x given y is defined as

$$p(x|y) = \frac{p(x,y)}{p(y)} = \frac{p(y|x)p(x)}{p(y)}$$

The naive Bayes classifier [Langley et al., 1992] assumes that all the explanatory variables are independent. Given a feature set $X = x_1, x_2, \ldots, x_n$ of n independent variables, naive Bayes assigns the instances to k possible outcome or classes, $p(C_k|X)$. Using Bayes rule, the conditional probability can be decomposed as:

$$p(C_k|X) = \frac{p(C_k)p(X|C_k)}{p(X)}.$$

In other words,

$$posterior = \frac{prior \times likelihood}{evidence}.$$

Using chain rule repetitively, it can be expressed as follows:

$$p(C_k|X) = p(C_k) \prod_{i=1}^{n} p(x_i|C_k)$$

The predicted class is the one which maximizes the conditional probabilities $p(C_k|X)$, that is, classifier assigns the class label $\hat{y} = C_k$ for some k that satisfies:

$$\hat{y} = \underset{k \in \{1,\ldots,K\}}{argmax}\, p(C_k) \prod_{i=1}^{n} p(x_i|C_k)$$

Decision Tree

Decision trees (DTs) are another popular genre of non-parametric supervised learning algorithms. Trees allow a way to graphically organize a sequential decision process. Decision tree, a directed acyclic graph, contains decision nodes, each with branches for all alternate decisions. Leaf nodes are labeled with respective class label. Each path from root to leaf is a decision rule, consisting of conditional part (unions of all internal nodes' conditions) and a decision (of leaf node).

Decision trees use *information gain* to select the splitting attribute that would maximize the total entropy of each of the subtrees resulting from the spit. Entropy is a measurement of purity of an attribute, typically ranged between 0 and 1. A pure attribute, attribute with definitive value-class relationship, would have a low entropy and is easy to predict. On the other hand attribute with highly mixed value-class relationship would yield high entropy. A common entropy function is-

$$entropy = -\sum_{i=1}^{n} p_i \lg p_i.$$

where $p_1, p_2, \ldots, p_n$ are the distributions of several class attributes and $\sum p_i = 1$. Information gain is the difference between old and new entropy after a split. This is good measurement of relevance of an attribute. However, in cases where an attribute can take on a large number of distinct values, information gain is not ideal. Presence of large set of values could uniquely identify the classes but also reduces chance to generalize unseen instance and should be avoided to be place near root.

Decision tree is, however, computationally feasible. Average cost of constructing a decision tree for n instances with m attributes is $O(mn \lg n)$, and querying time is $O(\lg n)$.

Neural Network

The Neural Network (NN) is a learning method inspired by biological neural network. A set of inter-connected nodes (also known as perceptrons and neurons) mimics biological neural network. Figure C.2 shows a skeleton of a neural network. Neural network is a weighted directed graph where an input layer is connected to the output layer via some layers of hidden layers. This is commonly known as Multi-Layer Perceptron (MLP). Decisions are made looking only into the output layer only. Hidden layers enlarge the space of hypotheses.

If there is no hidden layer, then neural network becomes a simple regression problem i.e. output is a linear function of inputs. Each connection among these nodes has an associated weight. Given an input data set, these weights are updated accordingly to best fit the classification model. Learning is done by back-propagation algorithm [Rojas, 1996]. Errors are propagated back from the output layer to the hidden layer and weights are updated to minimize error.

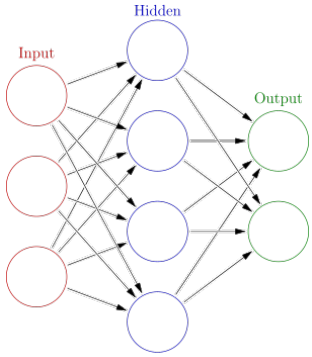


Figure C.2: Neural network layers

Learning starts with assigning a small value as initial weights to all the connection weights. In case of a single perceptron, learning process is analogous to moving a parametric hyperplane around. Let w_i be the weight of i-th input, then after $t+1$ iteration weight $w_i(t+1) = w_i(t) + \nabla E_i(t)$, where ∇E_i is the gradient of the error function. For an input vector $\mathbf{x}$ and true output y, E is defined as the squared error:

$$E = \frac{1}{2}Err^2 = \frac{1}{2}(y - h_w(\mathbf{x}))^2$$

Thus gradient of E would be:

$$\begin{aligned}\nabla E = \frac{\partial E}{\partial W_j} &= Err \times \frac{\partial Err}{\partial W_j} = Err \times \frac{\partial}{\partial W_j}(y - g(\sum_{j=0}^{n} w_j x_j)) \\ &= -Err \times g'(in) \times x_j\end{aligned}$$

Based on this, weight update rule would be:

$$w_j = w_j + \alpha \times Err \times g'(in) \times x_j$$

where α is learning rate coefficient. Back propagation algorithm for multi-layer perceptron also works in similar fashion. Weights in the output layer are updated using following equation:

$$w_{j,i} = w_{j,i} + \alpha \times a_j \times \nabla_i$$

where $\nabla_i = Err \times g'(in_i)$. Hidden layers propagates this error from the output layer using:

$$\nabla_j = g'(in_i) \sum_i w_{j,i} \nabla_i$$

$$w_{k,j} = w_{k,j} + \alpha \times a_k \times \nabla_j$$

MLP is a very expressive method. Only 1 hidden layer is sufficient to represent all continuous functions and 2 can represent any functions. MLP is prone to local minima, which is avoided by running MLP multiple times with different initial weight settings.

Appendix D

Census Income Dataset

Census income data set, also commonly known as adult data set has 48842 instances consisting of 14 attributes that map to a binomial Income class (<=50K or >50K). Attributes are age, sex, education, occupation, work class, working hours, marital status, relationship, country, race, final weight, capital gain, and capital loss. 2809 instance have missing values for either occupation or work class or both.

Occupation, education, marital status are the most important attributes. They can sufficiently classify the instance with an accuracy within 1% of the accuracy achieved using all attributes. Education attribute has 16 values from pre-school to doctorate. 66% of the instances belong to high-school grad, some college, and bachelor. Occupation attribute has 15 values. Instances are more balanced for this attribute. Marital status consists of values from married, never married, and other status. The target income class is skewed towards <=50K class with 76% of the instance.

Five primary clusters can be found in this dataset. Cluster 1 being mostly student groups, still studying and earning <= 50K. Cluster 2 are the young professionals, just started their career, and mostly earning less than 50K. Cluster 3 are the people with white collar jobs. People with bachelor or more education here earn >50K. Cluster 4 are the people with blue collar jobs, mostly earning <=50K. Cluster 5 are females. In this group too, people with bachelor or more degree earn >50K.

Most batched leaning algorithm achieves about 80-83% accuracy with about 85-88% F1 value.

Appendix E

Implementation Details

This appendix contains the general information about the implementation details. The implementation is primarily based on MOA implementation in Java.

Packages, Sources, and Libraries

```
Packages:
src.test
src.wrapper
src.wrapper.generators

Sources:
/src/test/MainRandRBF.java                // contains tests for random RBF generator
/src/test/MainVSRBF.java                  // contains tests for VSRBF generator
/src/test/MainCensus.java                 // contains tests for census income data set
/src/test/RunTest.java                    // provides test functionality
/src/test/TestParameters.java             // universal test parameters

/src/wrapper/My*.java                     // wrapper for MOA implementation of * class
/src/wrapper/SizeRestrictedHT.java        // new Hoeffding tree variant
/src/wrapper/CoBagSRHT.java               // new ensemble method introduced in this thesis

/src/wrapper/generator/InstancePool.java                // instance information within pools
/src/wrapper/generator/My*.java                         // wrapper for MOA implementation
/src/wrapper/generator/VarSpeedRBFGenerator.java        // new generator

Libraries:
weka
moa
```

Data Generation

Three different data sets are being used for the experimentations. Firstly, for census income data set original "arff" data file is read using MOA's ArffFileStream.

Code Snippet E.1: Arff file reader

```
1  public class ArffFileStream extends AbstractOptionHandler
2          implements InstanceStream {
3      ...
4      /*
5       * @param arffFileName file to read
6       * @param classIndex index of the class attribute, 0 for none, -1 for last
7       */
8      public ArffFileStream(String arffFileName, int classIndex) {
9          this.arffFileOption.setValue(arffFileName);
10         this.classIndexOption.setValue(classIndex);
11         restart();
12     }
13     ...
14 }
```

Other generators are placed in src.wrapper.generators package. To have the flexibility of experimenting with various options available, original MOA implementation of RandomRBFGeneratorDrift is copied into a wrapper class (MyRandomRBFGeneratorDrift). In this class, required number of centroids are first generated at random with a user given values as the seed.

Code Snippet E.2: Generating initial centroids

```
public class MyRandomRBFGenerator extends AbstractOptionHandler
        implements InstanceStream {
    protected void generateCentroids() {
        Random modelRand = new Random(this.modelRandomSeedOption.getValue());
        this.centroids = new Centroid[this.numCentroidsOption.getValue()];
        this.centroidWeights = new double[this.centroids.length];
        for (int i = 0; i < this.centroids.length; i++) {
            this.centroids[i] = new Centroid();
            double[] randCentre = new double[this.numAttsOption.getValue()];
            for (int j = 0; j < randCentre.length; j++) {
                randCentre[j] = modelRand.nextDouble();
            }
            this.centroids[i].centre = randCentre;
            this.centroids[i].classLabel = modelRand.nextInt(this.numClassesOption.getValue());
            this.centroids[i].stdDev = modelRand.nextDouble();
            this.centroidWeights[i] = modelRand.nextDouble();
        }
    }
}
```

Then, a random drift amount is associated with each centroid.

Code Snippet E.3: Associating drift with centroids

```
public class MyRandomRBFGeneratorDrift extends MyRandomRBFGenerator {
    protected void generateCentroids() {
        super.generateCentroids();              // first generate all centroids
        Random modelRand = new Random(this.modelRandomSeedOption.getValue());
        int len = this.numDriftCentroidsOption.getValue();
        if (len > this.centroids.length) {
            len = this.centroids.length;
        }
        this.speedCentroids = new double[len][this.numAttsOption.getValue()];
        for (int i = 0; i < len; i++) {
            double[] randSpeed = new double[this.numAttsOption.getValue()];
            double normSpeed = 0.0;
            for (int j = 0; j < randSpeed.length; j++) {
                randSpeed[j] = modelRand.nextDouble();
                normSpeed += randSpeed[j] * randSpeed[j];
            }
            normSpeed = Math.sqrt(normSpeed);
            for (int j = 0; j < randSpeed.length; j++) {
                randSpeed[j] /= normSpeed;
            }
            this.speedCentroids[i] = randSpeed;
        }
    }
}
```

This drift amount is used to update the centroids' center every time a new instance is requested.

Code Snippet E.4: Update centroid locations before getting a new instance

```
public class MyRandomRBFGeneratorDrift extends MyRandomRBFGenerator {
    public Instance nextInstance() {
        int len = this.numDriftCentroidsOption.getValue();
        if (len > this.centroids.length) {
            len = this.centroids.length;
        }
        for (int j = 0; j < len; j++) {
            for (int i = 0; i < this.numAttsOption.getValue(); i++) {
                this.centroids[j].centre[i] += this.speedCentroids[j][i] *
                    this.speedChangeOption.getValue();
                if (this.centroids[j].centre[i] > 1) {
                    this.centroids[j].centre[i] = 1;
```

```
                    this.speedCentroids[j][i] = -this.speedCentroids[j][i];
                }
                if (this.centroids[j].centre[i] < 0) {
                    this.centroids[j].centre[i] = 0;
                    this.speedCentroids[j][i] = -this.speedCentroids[j][i];
                }
            }
        }
        return super.nextInstance();
    }
}
```

Finally, the instance is generated using specified values as parameters of normal distribution.

Code Snippet E.5: Get new instance from the generator

```
public class MyRandomRBFGenerator extends AbstractOptionHandler
        implements InstanceStream {
    public Instance nextInstance() {
        _curCentroidIndex = MiscUtils.chooseRandomIndexBasedOnWeights(this.centroidWeights,
        this.instanceRandom);
        Centroid centroid = this.centroids[_curCentroidIndex];
        int numAtts = this.numAttsOption.getValue();
        double[] attVals = new double[numAtts + 1];
        for (int i = 0; i < numAtts; i++) {
            attVals[i] = (this.instanceRandom.nextDouble() * 2.0) - 1.0;
        }
        double magnitude = 0.0;
        for (int i = 0; i < numAtts; i++) {
            magnitude += attVals[i] * attVals[i];
        }
        magnitude = Math.sqrt(magnitude);
        double desiredMag = this.instanceRandom.nextGaussian() * centroid.stdDev;
        double scale = desiredMag / magnitude;
        for (int i = 0; i < numAtts; i++) {
            attVals[i] = centroid.centre[i] + attVals[i] * scale;
        }
        Instance inst = new DenseInstance(1.0, attVals);
        inst.setDataset(getHeader());
        inst.setClassValue(centroid.classLabel);
        return inst;
    }
}
```

Variable speed RBF generator is implemented into VarSpeedRBFGenerator class. Centroids are generated using the same methods as random RBF generator. Then they are assigned to selected number of pools. A configuration step then assigns desired drift speed and activation percentage to the centroids among pools.

Code Snippet E.6: Assign centroids to pools for variable speed RBF generator

```
public class VarSpeedRBFGeneratorDrift  extends AbstractOptionHandler
        implements InstanceStream {
    protected void generateCentroids() {
        ...             // follow generation process of random RBF generator
        Random poolrand = new Random((int) System.currentTimeMillis());
        for (int i = 0; i < centroids.length; i++) {
            pools[poolrand.nextInt(pools.length)].centroidIndices.add(i);
        }
        reconfig();
    }
}
```

For sake of simplicity, instances are produced in batches. Within each batch they maintain their designated weights. Before each batch is generated, a reconfiguration step is performed. This step rearranges active centroids and their drift coefficients. For slower streams most of the centroids remain active, while for faster streams, chances of new centroids to be activated are very high.

Code Snippet E.7: Reconfiguration step for new batch

```
public class VarSpeedRBFGeneratorDrift  extends AbstractOptionHandler
        implements InstanceStream {
    public void reconfig() {
        for (int i = 0; i < centroids.length; i++) {
            centroids[i].isActive = true;
        }
        for (int i = 0; i < pools.length; i++) {
            Random poolrand = new Random((int) System.currentTimeMillis());
            pools[i].activationPercent = 1.0- i * 0.2;
            int active =(int) (pools[i].centroidIndices.size() * pools[i].activationPercent);
            int toDeactivate = pools[i].centroidIndices.size() - active;

            for (int j = 0, k = 0; k < toDeactivate; j++) {
                int index = pools[i].centroidIndices.get(j % pools[i].centroidIndices.size());
                if (poolrand.nextInt(100) % 2 == 0 && centroids[index].isActive != false) {
                    centroids[index].isActive = false;
                    k++;
                }
                centroids[index].driftCoeffient = this.speedChangeOption.getValue()
                    * (1.0 - 1.0/(i+1));
            }
        }
    }
}
```

Algorithm Implementations

Algorithms are placed in src.wrapper package. All the algorithms to be analyzed are wrapped or extended to facilitate capturing of various statistics during the learning process. Implementation of new algorithms introduced in the thesis are also placed in this package. Size restricted Hoeffding tree is implemented in SizeRestrictedHT class and carry-over bagging is placed in CoBagSRHT class.

SizeRestrictedHT is extended from MyHoeffdingAdaptiveTree, the ADWIN variant of Hoeffding tree. Learning process of SizeRestrictedHT is similar to MyASHoeffdingTree except for direct resetting.

Code Snippet E.8: Training method of Size Restricted Hoeffding Tree

```
public class SizeRestrictedHT extends MyHoeffdingAdaptiveTree {
    public void trainOnInstanceImpl(Instance inst) {
        ... // if root is null, create new node
        MyHoeffdingTree.FoundNode foundNode = this.treeRoot.filterInstanceToLeaf(inst, null,
            -1);
        MyHoeffdingTree.Node leafNode = foundNode.node;
        ... // if leaf is null, create new leaf
        if (leafNode instanceof MyHoeffdingTree.LearningNode) {
            MyHoeffdingTree.LearningNode learningNode = (MyHoeffdingTree.LearningNode) leafNode;
            learningNode.learnFromInstance(inst, this);
            if (this.growthAllowed && (learningNode instanceof
                MyHoeffdingTree.ActiveLearningNode)) {
                MyHoeffdingTree.ActiveLearningNode activeLearningNode =
                    (MyHoeffdingTree.ActiveLearningNode) learningNode;
                double weightSeen = activeLearningNode.getWeightSeen();
                if (weightSeen - activeLearningNode.getWeightSeenAtLastSplitEvaluation() >=
                    this.gracePeriodOption.getValue()) {
                    int currentDNcount = this.decisionNodeCount;
                    if (this.decisionNodeCount < this.maxSize) {
                        attemptToSplit(activeLearningNode, foundNode.parent,
                            foundNode.parentBranch);
                    } else {
                        ... // try to learn from isntance if tree size is not increased
                    }
                    if (numSizeLimitCrossed >= shouldResetAfter) {
                        ... // reset or prune
                    }
                    activeLearningNode.setWeightSeenAtLastSplitEvaluation(weightSeen);
                }
            }
```

```
26          }
27      }
28  }
```

This training method of SizeRestrictedHT is eventually called by the training method of CoBagSRHT. CobagSRHT is derived from MyOzabag, the very primitive implementation of Oza online bagging method. Both training and voting methods are shown in the fowling code snippet.

Code Snippet E.9: Learning and voting with CoBagSRHT

```
1   public class CoBagSRHT extends MyOzaBag {
2       public void trainOnInstanceImpl(Instance inst) {
3           int trueClass = (int) inst.classValue();
4           for (Classifier cl : extraClassifiers) {
5               SizeRestrictedHT cx = (SizeRestrictedHT) cl;
6               cx.trainOnInstance(inst);
7               if (cx.getToleranceRemain() == 0)
8               extraClassifiers.remove(cl);
9           }
10          for (int i = 0; i < this.ensemble.length; i++) {
11              int k = MiscUtils.poisson(1.0, this.classifierRandom);
12              if (k > 0) {
13                  Instance weightedInst = (Instance) inst.copy();
14                  ... // update error estimation
15                  this.ensemble[i].trainOnInstance(weightedInst);
16
17                  SizeRestrictedHT ens = (SizeRestrictedHT) ensemble[i];
18                  if (ens.getDecisionNodeCount() >= ens.maxSize) {
19                      if (i > this.ensembleSizeOption.getValue()/2
20                      && extraClassifiers.size() <= this.ensembleSizeOption.getValue() )
21                      extraClassifiers.add(ens);
22                      if (extraClassifiers.size() > this.ensembleSizeOption.getValue()/2) {
23                          extraClassifiers.remove(0);
24                      }
25                      ... // create a new classifier with capasity of ens.maxSize
26                  }
27              }
28          }
29      }
30
31      public double[] getVotesForInstance(Instance inst) {
32          DoubleVector combinedVote = new DoubleVector();
33          for (Classifier cl : extraClassifiers) {
34              ... // get votes
35          }
36          for (int i = 0; i < this.ensemble.length; i++) {
37              ... // get votes
38          }
39          return combinedVote.getArrayRef();
40      }
41  }
```

Run Experiments

For testing, a generalized class with classifier, stream, and output buffer as parameters is implemented (class RunTest). It initializes the basic configuration for passed parameters, and performs test-then-train method for predefined (in TestParameters class) number of instances.

Code Snippet E.10: Universal test function

```
1   public class RunTest {
2       public void test(InstanceStream inStream, Classifier inclassifier, StringBuilder outstats)
3       throws Exception {
4           ... // intialization
5           while (trainingStream.hasMoreInstances()
6           && numberInstance++ < TestParameters.NUMBER_OF_INSTANCES) {
7
```

```
8               double[] votes = classifier.getVotesForInstance(trainInst);
9               windowEval.addResult(trainInst, votes);
10              basicEval.addResult(trainInst, votes);
11              ... // update stats
12
13              classifier.trainOnInstance(trainInst);
14          }
15          ... // record stats
16      }
17  }
```

The usage of test function is simple. Output buffer, stream, and classifier are instantiated; and set with required parameter values. Then, these are passed to the test function. The function takes the responsibility of preparing the stream and classifier for use, and result is returned into the output buffer.

Code Snippet E.11: Using the test function

```
1   public String test_OzaBagASHT_RandRBF () {
2       StringBuilder stats = new StringBuilder();  // stats recorder
3       RandomRBFGeneratorDrift inStream = new RandomRBFGeneratorDrift();
4       ... // set stream properties
5       inStream.prepareForUse();
6       MyOzaBagASHT classifier = new MyOzaBagASHT();
7       MyASHoeffdingTree base = new MyASHoeffdingTree();
8       ... // set base classifier properties
9       ... // set ensemble classifier properties
10
11      try {
12          new RunTest().test(inStream, classifier, stats);
13      } catch (Exception ex) {
14          ...     // handle exception
15      }
16      return stats.toString();
17  }
```