Department of Computer Science

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Hamilton, New Zealand

Improving Hoeffding Trees

by

Richard Kirkby

A thesis submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

at the

University of Waikato

in the subject of

Computer Science

November 2007

Abstract

Modern information technology allows information to be collected at a far greater rate than ever before. So fast, in fact, that the main problem is making sense of it all. *Machine learning* offers promise of a solution, but the field mainly focusses on achieving high accuracy when data supply is limited. While this has created sophisticated classification algorithms, many do not cope with increasing data set sizes. When the data set sizes get to a point where they could be considered to represent a *continuous* supply, or *data stream*, then incremental classification algorithms are required. In this setting, the effectiveness of an algorithm cannot simply be assessed by accuracy alone. Consideration needs to be given to the memory available to the algorithm and the speed at which data is processed in terms of both the time taken to predict the class of a new data sample and the time taken to include this sample in an incrementally updated classification model.

The *Hoeffding tree* algorithm is a state-of-the-art method for inducing *decision trees* from data streams. The aim of this thesis is to improve this algorithm. To measure improvement, a comprehensive framework for evaluating the performance of data stream algorithms is developed. Within the framework memory size is fixed in order to simulate realistic application scenarios. In order to simulate continuous operation, classes of synthetic data are generated providing an evaluation on a large scale. Improvements to many aspects of the Hoeffding tree algorithm are demonstrated. First, a number of methods for handling continuous numeric features are compared. Second, tree prediction strategy is investigated to evaluate the utility of various methods. Finally, the possibility of improving accuracy using ensemble methods is explored.

The experimental results provide meaningful comparisons of accuracy and processing speeds between different modifications of the Hoeffding tree algorithm under various memory limits. The study on numeric attributes demonstrates that sacrificing accuracy for space at the local level often results in improved global accuracy. The prediction strategy shown to perform best adaptively chooses between standard majority class and Naive Bayes prediction in the leaves. The ensemble method investigation shows that combining trees can be worthwhile, but only when sufficient memory is available, and improvement is less likely than in traditional machine learning. In particular, issues are encountered when applying the popular *boosting* method to streams.

Acknowledgments

This thesis could not exist without the monumental support of my co-supervisors, Geoff Holmes and Bernhard Pfahringer.

Geoff is a great leader, providing the environment and opportunity, including generous financial support. He has patiently endured some of my less acceptable attempts at technical writing, always pushing for greater quality. For this I am incredibly grateful, as the thesis is so much better for it. It was his masterful vision and overall wisdom that has secured my success.

Bernhard has always offered an open office door for me to drop by any time and discuss the latest conundrum, happy to put aside other demands on his time. His keen insight and probing questions always managed to crystallize my thoughts and draw me closer to a solution. His exceptional problem solving skills have meant that no problem was too great.

They have both been a pleasure to work with, a supervision team I really could not have asked more from. I will forever remain in their debt.

I would also like to thank the Machine Learning group, my colleagues Eibe Frank, Mark Hall, Remco Bouckaert, Gabi Schmidberger, Grant Anderson, and the countless students who have come and gone over the years. They have all helped to make the Machine Learning lab an interesting and fun environment to work in.

My final but no less deserving thanks are reserved for my support system outside university. Thanks go to my family and friends, who may not always understand what it is that I do, but who are encouraging just the same. My parents especially for raising and supporting me in ways I can never repay. Their encouragement and endorsement of my expensive childhood computer hobby has led me to this point.

My wife and two lovely boys are the ultimate drive that helped me follow this sometimes difficult journey to completion. I thank them so much for making my life the pleasure that it is.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

A largely untested hypothesis of modern society is that it is important to record data as it may contain valuable information. This occurs in almost all facets of life from supermarket checkouts to the movements of cows in a paddock. To support the hypothesis, engineers and scientists have produced a raft of ingenious schemes and devices from loyalty programs to RFID tags. Little thought however, has gone into how this *quantity* of data might be analyzed.

Machine learning, the field for finding ways to automatically extract information from data, was once considered the solution to this problem. Historically it has concentrated on learning from small numbers of examples, because only limited amounts of data were available when the field emerged. Some very sophisticated algorithms have resulted from the research that can learn highly accurate models from limited training examples. It is commonly assumed that the entire set of training data can be stored in working memory.

More recently the need to process larger amounts of data has motivated the field of *data mining*. Ways are investigated to reduce the computation time and memory needed to process large but static data sets. If the data cannot fit into memory, it may be necessary to sample a smaller training set. Alternatively, algorithms may resort to temporary external storage, or only process subsets of data at a time. Commonly the goal is to create a learning process that is linear in the number of examples. The essential learning procedure is treated like a scaled up version of classic machine learning, where learning is considered a single, possibly expensive, operation—a set of training examples are processed to output a final static model.

The data mining approach may allow larger data sets to be handled, but it still does not address the problem of a *continuous* supply of data. Typically, a model that was previously induced cannot be updated when new information arrives. Instead, the entire training process must be repeated with the new examples included. There are situations where this limitation is undesirable and is likely to be inefficient.

The *data stream* paradigm has recently emerged in response to the continuous data problem. Algorithms written for data streams can naturally cope with data sizes many times greater than memory, and can extend to challenging real-time applications not previously tackled by machine learning or data mining. The core assumption of data stream processing is that training examples can be briefly inspected a single time only, that is, they arrive in a high speed stream, then must be discarded to make room for subsequent examples. The algorithm processing the stream has no control over the order of the examples seen, and must update its model incrementally as each example is inspected. An additional desirable property, the so-called *anytime* property, requires that the model is ready to be applied at any point between training examples.

The *Hoeffding tree* induction method, a method for producing *decision tree* models, represents one of the best known algorithms for classifying streams of examples. Improvements to it are the focus of this thesis. As the method is already state-of-the-art, it is not expected that massive gains will be possible, but rather smaller incremental improvements that are beneficial nonetheless. To measure improvements to this algorithm, an evaluation framework is developed to provide useful insight about classification performance. This fosters algorithm development that results in measurably improved performance.

Studying purely theoretical advantages of algorithms is certainly useful and enables new developments, but the demands of data streams require this to be followed up with empirical evidence of performance. Claiming that an algorithm is suitable for data stream scenarios implies that it possesses the necessary practical capabilities. Doubts remain if these claims cannot be backed by reasonable empirical evidence.

A central argument of this thesis is that data stream classification algorithms require appropriate and complete evaluation practices. The evaluation should allow users to be sure that particular problems can be handled, to quantify improvements to algorithms, and to determine which algorithms are most suitable for their problem. The framework is suggested with these needs in mind.

Measuring data stream classification performance is a three dimensional problem involving processing speed, memory and accuracy. It is not possible to enforce and simultaneously measure all three at the same time so this thesis argues that it is necessary to fix the memory size and then record the other two. Various memory sizes can be associated with data stream application scenarios so that basic questions can be asked about expected performance of algorithms in a given application scenario.

1.1 Assumptions

This thesis is concerned with the problem of *classification*, perhaps the most commonly researched machine learning task. The goal of classification is to produce a model that can predict the class of unlabeled examples, by training on examples whose label, or *class*, is supplied. To clarify the problem setting being addressed, several assumptions are made about the typical learning scenario:

1. The data is assumed to have a small and fixed number of columns, or *attributes/features*—several hundred at the most.

2. The number of rows, or *examples*, is very large—millions of examples at the smaller scale. In fact, algorithms should have the potential to process an infinite amount of data, meaning that they will not exceed memory limits or otherwise fail no matter how many training examples are processed.

3. The data has a limited number of possible class labels, typically less than ten.

4. The amount of memory available to a learning algorithm depends on the application. The size of the training data will be considerably larger than the available memory.

5. There should be a small upper bound on the time allowed to train or classify an example. This permits algorithms to scale linearly with the number of examples, so users can process N times more than an existing amount simply by waiting N times longer than they already have.

6. Stream concepts are assumed to be stationary, that is, the problem of *concept drift* is not directly addressed. Concept drift occurs when the underlying concept defining the target being learned begins to shift over

time. The solutions explored here could be extended to handle concept drift, but this is reserved for future work.

The first three points emphasize that the aim is to scale with the number of examples. Data sources that are large in other dimensions, such as numbers of attributes or possible labels are not the intended problem domain. Points 4 and 5 outline what is needed from a solution. Regarding point 6, some researchers argue that addressing concept drift is one of the most crucial issues in processing data streams. For this thesis it is believed more important that the other requirements are met first, otherwise a solution will not be satisfactory when demands are too high regardless of whether concept drift is addressed.

1.2 Requirements

The conventional machine learning setting, referred to in this thesis as the *batch* setting, operates assuming that the training data is available as a whole set—any example can be retrieved as needed for little cost. An alternative is to treat the training data as a *stream*, a potentially endless flow of data that arrives in an order that cannot be controlled. Note that an algorithm capable of learning from a stream is, by definition, a data mining algorithm.

Placing classification in a data stream setting offers several advantages. Not only is the limiting assumption of early machine learning techniques addressed, but other applications even more demanding than mining of large databases can be attempted. An example of such an application is the monitoring of high-speed network traffic, where the unending flow of data is too overwhelming to consider storing and revisiting.

A classification algorithm must meet several requirements in order to work with the assumptions and be suitable for learning from data streams. The requirements, numbered 1 through 4, are detailed below.

Requirement 1: Process an example at a time, and inspect it only once (at most)

The key characteristic of a data stream is that data 'flows' by one example after another. There is no allowance for *random access* of the data being supplied. Each example must be accepted as it arrives in the order that it arrives. Once inspected or ignored, an example is discarded with no ability to retrieve it again.

Although this requirement exists on the input to an algorithm, there is no rule preventing an algorithm from remembering examples internally in the short term. An example of this may be the algorithm storing up a *batch* of examples for use by a conventional learning scheme. While the algorithm is free to operate in this manner, it will have to discard stored examples at some point if it is to adhere to requirement 2.

The inspect-once rule may only be relaxed in cases where it is practical to re-send the entire stream, equivalent to multiple scans over a database. In this case an algorithm may be given a chance during subsequent passes to refine the model it has learned. However, an algorithm that requires any more than a single pass to operate is not flexible enough for universal applicability to data streams.

Requirement 2: Use a limited amount of memory

The main motivation for employing the data stream model is that it allows processing of data that is many times larger than available working memory. The danger with processing such large amounts of data is that memory is easily exhausted if there is no intentional limit set on its use.

Memory used by an algorithm can be divided into two categories: memory used to store running statistics, and memory used to store the current model. For the most memory-efficient algorithm they will be one and the same, that is, the running statistics directly constitute the model used for prediction.

This memory restriction is a physical restriction that can only be relaxed if external storage is used, temporary files for example. Any such work-around needs to be done with consideration of requirement 3.

Requirement 3: Work in a limited amount of time

For an algorithm to scale comfortably to any number of examples, its runtime complexity must be linear in the number of examples. This can be achieved in the data stream setting if there is a constant, preferably small, upper bound on the amount of processing per example.

Furthermore, if an algorithm is to be capable of working in *real-time*, it must process the examples as fast if not faster than they arrive. Failure to do so inevitably means loss of data.

Absolute timing is not as critical in less demanding applications, such as when the algorithm is being used to classify a large but persistent data source.

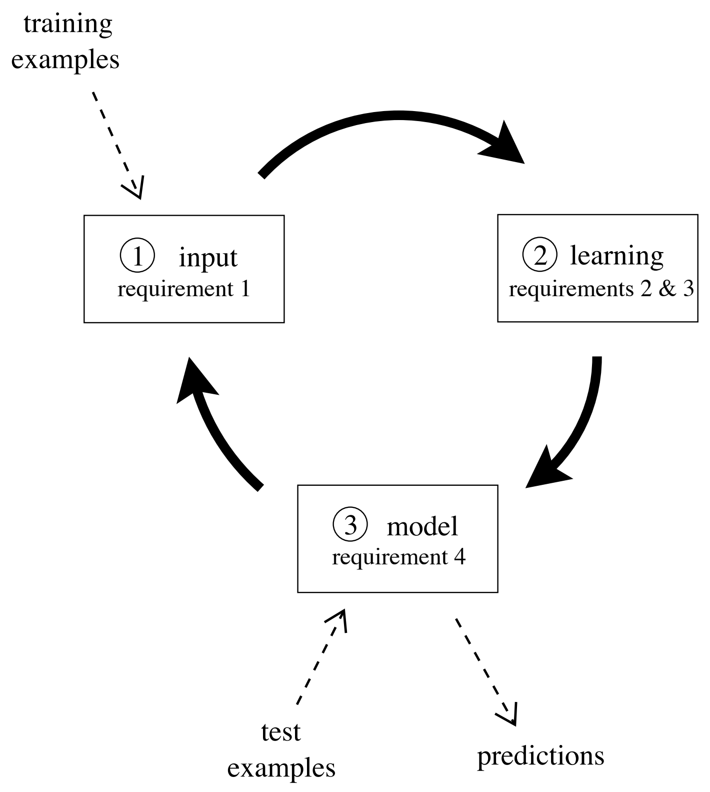

Figure 1.1: The data stream classification cycle.

However, the slower the algorithm is, the less value it will be for users who require results within a reasonable amount of time.

Requirement 4: Be ready to predict at any point

An ideal algorithm should be capable of producing the best model it can from the data it has observed after seeing any number of examples. In practice it is likely that there will be periods where the model stays constant, such as when a batch based algorithm is storing up the next batch.

The process of generating the model should be as efficient as possible, the best case being that no translation is necessary. That is, the final model is directly manipulated in memory by the algorithm as it processes examples, rather than having to recompute the model based on running statistics.

The data stream classification cycle

Figure 1.1 illustrates the typical use of a data stream classification algorithm, and how the requirements fit in. The general model of data stream classification follows these three steps in a repeating cycle:

1. The algorithm is passed the next available example from the stream (requirement 1).

2. The algorithm processes the example, updating its data structures. It does so without exceeding the memory bounds set on it (requirement 2), and as quickly as possible (requirement 3).

3. The algorithm is ready to accept the next example. On request it is able to supply a model that can be used to predict the class of unseen examples (requirement 4).

1.3 Strategies

The task of modifying machine learning algorithms to handle large data sets is known as *scaling up* [27]. Analogous to approaches used in data mining, there are two general strategies for taking machine learning concepts and applying them to data streams. The *wrapper* approach aims at maximum reuse of existing schemes, whereas *adaptation* looks for new methods tailored to the data stream setting.

Using a wrapper approach means that examples must in some way be collected into a batch so that a traditional batch learner can be used to induce a model. The models must then be chosen and combined in some way to form predictions. The difficulties of this approach include determining appropriate training set sizes, and also that training times will be out of the control of a wrapper algorithm, other than the indirect influence of adjusting the training set size. When wrapping around complex batch learners, training sets that are too large could stall the learning process and prevent the stream from being processed at an acceptable speed. Training sets that are too small will induce models that are poor at generalizing to new examples. Memory management of a wrapper scheme can only be conducted on a per-model basis, where memory can be freed by forgetting some of the models that were previously induced. Examples of wrapper approaches from the literature include Wang et al. [121], Street and Kim [114] and Chu and Zaniolo [25].

Purposefully adapted algorithms designed specifically for data stream problems offer several advantages over wrapper schemes. They can exert greater control over processing times per example, and can conduct memory management at a finer-grained level. Common varieties of machine learning approaches to classification fall into several general classes. These classes of method are discussed below, along with their potential for adaptation to data streams:

decision trees This class of method is the main focus of the thesis. Chapter 3 studies a successful adaptation of decision trees to data streams [32] and outlines the motivation for this choice.

rules Rules are somewhat similar to decision trees, as a decision tree can be decomposed into a set of rules, although the structure of a rule set can be more flexible than the hierarchy of a tree. Rules have an advantage that each rule is a disjoint component of the model that can be evaluated in isolation and removed from the model without major disruption, compared to the cost of restructuring decision trees. However, rules may be less efficient to process than decision trees, which can guarantee a single decision path per example. Ferrer-Troyano et al. [39, 40] have developed methods for inducing rule sets directly from streams.

lazy/nearest neighbour This class of method is described as *lazy* because in the batch learning setting no work is done during training, but all of the effort in classifying examples is delayed until predictions are required. The typical *nearest neighbour* approach will look for examples in the training set that are most similar to the example being classified, as the class labels of these examples are expected to be a reasonable indicator of the unknown class. The challenge with adapting these methods to the data stream setting is that training can not afford to be lazy, because it is not possible to store the entire training set. Instead the examples that are remembered must be managed so that they fit into limited memory. An intuitive solution to this problem involves finding a way to merge new examples with the closest ones already in memory, the main question being what merging process will perform best. Another issue is that searching for the nearest neighbours is costly. This cost may be reduced by using efficient data structures designed to reduce search times. Nearest neighbour based methods are a popular research topic for data stream classification. Examples of systems include [89, 47, 81, 8].

support vector machines/neural networks Both of these methods are related and of similar power, although *support vector machines* [22] are induced via an alternate training method and are a hot research topic due to their flexibility in allowing various *kernels* to offer tailored solutions. Memory management for support vector machines could be based on limiting the number of support vectors being induced. Incremental training of support vector machines has been explored previously, for example [115]. Neural networks are relatively straightforward to train on a data stream. A real world application using neural networks is given by Gama and Rodrigues [53]. The typical procedure assumes a fixed network, so there is no memory management problem. It is straightforward to use the typical *backpropagation* training method on a stream of examples, rather than repeatedly scanning a fixed training set as required in the batch setting.

Bayesian methods These methods are based around *Bayes' theorem* and compute probabilities in order to perform *Bayesian inference.* The simplest Bayesian method, Naive Bayes, is described in Section 5.2, and is a special case of algorithm that needs no adaptation to data streams. This is because it is straightforward to train incrementally and does not add structure to the model, so that memory usage is small and bounded. A single Naive Bayes model will generally not be as accurate as more complex models. The more general case of *Bayesian networks* is also suited to the data stream setting, at least when the structure of the network is known. Learning a suitable structure for the network is a more difficult problem. Hulten and Domingos [71] describe a method of learning Bayesian networks from data streams using Hoeffding bounds. Bouckaert [10] also presents a solution.

meta/ensemble methods These methods wrap around other existing methods, typically building up an *ensemble* of models. Examples of this include [102, 89]. This is the other major class of algorithm studied in-depth by this thesis, beginning in Chapter 6.

Gaber et al. [48] survey the field of data stream classification algorithms and list those that they believe are major contributions. Most of these have already been covered: Domingos and Hulten's *VFDT* [32], the decision tree algorithm studied in-depth by this thesis; *ensemble-based classification* by Wang et al. [121] that has been mentioned as a wrapper approach; *SCALLOP*, a rule-based learner that is the earlier work of Ferrer-Troyano et al. [39]; *ANNCAD*, which is a nearest neighbour method developed by Law and Zaniolo [89] that operates using *Haar wavelet* transformation of data and small classifier ensembles; and *LWClass* proposed by Gaber et al. [47], another nearest neighbour based technique, that actively adapts to fluctuating time and space demands by varying a distance threshold. The other methods in their survey that have not yet been mentioned include *on demand classification*. This method by Aggarwal et al. [1] performs dynamic collection of training examples into supervised *micro-clusters*. From these clusters, a nearest neighbour type classification is performed, where the goal is to quickly adapt to concept drift. The final method included in the survey is known as an *online information network* (*OLIN*) proposed by Last [88]. This method has a strong focus on concept drift, and uses a *fuzzy* technique to construct a model similar to decision trees, where the frequency of model building and training window size is adjusted to reduce error as concepts evolve.

Of the broad classes of algorithm, there are no existing benchmarks to determine which class is superior to any other at classifying data streams. Decision trees were chosen because they are a classic area of research, and there is existing evidence to suggest that they are an effective method for data stream classification.

1.4 Thesis Structure

The chapters that follow build the thesis in logical sequence:

Chapter 2 reviews the common methodologies in practice for evaluating data stream algorithms. After considering the desirable attributes of a comprehensive evaluation strategy a final framework is proposed. The framework includes simulation of three environments by varying memory demands, and a set of synthetic data generators in preparation for evaluation of learning algorithms in the thesis.

Chapter 3 introduces the basic decision tree learning algorithm that operates on data streams by relying on Hoeffding bounds to decide when sufficient information has been seen to justify tree expansion. Many aspects of the basic algorithm are explored.

Chapter 4 sets out to resolve the issue of how Hoeffding trees should handle continuous numeric attributes. Potential approaches are surveyed, and then candidates are tested using the evaluation framework from Chapter 2 in an experiment to determine which strategy performs best.

Chapter 5 conducts a study of prediction methods, using the evaluation framework from Chapter 2 to produce empirical evidence in support of a newly suggested method. This hybrid approach adaptively combines the strengths of two previous methods.

Chapter 6 surveys common methods for improving accuracy in the batch setting via ensembles of models. Possibilities for transferring these methods to data streams are explored, along with the theoretical implications of combining increasing numbers of models when memory is limited. Algorithms are elaborated, including a novel method of inducing *option trees*—a generalized representation of decision trees offering the benefits of ensembles in a single and potentially more memory-efficient structure.

Chapter 7 experimentally compares the main algorithms suggested in Chapter 6 using the evaluation framework from Chapter 2. Some of the results on data stream problems are surprising, and some do not entirely match the expectations arising from generalizations in the batch setting, so a deeper analysis and discussion looks more closely at the issues.

Chapter 8 summarizes the findings and lists the contributions made. It concludes with a discussion of potential for future work.

Chapter 2

Experimental Setting

This chapter establishes the settings under which experiments are conducted, creating the framework necessary to place various learning algorithms under test. The experimental methodology adopted by this thesis is motivated by the requirements of the end user and their desired application.

A user wanting to classify examples in a stream of data will have a set of requirements. They will have a certain volume of data, composed of a number of features per example, and a rate at which examples arrive. They will have the computing hardware on which the training of the model and the classification of new examples is to occur. Users will naturally seek the most accurate predictions possible on the hardware provided. They are, however, more likely to accept a solution that sacrifices accuracy in order to function, than no solution at all. Within reason the user's requirements may be relaxed, such as reducing the training features or upgrading the hardware, but there comes a point at which doing so would be unsatisfactory.

The behaviour of a data stream learning algorithm has three dimensions of interest—the amount of space (computer memory) required, the time required to learn from training examples and to predict labels for new examples, and the error of the predictions. When the user's requirements cannot be relaxed any further, the last remaining element that can be tuned to meet the demands of the problem is the *effectiveness* of the learning algorithm—the ability of the algorithm to output minimum error in limited time and space.

The error of an algorithm is the dimension that people would like to control the most, but it is the least controllable. The biggest factors influencing error are the *representational power* of the algorithm, how capable the model is at capturing the true underlying concept in the stream, and its *generalization power*, how successfully it can ignore noise and isolate useful patterns in the data.

Adjusting the time and space used by an algorithm can influence error. Time and space are interdependent. By storing more pre-computed information, such as look up tables, an algorithm can run faster at the expense of space. An algorithm can also run faster by processing less information, either by stopping early or storing less, thus having less data to process. The more time an algorithm has to process, or the more information that is processed, the more likely it is that error can be reduced.

The time and space requirements of an algorithm can be controlled by design. The algorithm can be optimised to reduce memory footprint and runtime. More directly, an algorithm can be made aware of the resources it is using and dynamically adjust. For example, an algorithm can take a memory limit as a parameter, and take steps to obey the limit. Similarly, it could be made aware of the time it is taking, and scale computation back to reach a time goal.

The easy way to limit either time or space is to stop once the limit is reached, and resort to the best available output at that point. For a time limit, continuing to process will require the user to wait longer, a compromise that may be acceptable in some situations. For a space limit, the only way to continue processing is to have the algorithm specifically designed to discard some of its information, hopefully information that is least important. Additionally, time is highly dependent on physical processor implementation, whereas memory limits are universal. The space requirement is a hard overriding limit that is ultimately dictated by the hardware available. An algorithm that requests more memory than is available will cease to function, a consequence that is much more serious than either taking longer, or losing accuracy, or both.

It follows that the space dimension should be fixed in order to evaluate algorithmic performance. Accordingly, to evaluate the ability of an algorithm to meet user requirements, a memory limit is set, and the resulting time and error performance of the algorithm is measured on a data stream. Different memory limits have been chosen to gain insight into general performance of algorithmic variations by covering a range of plausible situations.

Several elements are covered in order to establish the evaluation framework used in this thesis. Evaluation methods already established in the field are surveyed in Section 2.1. Possible procedures are compared in 2.2 and the final evaluation framework is described in Section 2.3. The memory limits used for testing are motivated in Section 2.4, and Section 2.5 describes the data streams used for testing. Finally, Section 2.6 analyzes the speeds and sizes of the data streams involved. The particular algorithms under examination are the focus of the remainder of the thesis.

2.1 Previous Evaluation Practices

This section assumes that the critical variable being measured by evaluation processes is the *accuracy* of a learning algorithm. Accuracy, or equivalently its converse, *error*, may not be the only concern, but it is usually the most pertinent one. Accuracy is typically measured as the percentage of correct classifications that a model makes on a given set of data, the most accurate learning algorithm is the one that makes the fewest mistakes when predicting labels of examples. With classification problems, achieving the highest possible accuracy is the most immediate and obvious goal. Having a reliable estimate of accuracy enables comparison of different methods, so that the best available method for a given problem can be determined.

It is very *optimistic* to measure the accuracy achievable by a learner on the same data that was used to train it, because even if a model achieves perfect accuracy on its training data this may not reflect the accuracy that can be expected on unseen data—its *generalization* accuracy. For the evaluation of a learning algorithm to measure practical usefulness, the algorithm's ability to generalize to previously unseen examples must be tested. A model is said to *overfit* the data if it tries too hard to explain the training data, which is typically noisy, so performs poorly when predicting the class label of examples it has not seen before. One of the greatest challenges of machine learning is finding algorithms that can avoid the problem of overfitting.

2.1.1 Batch Setting

Previous work on the problem of evaluating batch learning has concentrated on making the best use of a limited supply of data. When the number of examples available to describe a problem is in the order of hundreds or even less then reasons for this concern are obvious. When data is scarce, ideally all data that is available should be used to train the model, but this will leave no remaining examples for testing. The following methods discussed are those that have in the past been considered most suitable for evaluating batch machine learning algorithms, and are studied in more detail by Kohavi [82].

The *holdout* method divides the available data into two subsets that are mutually exclusive. One of the sets is used for training, the *training* set, and the remaining examples are used for testing, the *test* or *holdout* set. Keeping these sets separate ensures that generalization performance is being measured. Common size ratios of the two sets used in practice are 1/2 training and 1/2 test, or 2/3 training and 1/3 test. Because the learner is not provided the full amount of data for training, assuming that it will improve given more data, the performance estimate will be pessimistic. The main criticism of the holdout method in the batch setting is that the data is not used efficiently, as many examples may never be used to train the algorithm. The accuracy estimated from a single holdout can vary greatly depending on how the sets are divided. To mitigate this effect, the process of *random subsampling* will perform multiple runs of the holdout procedure, each with a different random division of the data, and average the results. Doing so also enables measurement of the accuracy estimate's variance. Unfortunately this procedure violates the assumption that the training and test set are independent—classes over-represented in one set will be under-represented in the other, which can skew the results.

In contrast to the holdout method, *cross-validation* maximizes the use of examples for both training and testing. In k-fold cross-validation the data is randomly divided into k independent and approximately equal-sized *folds*. The evaluation process repeats k times, each time a different fold acts as the holdout set while the remaining folds are combined and used for training. The final accuracy estimate is obtained by dividing the total number of correct classifications by the total number of examples. In this procedure each available example is used $k - 1$ times for training and exactly once for testing. This method is still susceptible to imbalanced class distribution between folds. Attempting to reduce this problem, *stratified cross-validation* distributes the labels evenly across the folds to approximately reflect the label distribution of the entire data. Repeated cross-validation repeats the cross-validation procedure several times, each with a different random partitioning of the folds, allowing the variance of the accuracy estimate to be measured.

The *leave-one-out* evaluation procedure is a special case of cross-validation where every fold contains a single example. This means with a data set of n examples that n-fold cross validation is performed, such that n models are induced, each of which is tested on the single example that was held out. In special situations where learners can quickly be made to 'forget' a single training example this process can be performed efficiently, otherwise in most cases this procedure is expensive to perform. The leave-one-out procedure is attractive because it is completely deterministic and not subject to random effects in dividing folds. However, stratification is not possible and it is easy to construct examples where leave-one-out fails in its intended task of measuring generalization accuracy. Consider what happens when evaluating using completely random data with two classes and an equal number of examples per class—the best an algorithm can do is predict the majority class, which will always be incorrect on the example held out, resulting in an accuracy of 0%, even though the expected estimate should be 50%.

An alternative evaluation method is the *bootstrap* method introduced by Efron [35]. This method creates a *bootstrap sample* of a data set by sampling with replacement a training data set of the same size as the original. Under the process of sampling with replacement the probability that a particular example will be chosen is approximately 0.632, so the method is commonly known as the 0.632 bootstrap. All examples not present in the training set are used for testing, which will contain on average about 36.8% of the examples. The method compensates for lack of unique training examples by combining accuracies measured on both training and test data to reach a final estimate:

$$accuracy_{bootstrap} = 0.632 \times accuracy_{test} + 0.368 \times accuracy_{train} \tag{2.1}$$

As with the other methods, repeated random runs can be averaged to increase the reliability of the estimate. This method works well for very small data sets but suffers from problems that can be illustrated by the same situation that causes problems with leave-one-out, a completely random two-class data set—Kohavi [82] argues that although the true accuracy of any model can only be 50%, a classifier that memorizes the training data can achieve $accuracy_{train}$ of 100%, resulting in $accuracy_{bootstrap} = 0.632 \times 50\% + 0.368 \times 100\% = 68.4\%$. This estimate is more optimistic than the expected result of 50%.

Having considered the various issues with evaluating performance in the batch setting, the machine learning community has settled on stratified ten-fold cross-validation as the standard evaluation procedure, as recommended by Kohavi [82]. For increased reliability, ten repetitions of ten-fold cross-validation are commonly used. Bouckaert [9] warns that results based on this standard should still be treated with caution.

2.1.2 Data Stream Setting

The data stream setting has different requirements from the batch setting. In terms of evaluation, batch learning's focus on reusing data to get the most out of a limited supply is not a concern as data is assumed to be abundant. With plenty of data, generalization accuracy can be measured via the holdout method without the same drawbacks that prompted researchers in the batch setting to pursue other alternatives. The essential difference is that a large set of examples for precise accuracy measurement can be set aside for testing purposes without starving the learning algorithms of training examples.

Instead of maximizing data use, the focus shifts to trends over time—in the batch setting a single static model is the final outcome of training, whereas in the stream setting the model evolves over time and can be employed at different stages of growth. In batch learning the problem of limited data is overcome by analyzing and averaging multiple models produced with different random arrangements of training and test data. In the stream setting the problem of (effectively) unlimited data poses different challenges. One solution involves taking snapshots at different times during the induction of a model to see how much the model improves with further training.

Data stream classification is a relatively new field, and as such evaluation practices are not nearly as well researched and established as they are in the batch setting. Although there are many recent computer science papers about data streams, only a small subset actually deal with the stream classification problem as defined in this thesis. A survey of the literature in this field was done to sample typical evaluation practices. Eight papers were found representing examples of work most closely related to this study. The papers are Domingos and Hulten [32], Gama et al. [52], Gama et al. [50], Jin and Agrawal [77], Oza and Russell [101], Street and Kim [114], Fern and Givan [38], and Chu and Zaniolo [25]. Important properties of these papers are summarized in Tables 2.1 and 2.2.

The 'evaluation methods' column of Table 2.1 reveals that the most common method for obtaining accuracy estimates is to use a single holdout set. This is consistent with the argument that nothing more elaborate is required in the stream setting, although some papers use five-fold cross-validation, and Fern and Givan [38] use different repeated sampling methods.

In terms of memory limits enforced during experimentation, the majority of papers do not address the issue and make no mention of explicit memory

Table 2.1: Paper survey part 1—Evaluation methods and data sources.

paper ref.	evaluation methods	enforced memory limits	data sources	max # of training examples	max # of test examples
[32]	holdout	40MB,	14 custom syn.	100m	50k
		80MB	1 private real	4m	267k
[52]	holdout	none	3 public syn. (UCI)	1m	250k
[50]	holdout	none	4 public syn. (UCI)	1.5m	250k
[77]	holdout?	60MB	3 public syn. (genF1/6/7)	10m	?
[101]	5-fold cv,	none	10 public real (UCI)	54k	13.5k
	holdout		3 custom syn.	80k	20k
			2 public real (UCIKDD)	465k	116k
[114]	5-fold cv,	none	2 public real (UCI)	45k	(5-fold cv)
	holdout		1 private real	33k	(5-fold cv)
			1 custom syn.	50k	10k
[38]	various	strict	4 public real (UCI)	100k	
		hardware	8 public real (spec95)	2.6m	
[25]	holdout	none	1 custom syn.	400k	50k
			1 private real	100k	?

Table 2.2: Paper survey part 2—Presentation styles.

paper ref.	presentation of results and comparisons
[32]	3 plots of accuracy vs examples
	1 plot of tree nodes vs examples
	1 plot of accuracy vs noise
	1 plot of accuracy vs concept size
	extra results (timing etc.) in text
[52]	1 table of error, training time & tree size
	(after 100k, 500k & 1m examples)
	1 plot of error vs examples
	1 plot of training time vs examples
	extra results (bias variance decomp., covertype results) in text
[50]	1 table of error, training time & tree size
	(after 100k, 500k, 750k/1m & 1m/1.5m examples)
[77]	1 plot of tree nodes vs noise
	2 plots of error vs noise
	3 plots of training time vs noise
	6 plots of examples vs noise
	1 plot of training time vs examples
	1 plot of memory usage vs examples
[101]	3 plots of online error vs batch error
	3 plots of accuracy vs examples
	2 plots of error vs ensemble size
	2 plots of training time vs examples
[114]	8 plots of error vs examples
[38]	25 plots of error vs ensemble size
	13 plots of error vs examples
	6 plots of error vs tree nodes
[25]	3 plots of accuracy vs tree leaves
	2 tables of accuracy for several parameters and methods
	3 plots of accuracy vs examples

limits placed on algorithms. Domingos and Hulten [32] makes the most effort to explore limited memory, and the followup work by Jin and Agrawal [77] is consistent by also mentioning a fixed limit. The paper by Fern and Givan [38] is a specialized study in CPU branch prediction that carefully considers hardware memory limitations.

The 'data sources' column lists the various sources of data used for evaluating data stream algorithms. Synthetic data (abbreviated syn. in the table), is artificial data that is randomly generated, so in theory is unlimited in size, and is noted as either public or custom. Custom data generators are those that are described for the first time in a paper, unlike public synthetic data that have been used before and where source code for their generation is freely available. Real data is collected from a real-world problem, and is described as being either public or private. All public sources mention where they come from, mostly from UCI [7], although Jin and Agrawal [77] make use of the generator described in Section 2.5.6, and Fern and Givan [38] use benchmarks specific to the CPU branch prediction problem. Section 2.5 has more discussion about common data sources.

Reviewing the numbers of examples used to train algorithms for evaluation the majority of previous experimental evaluations use less than one million training examples. Some papers use more than this, up to ten million examples, and only very rarely is there any study like Domingos and Hulten [32] that is in the order of tens of millions of examples. In the context of data streams this is disappointing, because to be truly useful at data stream classification the algorithms need to be capable of handling very large (potentially infinite) streams of examples. Only demonstrating systems on small amounts of data does not build a convincing case for capacity to solve more demanding data stream applications.

There are several possible reasons for the general lack of training data for evaluation. It could be that researchers come from a traditional machine learning background with entrenched community standards, where results involving cross-validation on popular real-world data sets are expected for credibility, and alternate practices are less understood. Emphasis on using real-world data will restrict the sizes possible, because as explained in Section 2.5.1 there is very little data freely available that is suitable for data stream evaluation. Another reason could be that the methods are being directly compared with batch learning algorithms, as several of the papers do, so the sizes may deliberately be kept small to accommodate batch learning. Hopefully no evaluations are intentionally small due to proposed data stream algorithms being too slow or memory hungry to cope with larger amounts of data in reasonable time or memory, because this would raise serious doubts about the algorithm's practical utility.

In terms of the sizes of test sets used, for those papers using holdout and where it could be determined from the text, the largest test set surveyed was less than 300 thousand examples in size, and some were only in the order of tens of thousands of examples. This suggests that the researchers believe that such sizes are adequate for accurate reporting of results.

Table 2.2 summarizes the styles used to present and compare results in the papers. The most common medium used for displaying results is the graphical plot, typically with the number of training examples on the x-axis. This observation is consistent with the earlier point that trends over time should be a focus of evaluation. The classic *learning curve* plotting accuracy/error versus training examples is the most frequent presentation style. Several other types of plot are used to discuss other behaviours such as noise resistance and model sizes. An equally reasonable but less common style presents the results as figures in a table, perhaps not as favoured because less information can be efficiently conveyed this way.

In terms of claiming that an algorithm significantly outperforms another, the accepted practice is that if a learning curve looks better at some point during the run (attains higher accuracy, and the earlier the better) and manages to stay that way by the end of the evaluation, then it is deemed a superior method. Most often this is determined from a single holdout run, and with an independent test set containing 300 thousand examples or less. It is rare to see a serious attempt at quantifying the significance of results with confidence intervals or similar checks. Typically it is claimed that the method is not highly sensitive to the order of data, that is, doing repeated random runs would not significantly alter the results.

A claim of this thesis is that in order to adequately evaluate data stream classification algorithms they need to be tested on large streams, in the order of hundreds of millions of examples where possible, and under explicit memory limits. Any less than this does not actually test algorithms in a realistically challenging setting. This is claimed because it is possible for learning curves to cross after substantial training has occurred, as discussed in Section 2.3 and seen later in the experimental results, for example Figure 4.5 on page 87.

Almost every data stream paper argues that innovative and efficient algorithms are needed to handle the substantial challenges of data streams but the survey shows that few of them actually follow through by testing candidate algorithms appropriately. The best paper found, Domingos and Hulten [32], represents a significant inspiration for this thesis because it also introduces the base algorithm expanded upon in Chapter 3 onwards. The paper serves as a model of what realistic evaluation should involve—limited memory to learn in, millions of examples to learn from, and several hundred thousand test examples.

2.2 Evaluation Procedures for Data Streams

The evaluation procedure of a learning algorithm determines which examples are used for training the algorithm, and which are used to test the model output by the algorithm. The procedure used historically in batch learning has partly depended on data size. Small data sets with less than a thousand examples, typical in batch machine learning benchmarking, are suited to the methods that extract maximum use of the data, hence the established procedure of ten repetitions of ten-fold cross-validation. As data sizes increase, practical time limitations prevent procedures that repeat training too many times. It is commonly accepted with considerably larger data sources that it is necessary to reduce the numbers of repetitions or folds to allow experiments to complete in reasonable time. With the largest data sources attempted in batch learning, on the order of hundreds of thousands of examples or more, a single holdout run may be used, as this requires the least computational effort. A justification for this besides the practical time issue may be that the reliability lost by losing repeated runs is compensated by the reliability gained by sheer numbers of examples involved.

When considering what procedure to use in the data stream setting, one of the unique concerns is how to build a picture of accuracy over time. Two main approaches were considered, the first a natural extension of batch evaluation, and the second an interesting exploitation of properties unique to data stream algorithms.

2.2.1 Holdout

When batch learning reaches a scale where cross-validation is too time consuming, it is often accepted to instead measure performance on a single holdout set. This is most useful when the division between train and test sets have been pre-defined, so that results from different studies can be directly compared. Viewing data stream problems as a large-scale case of batch learning, it then follows from batch learning practices that a holdout set is appropriate.

To track model performance over time, the model can be evaluated periodically, for example, after every one million training examples. Testing the model too often has potential to significantly slow the evaluation process, depending on the size of the test set.

A possible source of holdout examples is new examples from the stream that have not yet been used to train the learning algorithm. A procedure can 'look ahead' to collect a batch of examples from the stream for use as test examples, and if efficient use of examples is desired they can then be given to the algorithm for additional training after testing is complete. This method would be preferable in scenarios with concept drift, as it would measure a model's ability to adapt to the latest trends in the data.

Since no concept drift is assumed, a single static held out set should be sufficient, which avoids the problem of varying estimates between potential test sets. Assuming that the test set is independent and sufficiently large relative to the complexity of the target concept, it will provide an accurate measurement of generalization accuracy. As noted when looking at other studies, test set sizes on the order of tens of thousands of examples have previously been considered sufficient.

2.2.2 Interleaved Test-Then-Train

An alternate scheme for evaluating data stream algorithms is to interleave testing with training. Each individual example can be used to test the model before it is used for training, and from this the accuracy can be incrementally updated. When intentionally performed in this order, the model is always being tested on examples it has not seen. This scheme has the advantage that no holdout set is needed for testing, making maximum use of the available data. It also ensures a smooth plot of accuracy over time, as each individual example will become increasingly less significant to the overall average.

The disadvantages of this approach are that it makes it difficult to accurately separate and measure training and testing times. Also, the true accuracy that an algorithm is able to achieve at a given point is obscured—algorithms will be punished for early mistakes regardless of the level of accuracy they are

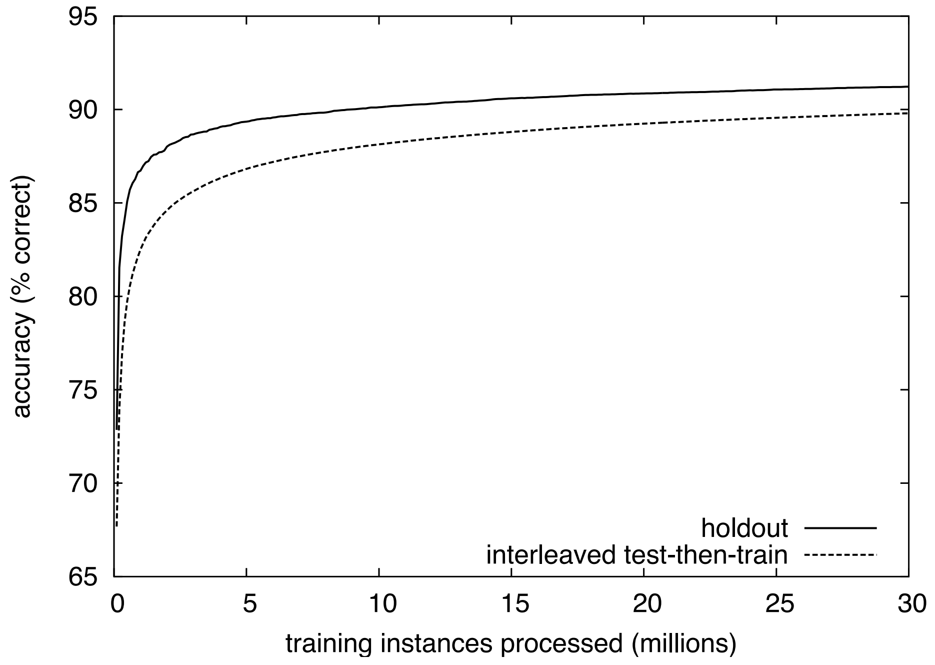

Figure 2.1: Learning curves produced for the same learning situation by two different evaluation methods, recorded every 100,000 examples.

eventually capable of, although this effect will diminish over time.

With this procedure the statistics are updated with every example in the stream, and can be recorded at that level of detail if desired. For efficiency reasons a sampling parameter can be used to reduce the storage requirements of the results, by recording only at periodic intervals like the holdout method.

2.2.3 Comparison

Figure 2.1 is an example of how learning curves can differ between the two approaches given an identical learning algorithm and data source. The holdout method measures immediate accuracy at a particular point, without memory of previous performance. During the first few million training examples the graph is not smooth. If the test set were small thus unreliable or the algorithm more unstable then fluctuations in accuracy could be much more noticeable. The interleaved method by contrast measures the average accuracy achieved to a given point, thus after 30 million training examples, the generalization accuracy has been measured on every one of the 30 million examples, rather than the independent one million examples used by the holdout. This explains why the interleaved curve is smooth. It also explains why the estimate of accuracy is more pessimistic, because during early stages of learning the model was less accurate, pulling the average accuracy down.

The interleaved method makes measuring estimates of both time and accuracy more difficult. It could be improved perhaps using a modification that introduces exponential decay, but this possibility is reserved for future work. The holdout evaluation method offers the best of both schemes, as the averaged accuracy that would be obtained via interleaved test-then-train can be estimated by averaging consecutive ranges of samples together. Having considered the relative merits of the approaches, the holdout method constitutes the foundation of the experimental framework described next.

2.3 Testing Framework

Algorithm 1 Evaluation procedure.

```
Fix $m_{bound}$, the maximum amount of memory allowed for the model
Hold out $n_{test}$ examples for testing
while further evaluation is desired do
  start training timer
  for $i = 1$ to $n_{train}$ do
    get next example $e_{train}$ from training stream
    train and update model with $e_{train}$, ensuring that $m_{bound}$ is obeyed
  end for
  stop training timer and record training time
  start test timer
  for $i = 1$ to $n_{test}$ do
    get next example $e_{test}$ from test stream
    test model on $e_{test}$ and update accuracy
  end for
  stop test timer and record test time
  record model statistics (accuracy, size etc.)
end while
```

Algorithm 1 lists pseudo-code of the evaluation procedure used for experimental work in this thesis. The process is similar to that used by Domingos and Hulten [32], the study that was found to have the most thorough evaluation practices of those surveyed in Section 2.1.2. It offers flexibility regarding which statistics are captured, with the potential to track many behaviours of interest.

The n_{train} parameter determines how many examples will be used for training before an evaluation is performed on the test set. A set of n_{test} examples

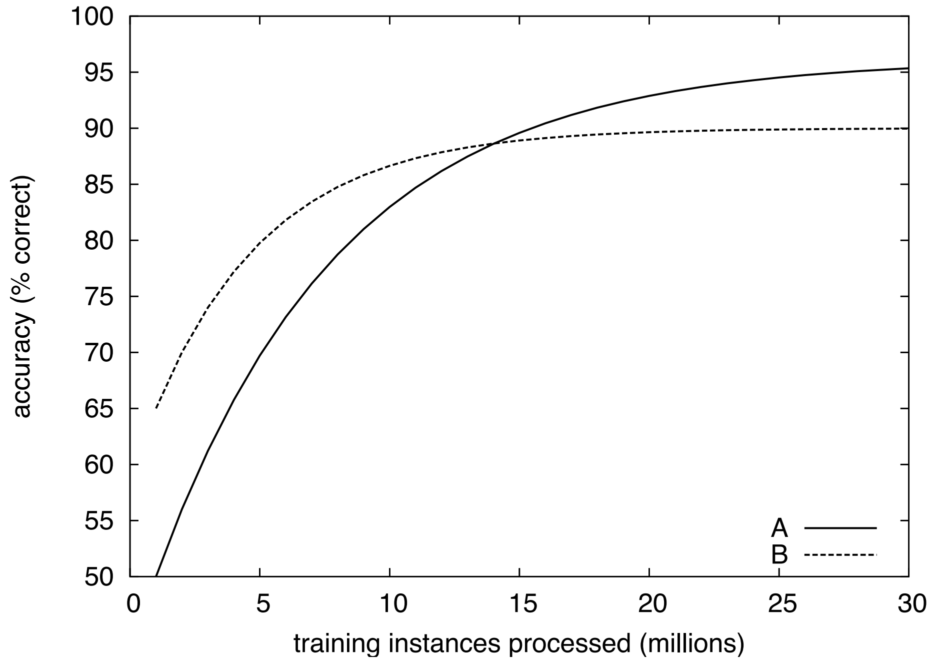

Figure 2.2: Learning curves demonstrating the problem of stopping early.

is held aside for testing. In the data stream case without concept drift this set can be easily populated by collecting the first n_{test} examples from the stream.

To get reliable timing estimates, n_{train} and n_{test} need to be sufficiently large. In the actual implementation, the timer measured the CPU runtime of the relevant thread, in an effort to reduce problems caused by the multithreaded operating system sharing other tasks. In all experiments, n_{test} was set to one million examples, which helps to measure timing but also ensures reliability of the accuracy estimates, where according to Table 2.1 previous studies in the field have typically used a tenth of this amount or even less.

The framework is designed to test an algorithm that tends to accumulate information over time, so the algorithm will desire more memory as it trains on more examples. The algorithm needs to be able to limit the total amount of memory used, thus obey m_{bound}, no matter how much training takes place.

One of the biggest issues with the evaluation is deciding when to stop training and start testing. In small memory situations, some algorithms will reach a point where they have exhausted all memory and can no longer learn new information. At this point the experiment can be terminated, as the results will not change from that point.

More problematic is the situation where time or training examples are exhausted before the final level of performance can be observed. Consider

Figure 2.2. Prior to 14 million examples, algorithm B is the clear choice in terms of accuracy, however in the long run it does not reach the same level of accuracy as algorithm A. Which algorithm is actually better depends on the application. If there is a shortage of time or data, algorithm B may be the better choice. Not only this, but if the models are employed for prediction during early stages of learning, then algorithm B will be more useful at the beginning.

To rule out any effect that data order may have on the learning process, the evaluation procedure may be run multiple times, each time with a different set of training data from the same problem. The observations gathered from each run can then be averaged into a single result. An advantage of this approach is that the variance of behaviour can also be observed. Ideally the data between runs will be unique, as is possible with synthetically generated or other abundant data sources. If data is lacking at least the training examples can be reordered.

An ideal method of evaluation would wait until an accuracy plateau is observable for every candidate before termination. It would also run multiple times with different data orders to establish confidence in the results. Unfortunately neither of these scenarios are feasible considering the large amount of experimental work needed for this thesis.

To see why this is so, consider the time required to run all of the experiments in this thesis. There are 19 different data sources. There are three memory limits, but only two are counted here because experiments terminate early in the smallest environment. There are at least 20 different variants of algorithm being tested, which underestimates the full amount of work by ignoring background experimentation such as performing bias/variance decomposition in Chapter 7. If ten hours are required per run, the total time required is:

$$19 \text{ data sets} \times 2 \text{ environments} \times 20 \text{ algorithms} \times 10 \text{ hours} = 7600 \text{ hours}$$

So a conservative estimate is that 317 days or more than 45 weeks of linear computing time is required to generate the results. Running the evaluation process in parallel is straightforward, so with several machines the practical runtime can be reduced, but not without access to substantial computing resources. Requiring multiple runs or more than ten hours per run will readily inflate the computing time needed.

For this practical reason, all experiments allowed a maximum of ten hours training time. The time required for the entire evaluation process is slightly longer than ten hours due to time required for testing, which is not included in the limit. Aside from algorithms being tested in the smallest memory environment, nearly every algorithm trained for the full ten hour period and could have continued for longer if permitted. The only exceptions were a small number of cases where the most memory-hungry ensemble methods described in Chapter 7 became incapable of doing any more work in the memory allowed.

Regarding the problem that terminating evaluation too early can bias results, there are two causes of early termination, shortage of data or shortage of time. Data shortage is avoided by using synthetic data generators thereby having unlimited data. Time shortage is more problematic but as explained it is unreasonable to expect more than ten hours of training per evaluation run, which is considered to be a reasonable amount of time.

The question of when an algorithm is superior to another is decided by looking at the final result recorded after the ten hour evaluation completed. The accuracy results are reported as percentages to two decimal places, and if a method's final accuracy reading in this context is greater than another then it is claimed to be superior. As with other similar studies there is no attempt to strictly analyze the confidence of results, although differences of several percentages are more convincing than fractional differences.

Measuring the standard error of results via multiple runs would enable the confidence of results to be more formally analyzed, but every additional run would multiply time requirements. A less expensive but useful alternative might be to examine differences between algorithms in another way, using McNemar's test [26] for example, which can be computed by tracking the agreement between competing methods on each test example. Extra analysis such as this was not considered necessary for this thesis, but the idea presents opportunity for future research into evaluation of data stream classification.

Two factors add informal confidence in the results obtained. Firstly, the class of base algorithm being studied (fully described in Chapter 3) has low sensitivity to data order, as reported previously [68, 50] and confirmed in smaller scale initial experiments. This suggests that even if multiple runs were performed and averaged, the final results would not change very much. Secondly, this study uses test set sizes that are several times larger than the largest used in previous studies, further decreasing the likelihood that methods can achieve high accuracy via chance alone.

The methodology employed is largely consistent with other studies, only

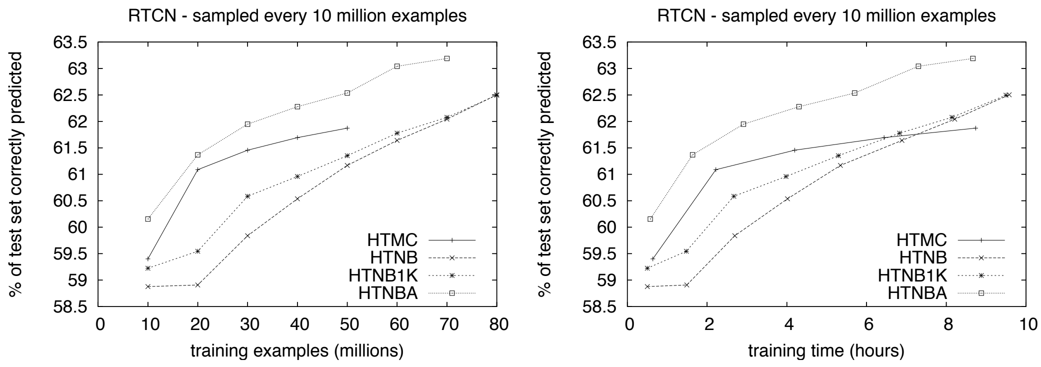

Figure 2.3: Example difference between learning curves based on training examples (left) versus training time (right).

on a greater scale. Where previous studies rarely trained on more than several million examples, the ten hour evaluation runs commonly involve several *hundreds* of millions of examples. There are two main ways that results are presented, as graphical plots and in tables. Graphical plots are interspersed with the text where they seem appropriate for demonstrating relevant behaviour. Unfortunately, because the graphs need to be of sufficient size to reasonably compare multiple methods, only selected graphs are shown. The space required to print readable graphs of every method on every data source in every environment and every interesting dimension is simply too high to justify.

The information collected during a run of the evaluation procedure can be plotted in several ways to give a visual analysis of an algorithm's behaviour. Of critical interest is the learning curve, as discussed previously, where the accuracy of the algorithm's predictions is plotted against either time or number of training examples. Previous studies have always plotted learning curves on a per-example basis, rather than a per-time basis that is dependent on the speed of computing hardware. This thesis uses both styles interchangeably, which can slightly alter the appearance of relationships between methods, see Figure 2.3. The main difference between these styles is that time-based plots have every line spanning almost the full width of the graph rather than having slower methods terminating earlier. Plotted this way, per-example performance is still observable because each sample point represents a fixed number of training examples, such that more frequent points in a line represent more training examples being processed. Often with these graphs the sampling period is purposely adjusted to ensure a reasonable separation between sample points, enhancing readability. Where appropriate the sampling period is noted in the title of the graph. The difference between per-example and per-time plots can be dramatic when the speeds of algorithms differ greatly, such as the example in Figure 3.2 on page 50.

The graphs are supplemented by tables in Appendix A. The tables cannot convey subtle changes over time like graphs can, but they can report the complete set of end results, allowing comparison of the key results for every method/data source/environment combination. The tables record the final readings at the end of evaluation, which is sufficient information for determining the best methods. Presented in this form the final results are meaningfully summarized in a reasonable amount of space.

The other important property of a learning algorithm besides accuracy is speed. It is important to differentiate learning speed from processing speed. Learning speed refers to the number of examples required to reach a given accuracy level. Processing speed is the rate at which examples are processed, consisting of both training and testing speeds, and can be measured in terms of examples processed per second. When comparing algorithms, it is the relative time taken on identical hardware that is important, as the absolute times will vary depending on the power of the computing hardware.

The hardware/software environment has a large influence on the results obtained, because a ten hour time limit is completely arbitrary when computing resources are unspecified. The experimental environment was purposely kept consistent for all time-dependent experiments, as follows:

Hardware: Intel Core2 6300 CPU running at 1.86Ghz with a 2048KB cache and 1GB of RAM

Software: Sun Java HotSpot Server VM (build 1.6.0_01-b06), running under GNU/Linux Fedora core 6

2.4 Environments

This section defines three environments that are simulated using memory limits, since memory limits cannot be ignored and can significantly limit capacity to learn from data streams. Potential practical deployment of data stream classification has been divided into scenarios of increasing memory utilization, from the restrictive *sensor* environment, to a typical consumer grade *handheld* PDA environment, to the least restrictive environment of a dedicated *server*.

Although technology advancements will mean that these environments are something of a moving target, the insights gained about the scalability of algorithms will still be valid. The environments chosen range from restrictive to generous, with an order of magnitude difference between them.

Note that when referring to memory sizes, the traditional meaning of the terms *kilobyte* and *megabyte* is adopted, such that 1 kilobyte = 1,024 bytes, and 1 megabyte = 1024^2 bytes = 1,048,576 bytes.

2.4.1 Sensor Network

This environment represents the most restrictive case, learning in 100 kilobytes of memory. Because this limit is so restrictive, it is an interesting test case for algorithm efficiency.

Sensor networks [4, 49] are a hot topic of research, and typically the nodes designed for these networks are low power devices with limited resources. In this setting it is often impractical to dedicate more than hundreds of kilobytes to processes because typically such devices do not support much working memory.

When memory limits are in the order of kilobytes, other applications requiring low memory usage also exist, such as specialized hardware in which memory is expensive. An example of such an application is CPU branch prediction, as explored by Fern and Givan [38]. Another example is a small 'packet sniffer' device designed to do real-time monitoring of network traffic [6].

2.4.2 Handheld Computer

In this case the algorithm is allowed 32 megabytes of memory. This simulates the capacity of lightweight consumer devices designed to be carried around by users and fit into a shirt pocket.

The ability to do analysis *on site* with a handheld device is desirable for certain applications. The papers [78] and [79] describe systems for analysing vehicle performance and stockmarket activity respectively. In both cases the authors describe the target computing hardware as personal handheld devices with 32 megabytes. Horovitz et al. [69] describe a road safety application using a device with 64 megabytes.

The promise of ubiquitous computing is getting closer with the widespread use of mobile phones, which with each generation are evolving into increasingly more powerful and multifunctional devices. These too fall into this category, representing large potential for portable machine learning given suitable algorithms. Imielinski and Nath [75] present a vision of this future 'dataspace'.

2.4.3 Server

This environment simulates either a modern laptop/desktop computer or server dedicated to processing a data stream. The memory limit assigned in this environment is 400 megabytes. Although at the time of writing this thesis a typical desktop computer may have several gigabytes of RAM, it is still generous to set aside this much memory for a single process. Considering that several algorithms have difficulty in fully utilizing this much working space, it seems sufficiently realistic to impose this limit.

A practical reason for imposing this limit is that the experiments need to terminate in reasonable time. In cases where memory is not filled by an algorithm it is harder to judge what the behaviour will be in the theoretical limit, but the practical ten hour limit is an attempt to run for sufficient time to allow accuracy to plateau.

There are many applications that fit into this higher end of the computing scale. An obvious task is analysing data arising from the Internet, as either web searches [61], web usage [113], site logs [112] or click streams [60]. Smaller scale computer networks also produce traffic of interest [90], as do other telecommunication activities [122], phone call logs for example. Banks may be interested in patterns of ATM transactions [62], and retail chains and online stores will want details about customer purchases [85]. Further still, there is the field of scientific observation [57], which can be astronomical [100], geophysical [97], or the massive volume of data output by particle accelerator experiments [70]. All of these activities are sources of data streams that users will conceivably want to analyze in a server environment.

2.5 Data Sources

For the purposes of research into data stream classification there is a shortage of suitable and publicly available real-world benchmark data sets. The UCI ML [7] and KDD [65] archives house the most common benchmarks for machine learning algorithms, but many of those data sets are not suitable for evaluating data stream classification. The KDD archive has several large data sets, but not classification problems with sufficient examples. The *Forest Covertype* data set is one of the largest, and that has less than 600,000 examples.

To demonstrate their systems, several researchers have used private real-world data that cannot be reproduced by others. Examples of this include the

Table 2.3: Properties of the data sources.

name	nominal	numeric	classes
RTS/RTSN	10	10	2
RTC/RTCN	50	50	2
RRBFS	0	10	2
RRBFC	0	50	2
LED	24	0	10
WAVE21	0	21	3
WAVE40	0	40	3
GENF1-F10	6	3	2

web trace from the University of Washington used by Domingos and Hulten to evaluate VFDT [32], and the credit card fraud data used by Wang et al. [121] and Chu and Zaniolo [25].

More typically, researchers publish results based on synthetically generated data. In many of these cases, the authors have invented unique data generation schemes for the purpose of evaluating their algorithm. Examples include the random tree generator also used to evaluate VFDT, and the custom generators described in Oza and Russell [101], Street and Kim [114] and Chu and Zaniolo [25]. Synthetic data has several advantages—it is easier to reproduce and there is little cost in terms of storage and transmission. Despite these advantages there is a lack of established and widely used synthetic data streams.

For this thesis, the data generators most commonly found in the literature have been collected, and and an extra scheme (RBF) has been introduced. For ease of reference, each data set variation is assigned a short name. The basic properties of each are summarized in Table 2.3.

2.5.1 Lack of Real-World Data

The first place to find real-world benchmark data for evaluating machine learning algorithms is the UCI machine learning repository [7]. Larger real-world data sets can be found in the UCI KDD archive [65], which was established to serve the needs of larger scale benchmarking.

Table 2.4 summarizes the sizes of the data sets in the KDD archive that were found to best suit the task of evaluating classification. The archive has other large data sets that are not considered because they are intended for topics with different requirements such as text categorization and time series classification.

Table 2.4: Sizes of data sets in UCI KDD archive that are suitable for evaluating classification.

name	attributes	training/test examples
Census-Income	40	199,523/99,762
COIL	17	340
Corel Image	89	68,040
Forest Covertype	54	581,012
Insurance Company (COIL2000)	86	5,822/4,000
Internet Usage	71	10,108
IPUMS	60	233,584
KDD Cup 1998	481	95,412
KDD Cup 1999	40	4,898,430/311,029
MS Anonymous Web	294	32,711

The data set with the most examples is *KDD Cup 1999* which has nearly five million training examples. The task is to detect network intrusion attempts. After initial experiments with this data it became clear that it is not a very useful benchmark—it is too easy to achieve near-perfect accuracy, there are many classes with a highly imbalanced distribution of examples between classes, making it hard to discern anything from the accuracies measured for competing methods. Further analysis of the data revealed that there is a high number of repeated examples, such that the number of unique training examples is several times less than the total number of examples specified. Brugger [19] has expressed concerns that this data set is flawed.

The next largest data set available is *Forest Covertype*, a more reasonable classification benchmark. It is not surprising that this data set has been used in several papers on data stream classification [52, 101], given the lack of alternatives.

For the style of evaluation required by this thesis, where several hundreds of millions of training examples are required to test genuine ability to handle data streams, these data sets are simply not adequate. The need to rely on artificial data is unfortunate but necessary.

2.5.2 Random Tree Generator

This generator is based on that proposed by Domingos and Hulten [32], producing concepts that in theory should favour decision tree learners. It constructs a decision tree by choosing attributes at random to split, and assigning a random class label to each leaf. Once the tree is built, new examples are generated by assigning uniformly distributed random values to attributes which then determine the class label via the tree.

The generator has parameters to control the number of classes, attributes, nominal attribute labels, and the depth of the tree. For consistency between experiments, two random trees were generated and fixed as the base concepts for testing—one *simple* and the other *complex*, where complexity refers to the number of attributes involved and the size of the tree.

The simple random tree (RTS) has ten nominal attributes with five values each, ten numeric attributes, two classes, a tree depth of five, with leaves starting at level three and a 0.15 chance of leaves thereafter. The final tree has 741 nodes, 509 of which are leaves.

The complex random tree (RTC) has 50 nominal attributes with five values each, 50 numeric attributes, two classes, a tree depth of ten, with leaves starting at level five and a 0.15 chance of leaves thereafter. The final tree has 127,837 nodes, 90,259 of which are leaves.

A degree of noise can be introduced to the examples after generation. In the case of discrete attributes and the class label, a probability of noise parameter determines the chance that any particular value is switched to something other than the original value. For numeric attributes, a degree of random noise is added to all values, drawn from a random Gaussian distribution with standard deviation equal to the standard deviation of the original values multiplied by noise probability. The streams RTSN and RTCN are introduced by adding 10% noise to the respective random tree data streams. It is hoped that experimenting with both noiseless and noisy versions of a problem can give insight into how well the algorithms manage noise.

2.5.3 Random RBF Generator

This generator was devised to offer an alternate concept type that is not necessarily as easy to capture with a decision tree model.

The RBF (Radial Basis Function) generator works as follows: A fixed number of random centroids are generated. Each center has a random position, a single standard deviation, class label and weight. New examples are generated by selecting a center at random, taking weights into consideration so that centers with higher weight are more likely to be chosen. A random direction is chosen to offset the attribute values from the central point. The length of the displacement is randomly drawn from a Gaussian distribution with standard deviation determined by the chosen centroid. The chosen centroid also determines the class label of the example. This effectively creates a normally distributed hypersphere of examples surrounding each central point with varying densities. Only numeric attributes are generated.

RRBFS refers to a simple random RBF data set—100 centers and ten attributes. RRBFC is more complex—1000 centers and 50 attributes. Both are two class problems.

2.5.4 LED Generator

This data source originates from the CART book [18]. An implementation in C was donated to the UCI [7] machine learning repository by David Aha. The goal is to predict the digit displayed on a seven-segment LED display, where each attribute has a 10% chance of being inverted. It has an optimal Bayes classification rate of 74%. The particular configuration of the generator used for experiments (LED) produces 24 binary attributes, 17 of which are irrelevant.

2.5.5 Waveform Generator

This generator shares its origins with LED, and was also donated by David Aha to the UCI repository. The goal of the task is to differentiate between three different classes of waveform, each of which is generated from a combination of two or three *base* waves. The optimal Bayes classification rate is known to be 86%. There are two versions of the problem. WAVE21 has 21 numeric attributes, all of which include noise. WAVE40 introduces an additional 19 irrelevant attributes.

2.5.6 Function Generator

This generator was introduced by Agrawal et al. in [2], and was a common source of data for early work on scaling up decision tree learners [96, 111, 54].

The generator produces a stream containing nine attributes, six numeric and three categorical, described in Table 2.5. Although not explicitly stated by the authors, a sensible conclusion is that these attributes describe hypothetical loan applications.

There are ten functions defined for generating binary class labels from the attributes. The functions are listed in Figures 2.4 and 2.5. Presumably these determine whether the loan should be approved. For the experiments the ten

Table 2.5: Function generator attributes.

name	description	values
salary	salary	uniformly distributed from 20K to 150K
commission	commission	**if** (*salary* $<$ 75K) **then** 0 **else** uniformly distributed from 10K to 75K
age	age	uniformly distributed from 20 to 80
elevel	education level	uniformly chosen from 0 to 4
car	make of car	uniformly chosen from 1 to 20
zipcode	zip code of town	uniformly chosen from 9 zipcodes
hvalue	value of house	uniformly distributed from $0.5k100000$ to $1.5k100000$ where $k \in \{1...9\}$ depending on *zipcode*
hyears	years house owned	uniformly distributed from 1 to 30
loan	total loan amount	uniformly distributed from 0 to 500K

functions are used as described, with a perturbation factor of 5% (referred to as GENF1-GENF10). Perturbation shifts numeric attributes from their true value, adding an offset drawn randomly from a uniform distribution, the range of which is a specified percentage of the total value range.

2.6 Generation Speed and Data Size

During evaluation the data is generated on-the-fly. This directly influences the amount of training examples that can be supplied in any given time period.

The speed of the data generators was measured in the experimental hardware/software environment. The results are shown in Table 2.6, where the full speed possible for generating each stream was estimated by timing how long it took to generate ten million examples. The possible speed ranges from around nine thousand examples per second on RTCN to over 500 thousand examples per second for the function generators GENFx. The biggest factor influencing speed is the number of attributes being generated, hence the fastest streams are those with the least attributes. The addition of noise to the streams also has a major impact on the speeds—going from RTS to RTSN and from RTC to RTCN causes the speed to roughly halve, where the only difference between these variants is the addition of noise. This result is consistent with the notion that a dominant cost of generating the streams is the time needed to generate random numbers, as adding noise involves producing at least one additional random number per attribute.

In terms of the sizes of the examples, the assumption is made that storage

```
1. if $(age < 40) \vee (age \geq 60)$ then
       $group = A$
   else
       $group = B$

2. if $((age < 40) \wedge (50000 \leq salary \leq 100000)) \vee$
       $((40 \leq age < 60) \wedge (75000 \leq salary \leq 125000)) \vee$
       $((age \geq 60) \wedge (25000 \leq salary \leq 75000))$ then
       $group = A$
   else
       $group = B$

3. if $((age < 40) \wedge (elevel \in [0..1])) \vee$
       $((40 \leq age < 60) \wedge (elevel \in [1..3])) \vee$
       $((age \geq 60) \wedge (elevel \in [2..4]))$ then
       $group = A$
   else
       $group = B$

4. if $((age < 40) \wedge (elevel \in [0..1]$ ?
           $(25000 \leq salary \leq 75000) : (50000 \leq salary \leq 100000))) \vee$
       $((40 \leq age < 60) \wedge (elevel \in [1..3]$ ?
           $(50000 \leq salary \leq 100000) : (75000 \leq salary \leq 125000))) \vee$
       $((age \geq 60) \wedge (elevel \in [2..4]$ ?
           $(50000 \leq salary \leq 100000) : (25000 \leq salary \leq 75000)))$ then
       $group = A$
   else
       $group = B$

5. if $((age < 40) \wedge ((50000 \leq salary \leq 100000)$ ?
           $(100000 \leq loan \leq 300000) : (200000 \leq loan \leq 400000))) \vee$
       $((40 \leq age < 60) \wedge ((75000 \leq salary \leq 125000)$ ?
           $(200000 \leq loan \leq 400000) : (300000 \leq loan \leq 500000))) \vee$
       $((age \geq 60) \wedge ((25000 \leq salary \leq 75000)$ ?
           $(30000 \leq loan \leq 500000) : (100000 \leq loan \leq 300000)))$ then
       group = A
   else
       group = B
```

Figure 2.4: Generator functions 1-5.

```
6.  if $((age < 40) \wedge (50000 \leq (salary + commission) \leq 100000)) \vee$
        $((40 \leq age < 60) \wedge (75000 \leq (salary + commission) \leq 125000)) \vee$
        $((age \geq 60) \wedge (25000 \leq (salary + commission) \leq 75000))$ then
        $group = \mathrm{A}$
    else
        $group = \mathrm{B}$

7.  if $(0.67 \times (salary + commission) - 0.2 \times loan - 20000 > 0)$ then
        $group = \mathrm{A}$
    else
        $group = \mathrm{B}$

8.  if $(0.67 \times (salary + commission) - 5000 \times elevel - 20000 > 0)$ then
        $group = \mathrm{A}$
    else
        $group = \mathrm{B}$

9.  if $(0.67 \times (salary + commission) - 5000 \times elevel$
        $- 0.2 \times loan - 10000 > 0)$ then
        $group = \mathrm{A}$
    else
        $group = \mathrm{B}$

10. if $(hyears \geq 20)$ then
        $equity = 0.1 \times hvalue \times (hyears - 20)$
    else
        $equity = 0$

    if $(0.67 \times (salary + commission) - 5000 \times elevel$
        $- 0.2 \times equity - 10000 > 0)$ then
        $group = \mathrm{A}$
    else
        $group = \mathrm{B}$
```

Figure 2.5: Generator functions 6-10.

Table 2.6: Generation speed and data size of the streams.

stream	examples per second (thousands)	attributes	bytes per example	examples in 10 hours (millions)	terabytes in 10 hours
RTS	274	20	168	9866	1.50
RTSN	97	20	168	3509	0.53
RTC	19	100	808	667	0.49
RTCN	9	100	808	338	0.25
RRBFS	484	10	88	17417	1.39
RRBFC	120	50	408	4309	1.60
LED	260	24	200	9377	1.71
WAVE21	116	21	176	4187	0.67
WAVE40	56	40	328	2003	0.60
GENF1	527	9	80	18957	1.38
GENF2	531	9	80	19108	1.39
GENF3	525	9	80	18917	1.38
GENF4	523	9	80	18838	1.37
GENF5	518	9	80	18653	1.36
GENF6	524	9	80	18858	1.37
GENF7	527	9	80	18977	1.38
GENF8	524	9	80	18848	1.37
GENF9	519	9	80	18701	1.36
GENF10	527	9	80	18957	1.47

of each attribute and class label requires eight bytes of memory, matching the actual Java implementation where all values are stored as double precision floating point numbers (Section 3.4). Certain attribute types could be stored more efficiently, but this approach offers maximum flexibility, and storing continuous values in less space would reduce precision.

Considering the entire evaluation period of ten hours, the total number of examples that can be produced at full speed range from around 300 to 19,000 million. With the eight-byte-per-attribute assumption, this translates to between approximately 0.25 to 1.7 terabytes of data. With data volumes of this magnitude the choice of generating data on-the-fly is justified, a solution that is cheaper than finding resources to store and retrieve several terabytes of data.

2.7 Summary

Aiming to best meet user requirements for data stream classification, the necessity for memory-limited algorithms has been argued. Looking first at prior studies, a comprehensive evaluation framework has been described, testing methods on a scale not previously attempted. The framework is complemented with three simulated memory-limited environments, defined to cover the range of potential deployment scenarios. A suite of synthetic benchmark data streams have been proposed, and their properties studied. The evaluation framework enables experimental work and comparison of algorithms to be performed throughout the thesis.

Chapter 3

Hoeffding Trees

This chapter describes the Hoeffding tree algorithm, chosen as the base method on which to build this study of data stream classification and experimental evaluation. The reasons for choosing this algorithm are elaborated below. Most importantly, the algorithm is an ideal starting point, as it is innovative and effective at high speed data stream classification, yet it also presents opportunities for improvement.

Hoeffding trees were introduced by Domingos and Hulten in the paper "Mining High-Speed Data Streams" [32]. They refer to their implementation as *VFDT*, an acronym for **V**ery **F**ast **D**ecision **T**ree learner. In that paper the Hoeffding tree algorithm is the basic theoretical algorithm, while VFDT introduces several enhancements for practical implementation. In this thesis the term *Hoeffding tree* refers to any variation or refinement of the basic principle, VFDT included.

In further work Domingos and Hulten went on to show how their idea can be generalized [71], claiming that any learner based on discrete search can be made capable of processing a data stream. The key idea depends on the use of *Hoeffding bounds*, described in Section 3.1. While from this point of view VFDT may only be one instance of a more general framework, not only is it the original example and inspiration for the general framework, but because it is based on decision trees it performs very well, for reasons given shortly.

Hoeffding trees are being studied because they represent current state-of-the-art for classifying high speed data streams. The algorithm fulfills the requirements necessary for coping with data streams while remaining efficient, an achievement that was rare prior to its introduction. Previous work on scaling up decision tree learning produced systems such as *SLIQ* [96], *SPRINT* [111] and *RAINFOREST* [54]. These systems perform batch learning of decision trees from large data sources in limited memory by performing multiple passes over the data and using external storage. Such operations are not suitable for high speed stream processing.

Other previous systems that are more suited to data streams are those that were designed exclusively to work in a single pass, such as the incremental systems *ID5R* [117] and its successor *ITI* [118], and other earlier work on incremental learning. Systems like this were considered for data stream suitability by Domingos and Hulten, who found them to be of limited practical use. In some cases these methods require more effort to update the model incrementally than to rebuild the model from scratch. In the case of ITI, all of the previous training data must be retained in order to revisit decisions, prohibiting its use on large data sources.

The Hoeffding tree induction algorithm induces a decision tree from a data stream incrementally, briefly inspecting each example in the stream only once, without need for storing examples after they have been used to update the tree. The only information needed in memory is the tree itself, which stores sufficient information in its leaves in order to grow, and can be employed to form predictions at any point in time between processing training examples.

Domingos and Hulten present a proof guaranteeing that a Hoeffding tree will be 'very close' to a decision tree learned via batch learning. This shows that the algorithm can produce trees of the same quality as batch learned trees, despite being induced in an incremental fashion. This finding is significant because batch learned decision trees are among the best performing machine learning models. The classic decision tree learning schemes *C4.5* [104] and *CART* [18], two similar systems that were independently developed, are widely recognised by the research community and regarded by many as de facto standards for batch learning.

There are several reasons why decision tree learners are highly regarded. They are fairly simple, the decision tree model itself being easy to comprehend. This high level of *interpretability* has several advantages. The decision process induced via the learning process is transparent, so it is apparent *how* the model works. Questions of *why* the model works can lead to greater understanding of a problem, or if the model manages to be successful without truly reflecting the real world, can highlight deficiencies in the data used.

The quest for accuracy means that interpretability alone will not guarantee widespread acceptance of a machine learning model. Perhaps the main reason decision trees are popular is that they are consistently accurate on a wide variety of problems. The classic decision tree systems recursively split the multi-dimensional data into smaller and smaller regions, using greedily chosen axis-orthogonal splits. This divide-and-conquer strategy is simple yet often successful at learning diverse concepts.

Another strong feature of decision trees is their efficiency. With n examples and m attributes, page 197 of [124] shows that the average cost of basic decision tree induction is $O(mn \log n)$, ignoring complexities such as numeric attributes and subtree raising. A more detailed study of tree induction complexity can be found in [94]. The cost of making a decision is $O(\text{tree depth})$ in the worst case, where typically the depth of a tree grows logarithmically with its size.

For the batch setting, recent studies [23] have shown that single decision trees are no longer the best off-the-shelf method. However, they are competitive when used as base models in ensemble methods. For this reason, ensemble methods employing Hoeffding trees are explored later in chapters 6 and 7.

3.1 The Hoeffding Bound for Tree Induction

Each internal node of a standard decision tree contains a test to divide the examples, sending examples down different paths depending on the values of particular attributes. The crucial decision needed to construct a decision tree is when to split a node, and with which example-discriminating test. If the tests used to divide examples are based on a single attribute value, as is typical in classic decision tree systems, then the set of possible tests is reduced to the number of attributes. So the problem is refined to one of deciding which attribute, if any, is the best to split on.

There exist popular and well established criteria for selecting decision tree split tests. Perhaps the most common is *information gain*, used by C4.5. Information gain measures the average amount of 'purity' that is gained in each subset of a split. The purity of the subsets is measured using *entropy*, which for a distribution of class labels consisting of fractions $p_1, p_2, ..., p_n$ summing to 1, is calculated thus:

$$\text{entropy}(p_1, p_2, ..., p_n) = \sum_{i=1}^{n} -p_i \log_2 p_i \tag{3.1}$$

Gain in information is measured by subtracting the weighted average entropy of the subsets of a split from the entropy of the class distribution before splitting. Entropy is a concept from information theory that measures the amount of information conveyed by a message in *bits*. Throughout this thesis the splitting criterion is assumed to be information gain, but this does not rule out other methods. As pointed out by Domingos and Hulten, other similar methods such as the Gini index used by CART can be just as equally applied.

The estimated information gain resulting from a split on each attribute is the heuristic used to guide split decisions. In the batch learning setting this decision is straightforward, as the attribute with the highest information gain over all of the available and applicable training data is the one used. How to make the same (or very similar) decision in the data stream setting is the innovation contributed by Domingos and Hulten. They employ the Hoeffding bound [67], otherwise known as an additive Chernoff bound.

The Hoeffding bound states that with probability $1 - \delta$, the true mean of a random variable of range R will not differ from the estimated mean after n independent observations by more than:

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}} \tag{3.2}$$

This bound is useful because it holds true regardless of the distribution generating the values, and depends only on the range of values, number of observations and desired confidence. A disadvantage of being so general is that it is more conservative than a distribution-dependent bound. An alternative bound has been suggested by Jin and Agrawal [77]. The Hoeffding bound formulation is well founded and works well empirically, so tighter bounds are not explored in this thesis.

For the purposes of deciding which attribute to split on, the random variable being estimated is the difference in information gain between splitting on the best and second best attributes. For example, if the difference in gain between the best two attributes is estimated to be 0.3, and ϵ is computed to be 0.1, then the bound guarantees that the maximum possible change in difference will be 0.1. From this the smallest possible difference between them in the future must be at least 0.2, which will always represent positive separation for the best attribute.

For information gain the range of values (R) is the base 2 logarithm of the number of possible class labels. With R and δ fixed, the only variable left to change the Hoeffding bound (ϵ) is the number of observations (n). As n increases, ϵ will decrease, in accordance with the estimated information gain getting ever closer to its true value.

A simple test allows the decision, with confidence $1-\delta$, that an attribute has superior information gain compared to others—when the difference in observed information gain is more than ϵ. This is the core principle for Hoeffding tree induction, leading to the following algorithm.

3.2 The Basic Algorithm

Algorithm 2 Hoeffding tree induction algorithm.

```
1:  Let $HT$ be a tree with a single leaf (the root)
2:  for all training examples do
3:    Sort example into leaf $l$ using $HT$
4:    Update sufficient statistics in $l$
5:    Increment $n_l$, the number of examples seen at $l$
6:    if $n_l \ mod \ n_{min} = 0$ and examples seen at $l$ not all of same class then
7:      Compute $\overline{G}_l(X_i)$ for each attribute
8:      Let $X_a$ be attribute with highest $\overline{G}_l$
9:      Let $X_b$ be attribute with second-highest $\overline{G}_l$
10:     Compute Hoeffding bound $\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n_l}}$
11:     if $X_a \neq X_\emptyset$ and $(\overline{G}_l(X_a) - \overline{G}_l(X_b) > \epsilon$ or $\epsilon < \tau)$ then
12:       Replace $l$ with an internal node that splits on $X_a$
13:       for all branches of the split do
14:         Add a new leaf with initialized sufficient statistics
15:       end for
16:     end if
17:   end if
18: end for
```

Algorithm 2 lists pseudo-code for inducing a Hoeffding tree from a data stream. Line 1 initializes the tree data structure, which starts out as a single root node. Lines 2-18 form a loop that is performed for every training example.

Every example is filtered down the tree to an appropriate leaf, depending on the tests present in the decision tree built to that point (line 3). This leaf is then updated (line 4)—each leaf in the tree holds the sufficient statistics needed to make decisions about further growth. The sufficient statistics that are updated are those that make it possible to estimate the information gain of splitting on each attribute. Exactly what makes up those statistics is discussed in Section 3.2.2. Line 5 simply points out that n_l is the example count at the leaf, and it too is updated. Technically n_l can be computed from the sufficient statistics.

For efficiency reasons the code block from lines 6-17 is only performed periodically, every n_{min} examples for a particular leaf, and only when necessary, when a mix of observed classes permits further splitting. The delayed evaluation controlled by n_{min} is discussed in Section 3.2.3.

Lines 7-11 perform the test described in the previous section, using the Hoeffding bound to decide when a particular attribute has won against all of the others. G is the splitting criterion function (information gain) and $\overline{G}$ is its estimated value. In line 11 the test for $X_\emptyset$, the null attribute, is used for pre-pruning (Section 3.2.4). The test involving τ is used for tie-breaking (Section 3.2.5).

If an attribute has been selected as the best choice, lines 12-15 split the node, causing the tree to grow. Preventing the tree from using too much memory is the topic of Section 3.3.

3.2.1 Split Confidence

The δ parameter used in the Hoeffding bound is one minus the desired probability that the correct attribute is chosen at every point in the tree. Because a high likelihood of correctness is desired, with probability close to one, this parameter is generally set to a small value. For the experiments described throughout the thesis, δ is set to the VFDT default of 10^{-7}.

Figure 3.1 shows a plot of the Hoeffding bound using the default parameters for a two-class problem ($R = \log_2(2) = 1$, $\delta = 10^{-7}$). The bound rapidly drops to below 0.1 within the first one thousand examples, and after ten thousand examples is less than 0.03. This means that after ten thousand examples the calculated information gain of an attribute needs to be 0.03 greater than the second best attribute for it to be declared the winner.

3.2.2 Sufficient Statistics

The statistics in a leaf need to be sufficient to enable calculation of the information gain afforded by each possible split. Efficient storage is important—it is costly to store information in the leaves, so storing unnecessary information would result in an increase in the total memory requirements of the tree.

For attributes with discrete values, the statistics required are simply counts of the class labels that apply for each attribute value. If an attribute has v unique attribute values and there are c possible classes, then this information can be stored in a table with vc entries. The node maintains a separate table

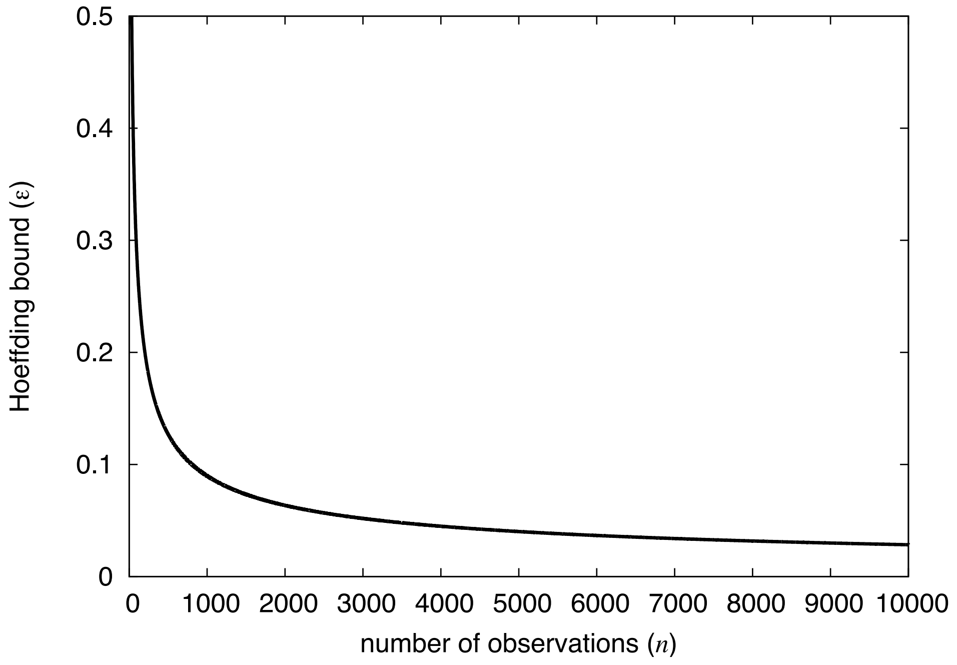

Figure 3.1: Hoeffding bound on a two-class problem with default parameters.

per discrete attribute. Updating the tables simply involves incrementing the appropriate entries according to the attribute values and class of the training example. Table 5.1 on page 94, used as an example when looking at prediction methods, shows what such tables can look like.

Continuous numeric attributes are more difficult to summarize. Chapter 4 is dedicated to this topic.

3.2.3 Grace Period

It is computationally costly to evaluate the information gain of attributes after each and every training example. Given that a single example will have little influence on the results of the calculation, it is sensible to wait for more examples before re-evaluating. The n_{min} parameter, or grace period, dictates how many examples since the last evaluation should be seen in a leaf before revisiting the decision.

This has the attractive effect of speeding up computation while not greatly harming accuracy. The majority of training time will be spent updating the sufficient statistics, a lightweight operation. Only a fraction of the time will splits be considered, a more costly procedure. The worst impact that this delay will have is a slow down of tree growth, as instead of splitting as soon as possible, a split will delayed by as much as $n_{min} - 1$ examples.

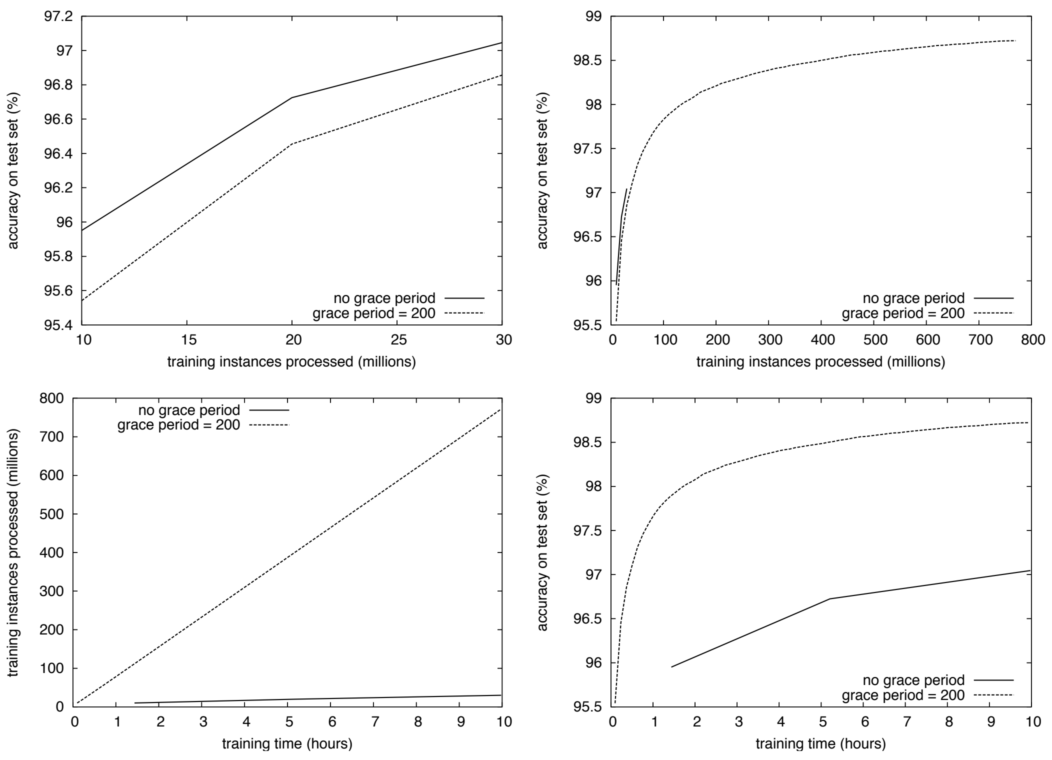

Figure 3.2: Effect of grace period on the RRBFC data with a 32MB memory limit.

For experimentation n_{min} is fixed at 200, the default setting found in the original paper [32]. Figure 3.2 shows the impact this setting has on the accuracy of trees induced from the RRBFC data. From the plot in the top-left it would appear that considering split decisions after every training example does improve the accuracy of the tree, at least within the first 30 million training examples shown on this data. From an accuracy per example perspective, the non grace period tree is superior. The plot to the top-right shows more of the picture, where each tree was allowed ten hours to grow, the tree that had n_{min} set to 200 was able to process almost 800 million examples and achieve significantly better accuracy within that time than the tree lacking a grace period. The plot to the bottom-left shows the total number of examples that were processed over that time—without a grace period 30 million examples in ten hours were possible, with a grace period approximately 25 times more examples were processed in the same time period. Viewing accuracy per time spent in the bottom-right plot makes the advantage of using a grace period clear.

3.2.4 Pre-pruning

It may turn out more beneficial to not split a node at all. The Hoeffding tree algorithm detects this case by also considering the merit of no split, represented by the null attribute $X_\emptyset$. A node is only allowed to split when an attribute looks sufficiently better than $X_\emptyset$, by the same Hoeffding bound test that determines differences between other attributes. This will suspend further growth below a leaf, until such time that another attribute looks better than the null one.

Pre-pruning in the stream setting is not a permanent decision as it is in batch learning. Nodes are prevented from splitting until it appears that a split will be useful, so in this sense, the memory management strategy of disabling nodes (Section 3.3) can also be viewed as a form of pre-pruning. Conventional knowledge about decision tree pruning is that pre-pruning can often be premature and is not as commonly used as post-pruning approaches. A reason for premature pre-pruning is lack of data, which is not a problem in abundant data streams. Another danger of pre-pruning is that helpful attribute interactions will be overlooked, since the trees do not look beyond the immediate decision.

All of the Hoeffding tree implementations used in experiments for this thesis had pre-pruning enabled, as suggested by Domingos and Hulten [32]. After the main experimentation was complete, further experiments were conducted with pre-pruning disabled. These experiments showed no noticeable difference in size, speed or accuracy of trees. This raises questions about the value of pre-pruning—it does not appear to be harmful and has theoretical merit, but in practice may not provide much benefit. This question remains open and is not further explored by this thesis.

In the batch learning setting it is very easy to induce a tree that perfectly fits the training data but does not generalize to new data. This problem is typically overcome by post-pruning the tree. On non-concept drifting data streams such as those described here, it is not as critical to prevent overfitting via post-pruning as it is in the batch setting. If concept drift were present, active and adaptive post-pruning could be used to adjust for changes in concept, a topic outside the scope of this thesis.

3.2.5 Tie-breaking

A situation can arise where two or more competing attributes cannot be separated. A pathological case of this would be if there were two attributes with

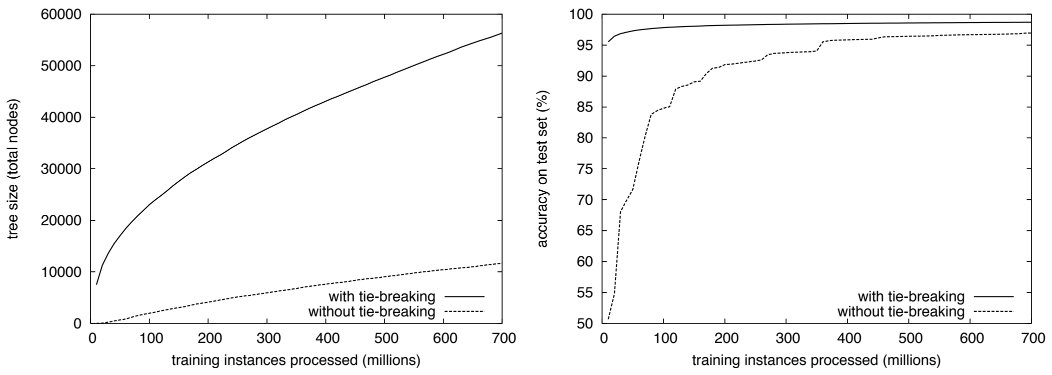

Figure 3.3: Effect of tie-breaking on the RRBFC data with a 32MB memory limit.

identical values. No matter how small the Hoeffding bound it would not be able to separate them, and tree growth would stall.

If competing attributes are equally good, and are superior to some of the other split options, then waiting too long to decide between them can do more harm than good to the accuracy of the tree. It should not make a difference which of the equal competitors is chosen. To alleviate this situation, Domingos and Hulten introduce a tie breaking parameter, τ. If the Hoeffding bound is sufficiently small, that is, less than τ, then the node is split on the current best attribute regardless of how close the next best option is.

The effect of this parameter can be viewed in a different way. Knowing the other variables used in the calculation of the Hoeffding bound, it is possible to compute an upper limit on the number of examples seen by a leaf before tie-breaking intervenes, forcing a split on the best observed attribute at that point. The only thing that can prevent tie-breaking is if the best option turns out to be not splitting at all, hence pre-pruning comes into effect.

τ is set to the literature default of 0.05 for the experiments. With this setting on a two-class problem, ties will be broken after 3,224 examples are observed. With n_{min} being set to 200, this will actually be delayed until 3,400 examples.

Tie-breaking can have a very significant effect on the accuracy of trees produced. An example is given in Figure 3.3, where without tie-breaking the tree grows much slower, ending up around five times smaller after 700 million training examples and taking much longer to come close to the same level of accuracy as the tie-breaking variant.

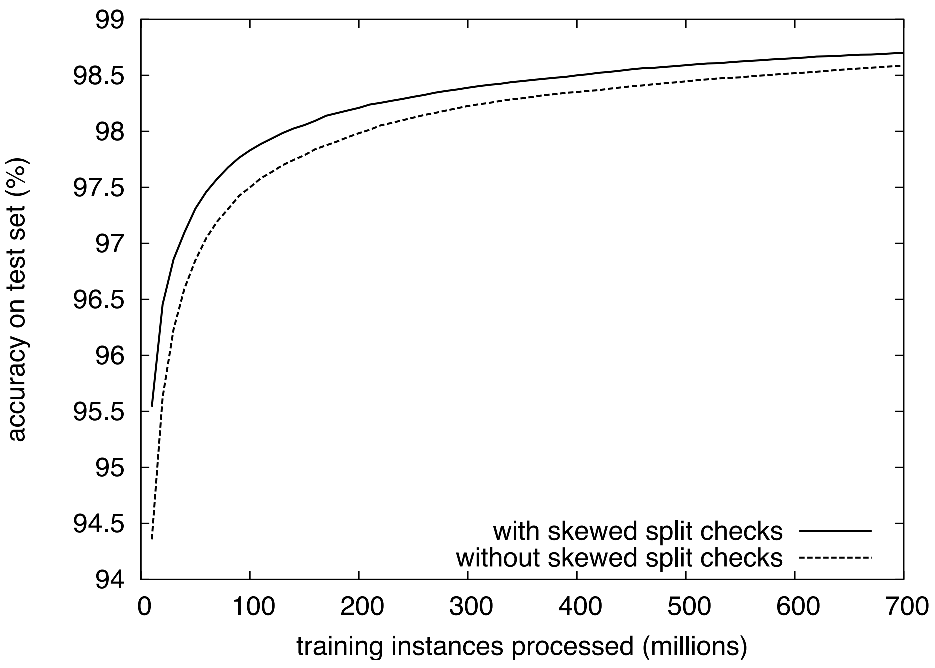

Figure 3.4: Effect of preventing skewed splits on the RRBFC data with a 32MB memory limit.

3.2.6 Skewed Split Prevention

There is another element present in the Hoeffding tree experimental implementation that is not mentioned in the pseudo-code. It is a small enhancement to split decisions that was first introduced by Gama et al. [52] and originally formulated for two-way numeric splits. In this thesis the concept has been generalized to apply to any split including splits with multiple branches.

The rule is that a split is only allowed if there are at least two branches where more than p_{min} of the total proportion of examples are estimated to follow the branch. The p_{min} threshold is an arbitrary parameter that can be adjusted, where a default value of 1% seems to work well enough. This prevents a highly skewed split from being chosen, where less than 1% of examples go down one path and over 99% of examples go down another. Such a split can potentially look attractive from an information gain perspective if it increases the purity of the subsets, but from a tree growth perspective is rather spurious.

In many cases this rule has no effect on tree classification accuracy, but Figure 3.4 shows it can have a positive effect in some cases.

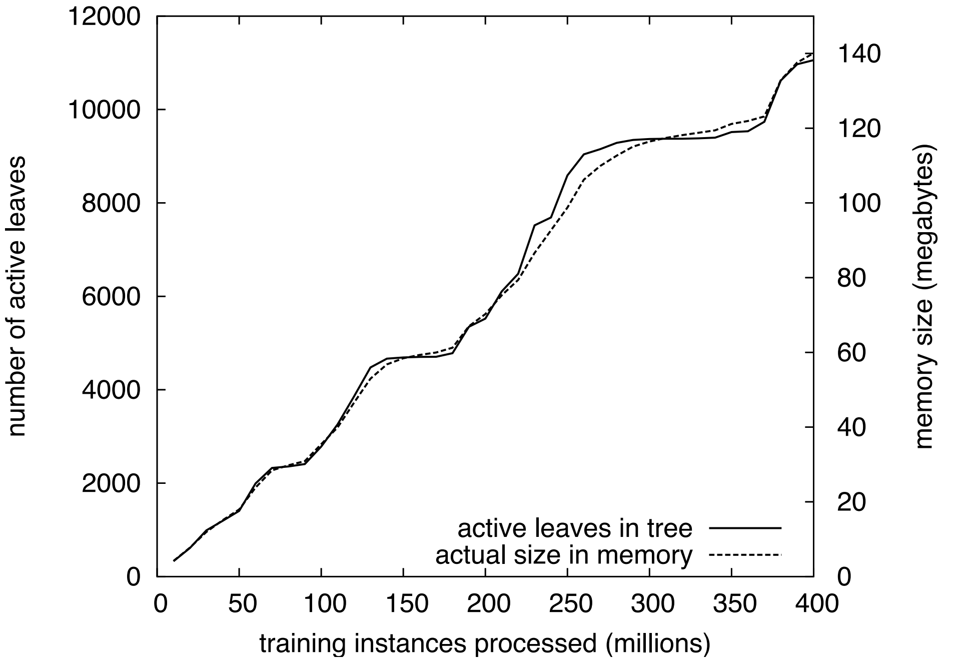

Figure 3.5: The size of an unbounded tree in memory is closely related to how many active leaves it has. This growth pattern occurs when learning a Hoeffding tree on the LED data.

3.3 Memory Management

The basic algorithm as described will continue to split the tree as needed, without regard for the ever-increasing memory requirements. This thesis argues that a core requirement of data stream algorithms is the ability to limit memory usage. To limit Hoeffding tree memory size, there must be a strategy that limits the total number of nodes in the tree. The node-limiting strategy used follows the same principles that Domingos and Hulten introduced for the VFDT system.

When looking at the memory requirements of a Hoeffding tree, the dominant cost is the storage of sufficient statistics in the leaves. Figure 3.5 is an example of how closely, in unbounded memory, the number of leaves in a tree can reflect its actual memory requirements. Section 3.4.1 describes in more detail how this relationship is exploited to efficiently estimate the actual memory size of the tree based on node counts.

The main idea behind the memory management strategy is that when faced with limited memory, some of the leaves can be *deactivated*, such that their sufficient statistics are discarded. Deciding which leaves to deactivate is based on a notion of how promising they look in terms of yielding accuracy gains for the tree.

In VFDT, the least promising nodes are defined to be the ones with the lowest values of $p_l e_l$, where p_l is the probability that examples will reach a particular leaf l, and e_l is the observed rate of error at l. Intuitively this makes sense, the leaves considered most promising for further splitting are those that see a high number of examples and also make a high number of mistakes. Such leaves will be the largest contributors to error in classification, so concentrating effort on splitting these nodes should see the largest reduction in error.

The implementation created for this thesis measures the 'promise' of leaves in an equivalent and straightforward way. Every leaf in the tree is capable of returning the full count of examples it has observed for each class since creation. The promise of a node is defined as the total remaining number of examples that have been seen to fall outside the currently observed majority class. Like VFDT's $p_l e_l$ measure, this estimates the potential that a node has for misclassifying examples. If n is the number of examples seen in a leaf, and E is the number of mistakes made by the leaf, and N is the total number of examples seen by the tree, then $p_l e_l = n/N \times E/n = E/N$. Promise is measured using E, which is equivalent to using E/N, as N is constant for all leaves.

To make memory management operate without introducing excessive runtime overhead, the size is estimated approximately whenever new nodes are introduced to the tree using the method described in Section 3.4.1. Periodically, a full and precise memory check is performed. This is done after every *mem-period* training examples are processed, where *mem-period* is a user defined constant. The periodic memory check calculates the actual memory consumption of the tree, a potentially costly process.

After checking memory usage, if there happen to be inactive nodes or if the maximum memory limit has been exceeded then a scan of the leaves is performed. The leaves are ordered from least promising to most promising, and a calculation is made based on the current sizes of the nodes to determine the maximum number of active nodes that can be supported. Once this threshold has been established, any active leaves found below the threshold are deactivated, and any inactive leaves above the threshold are reactivated. This process ensures that the tree actively and dynamically adjusts its focus on growing the most promising leaves first, while also keeping within specified memory bounds.

Figure 3.6 illustrates the process. The top part of the illustration shows

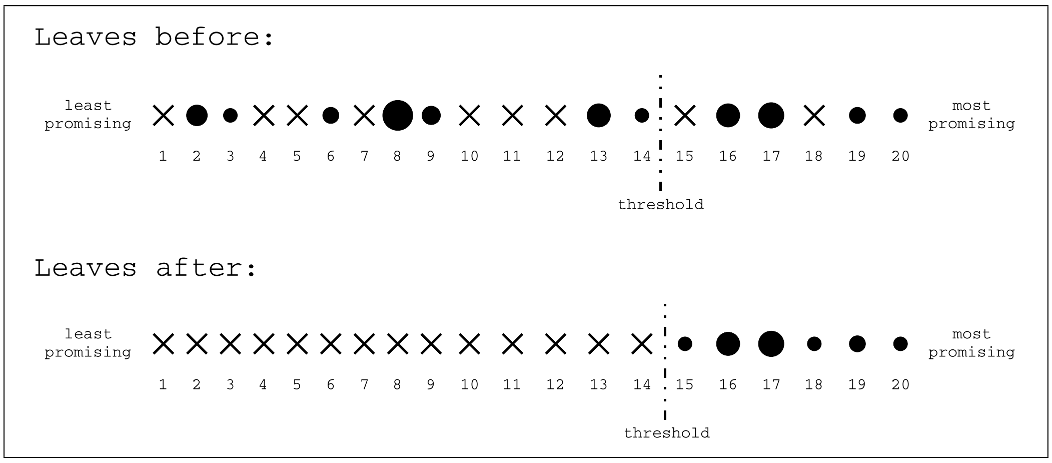

Figure 3.6: The memory management strategy employed after leaves of a tree have been sorted in order of *promise*. Dots represent *active* leaves which store sufficient statistics of various size, crosses represent *inactive* leaves which do not store sufficient statistics.

twenty leaf nodes from a hypothetical Hoeffding tree ordered from least to most promising. The size of active nodes in memory can differ due to the method used to track numeric attributes (Chapter 4) and when the sufficient statistics of poor attributes have been removed (Section 3.3.1). The sizes of the dots indicate the sizes of the active nodes, and inactive nodes are represented by crosses. Based on the average size of the active nodes and the total memory allowed, the threshold is determined in this case to allow a maximum of six active nodes. The bottom row shows the outcome after memory management is complete—below the threshold, the least promising nodes have all been deactivated, and above the threshold nodes 15 and 18 have been activated to make all of the most promising ones active.

For methods where the relative size between internal nodes, active leaves and inactive leaves is relatively constant this method is very effective at keeping the tree within a memory bound. For some methods where summary statistics in leaves can grow rapidly and vary substantially in size, such as the exhaustive binary tree numeric handling method (BINTREE) described in Chapter 4, it is less successful in maintaining a strict memory bound. In these cases, the tree's memory usage has a chance to creep beyond the limit in the period between memory checks, but will be brought back to the memory limit as soon as the next check is performed. Figure 3.7 demonstrates a case of memory 'creep' that occurred in the experiments—the target memory limit is 32 megabytes, and for efficiency the memory usage is only precisely measured every one hundred thousand examples, that is, *mem-period* = 100,000. In this case, the memory

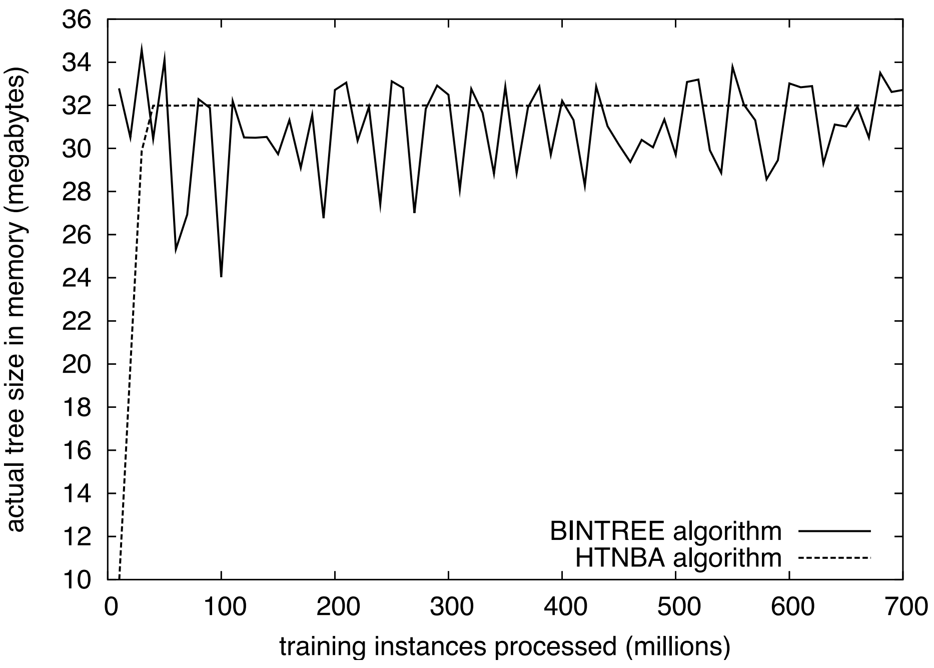

Figure 3.7: How closely two algorithms manage to obey a memory limit of 32 megabytes on the LED data.

used by BINTREE temporarily exceeds the memory bounds by as much as three megabytes or roughly 9%, but all the while fluctuating close to and more often further below the desired bound. The figure compares this with the memory requirements of the HTNBA variant on the same data, a much more stable method described in Chapter 5. The majority of the Hoeffding tree variants tested in the experiments exhibit stable memory behaviour, so most cases look like this, where the memory plot over time hits the target precisely before completely flattening out.

While the dominant storage cost is incurred for active nodes, limiting their number will eventually cause the size of internal nodes and inactive leaves to become significant. A point will be reached where no further growth of the tree can occur without the memory limit being exceeded. Once this stage is reached, all leaves of the tree will be made inactive, and the tree will no longer be able to grow.

One element of memory usage that has not yet been accounted for is the temporary working space needed to perform operations on the tree. Implemented in a modern computer language, update and prediction operations will make several function calls, and will store values and pointers in local memory, typically using some space on a working stack. This cost, which will

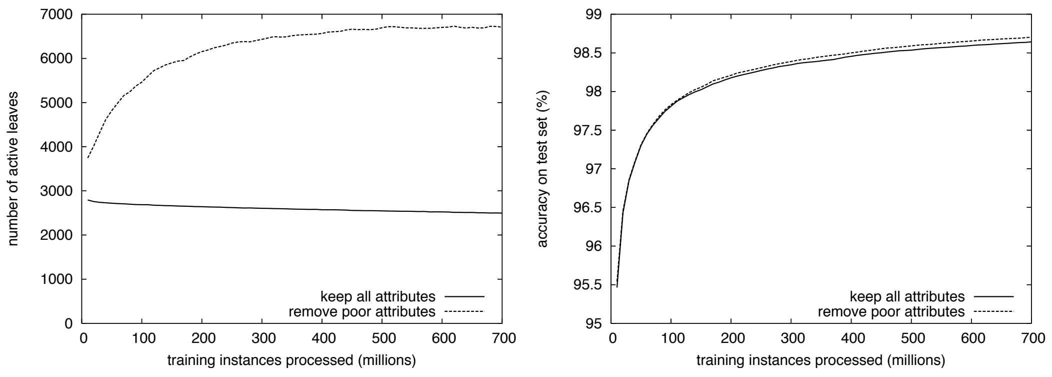

Figure 3.8: Effect of poor attribute removal on RRBFC data with a 32MB limit.

partly depend on implementation details, is assumed to be small and bounded such that it is insignificant compared to storage of the tree itself.

3.3.1 Poor Attribute Removal

An additional strategy for saving space was also suggested by Domingos and Hulten [32]. This strategy aims to reduce the size of the sufficient statistics in each leaf. The idea is to discard sufficient statistics for individual attributes when it looks very unlikely that they will be selected. When new leaves are formed, all attributes are considered as candidates for splitting. During every evaluation round that does not result in a decision to split, attributes are determined to be poor if their information gain is less than the gain of the current best attribute by more than the Hoeffding bound. According to the bound, such attributes are unlikely to be selected in that particular node of the tree, so the information tracking them in the leaf is discarded and they are ignored in split decisions from that point onward.

This strategy is not as powerful as full node deactivation, the best it can do is help to reduce the average size of leaf nodes. Theoretically this should benefit the accuracy of trees, because it will allow more leaves to remain active in limited memory. In practice the gains appear to be slight, as demonstrated in Figure 3.8, where on RRBFC the plot to the left shows a significant increase in the number of leaves allowed to remain active in a 32MB limit, while the plot to the right shows that this only translates to a small gain in accuracy. In this case the accuracy is measured when the tree makes predictions using standard majority class prediction. Removing poor attributes will affect the enhanced prediction methods described in Chapter 5, where this is discussed further.

3.4 Java Implementation Details

The data stream evaluation framework and all algorithms evaluated in this thesis were implemented in the Java programming language. The framework is named *MOA*, an acronym for **M**assive **O**nline **A**nalysis, and has evolved during the course of developing this thesis. MOA is related to *WEKA*[1], the **W**aikato **E**nvironment for **K**nowledge **A**nalysis [124], which is an award-winning[2] open-source workbench containing implementations of a wide range of batch machine learning methods. WEKA is also written in Java. The main benefits of Java are portability, where applications can be run on any platform with an appropriate Java virtual machine, and the strong and well-developed support libraries. Use of the language is widespread, and features such as the automatic garbage collection help to reduce programmer burden and error.

One of the key data structures used in MOA is the description of an example from a data stream. This structure borrows from WEKA, where an example is represented by an array of double precision floating point values. This provides freedom to store all necessary type of values—numeric attribute values can be stored directly, and discrete attribute values and class labels are represented by integer index values that are stored as floating point values in the array. Double precision floating point values require storage space of 64 bits, or 8 bytes. This detail can have implications for memory utilization.

A challenge in developing the system has been measuring the total sizes of objects in memory. Java deliberately hides details about how memory is allocated. This frees programmers from the burden of maintaining direct memory pointers that is otherwise typical in C programming, reducing dependence on a particular platform and eliminating a common source of error. The downside is that it makes the task of precisely measuring and limiting the memory usage of algorithms more difficult.

Early attempts at memory measurement revolved around exploiting Java's automatic *serialization* mechanism. It is easy to make all objects capable of being serialized, which means that they can be written out as a flat stream of bytes to be reconstructed later. The idea was to measure the size of the serialized stream of an object, which must be related to its true size in memory. The measuring process can be performed with minimum overhead by writing

[1]The moa and the weka are both birds native to New Zealand. The weka is a cheeky bird of similar size to a chicken. The moa was a large ostrich-like bird, an order of magnitude larger than a weka, that was hunted to extinction for its meat.

[2]Recipient of the 2005 SIGKDD Data Mining and Knowledge Discovery Service Award.

the object to a dummy stream that allocates no memory but instead simply counts the total number of bytes requested during write operations. It turns out that this method, while suitable for approximate relative measurement of object size, was not precise enough to achieve the level of control needed to confidently evaluate the algorithms. Often the true size of objects would be underestimated, so that even if the Java virtual machine was allocated a generous amount of memory it could still run into problems, strongly indicating that the serialized size estimates were inadequate.

Fortunately the release of Java 5 introduced a new mechanism allowing access to more accurate memory allocation measurements that are implementation specific. The *instrumentation* interface is harder to access as it requires extra work to be done by invoking the virtual machine with an *agent*, but once correctly set up can be queried for the size of a single object in memory. The size returned does not account for other sub-objects that are referenced, so it will not immediately return the total size of a complex data structure in memory, but this can be achieved by implementing additional code that uses *reflection* to traverse an entire data structure and compute the total size.

The Java code listing in Figure 3.9 tests the two size measuring methods. Five simple Java classes possessing an increasing number of fields are measured, along with the size of those objects when replicated 100 times in an array. Figure 3.10 displays the result of running the test in the same software environment as all of the experimental results reported in this thesis. The results show that the serialization method has a tendency to over-estimate the size of single small objects in memory, which would be expected due to the overhead that must be required to completely describe a serialized stream. Interestingly though, serialization also has a tendency to underestimate the size of a collection of objects, where for example the size of the `classA` array is estimated to be almost half of the instrumentation size. This behaviour explains the problems encountered when trying to rely on serialization measurements for experiments. The problem lies in hidden implementation details that make the serialization mechanism store information more efficiently than the virtual machine. The instrumentation measurements expose other effects that could not otherwise be predicted. There appears to be some form of *byte padding* effect present, where an object with a single integer field (4 bytes worth) requires the same space as one with two fields (16 bytes in both cases). The reason for this will be a technical decision on behalf of the virtual machine implementation, perhaps a byte alignment issue for the sake of efficiency. Whatever the

```
...
public static class ClassA implements Serializable {
   public int fieldA;
}
public static class ClassB implements Serializable {
   public int fieldA, fieldB;
}
public static class ClassC implements Serializable {
   public int fieldA, fieldB, fieldC;
}
public static class ClassD implements Serializable {
   public int fieldA, fieldB, fieldC, fieldD;
}
public static class ClassE implements Serializable {
   public int fieldA, fieldB, fieldC, fieldD, fieldE;
}
public static void main(String[] args) throws Exception {
   ClassA classAobject = new ClassA();
   ClassB classBobject = new ClassB();
   ClassC classCobject = new ClassC();
   ClassD classDobject = new ClassD();
   ClassE classEobject = new ClassE();
   ClassA[] classAarray = new ClassA[100];
   ClassB[] classBarray = new ClassB[100];
   ClassC[] classCarray = new ClassC[100];
   ClassD[] classDarray = new ClassD[100];
   ClassE[] classEarray = new ClassE[100];
   for (int i = 0; i < 100; i++) {
      classAarray[i] = new ClassA();
      classBarray[i] = new ClassB();
      classCarray[i] = new ClassC();
      classDarray[i] = new ClassD();
      classEarray[i] = new ClassE();
   }
   System.out.println("classAobject serialized size = "
                      + serializedByteSize(classAobject)
                      + " instrument size = "
                      + instrumentByteSize(classAobject));
   ...
```

Figure 3.9: Java code testing two methods for measuring object sizes in memory.

```
classAobject serialized size = 72 instrument size = 16
classBobject serialized size = 85 instrument size = 16
classCobject serialized size = 98 instrument size = 24
classDobject serialized size = 111 instrument size = 24
classEobject serialized size = 124 instrument size = 32
classAarray serialized size = 1124 instrument size = 2016
classBarray serialized size = 1533 instrument size = 2016
classCarray serialized size = 1942 instrument size = 2816
classDarray serialized size = 2351 instrument size = 2816
classEarray serialized size = 2760 instrument size = 3616
```

Figure 3.10: Output from running the code in Figure 3.9.

reason, this discovery serves to highlight the value of an accurate measurement mechanism, enabling the ability to account for such nuances that could not be anticipated otherwise.

3.4.1 Fast Size Estimates

For the Java implementation, the number of active and inactive nodes in a Hoeffding tree are used to estimate the total true size of the tree in memory. The node counts of a growing tree are easily maintained—whenever a new node is added an appropriate counter can be incremented, and when activating or deactivating leaves the counters are appropriately adjusted.

The size estimation procedure requires that actual measurements be performed every so often to establish and refine the parameters used for future estimation. The actual byte size in memory is measured (*trueSize*), along with the average byte size of individual active nodes (*activeSize*) and inactive nodes (*inactiveSize*). From this an extra parameter is calculated:

$$overhead = \frac{trueSize}{active \times activeSize + inactive \times inactiveSize} \qquad (3.3)$$

To increase the precision of the estimate the number of internal nodes in the tree, of similar size to inactive nodes, could also be included in the calculation. The implementation did not do this however, as the procedure described worked sufficiently well.

The estimated overhead is designed to account for the internal nodes of the tree, small inaccuracies in the node estimation procedure and any other structure associated with the tree that has otherwise not been accounted for, bringing the final estimate closer to the true value. Once these values are established, the actual byte size of the tree can be quickly estimated based solely on the number of active and inactive nodes in the tree:

$$size = (active \times activeSize + inactive \times inactiveSize) \times overhead \tag{3.4}$$

This calculation can be quickly performed whenever the number of inactive or active leaves changes, sparing the need to do a complete rescan and measurement of the tree after every change.

3.5 Summary

Representing one of the current best techniques for learning to classify examples in data streams, the basic algorithm for inducing decision trees from data streams via Hoeffding bounds has been described, and various parameter settings discussed. The challenge of memory management has been described, and some issues with the actual implementation in Java examined. Two important aspects of Hoeffding trees have not been covered, as they are studied in the following chapters. Creating decisions based on continuous numeric attributes is studied in Chapter 4. How the tree forms predictions is studied in Chapter 5.

Chapter 4

Numeric Attributes

The ability to learn from numeric attributes is very useful because many attributes needed to describe real-world problems are most naturally expressed by continuous numeric values. The decision tree learners C4.5 and CART successfully handle numeric attributes. Doing so is straightforward, because in the batch setting every numeric value is present in memory and available for inspection. A learner that is unable to deal with numeric attributes creates more burden for users. The data must first be pre-processed so that all numeric attributes are transformed into discrete attributes, a process referred to as *discretization*. Traditionally discretization requires an initial pass of the data prior to learning, which is undesirable for data streams.

The original Hoeffding tree algorithm demonstrated only how discrete attribute values could be handled. Domingos and Hulten [32] claim that the extension to numeric attributes:

> ...is immediate, following the usual method of allowing tests of the form "$(X_i < x_{ij})$?" and computing $\overline{G}$ for each allowed threshold x_{ij}.

While this statement is true, the practical implications warrant serious investigation. The storage of sufficient statistics needed to exactly determine every potential numeric threshold, and the result of splitting on each threshold, grows linearly with the number of unique numeric values. A high speed data stream potentially has an infinite number of numeric values, and it is possible that every value in the stream is unique. Essentially this means that the storage required to precisely track numeric attributes is unbounded and can grow rapidly.

For a Hoeffding tree learner to handle numeric attributes, it must track them in every leaf it intends to split. An effective memory management strategy will deactivate some leaves in favour of more promising ones when facing memory shortages, such as discussed in Section 3.3. This may reduce the impact of leaves with heavy storage requirements but may also significantly hinder growth. Instead it could be more beneficial to save space via some form of approximation of numeric distributions.

Section 4.1 looks at common methods used in batch learning. Several approaches for handling numeric attributes during Hoeffding tree induction have been suggested before, and are discussed in Section 4.2. Prior to this study the methods have not been compared, so Section 4.3 explores the tradeoff of accuracy versus size by empirical comparison.

4.1 Batch Setting Approaches

Strategies for handling continuous attribute values have been extensively studied in the batch setting. Some algorithms, for example *support vector machines* [22], naturally take continuous values as input due to the way they operate. Other algorithms are more naturally suited to discrete inputs, but have common techniques for accepting continuous values, Naive Bayes and C4.5 for example. It is useful in the batch setting to separate the numeric attribute problem entirely from the learning algorithm—a discretization algorithm can transform all inputs in the data to discrete values as a pre-processing step that is independent from the learning algorithm. This way, any learning algorithm that accepts discrete attributes can process data that originally contained continuous numeric attributes, by learning from a transformed version of the data. In fact, it has been claimed that in some cases, algorithms with built-in numeric handling abilities can improve when learning from pre-discretized data [34].

Methods for discretizing data in the batch setting are surveyed by Dougherty et al. [34]. They introduce three axes to categorize the various methods: *global* vs *local*, *supervised* vs *unsupervised* and *static* vs *dynamic*.

Methods that are *global* work over an entire set of examples, as opposed to *local* methods that work on smaller subsets of examples at a time. C4.5 is categorized as a *local* method, due to the example space being divided into smaller regions with every split of the tree, and discretization performed on increasingly smaller sets. Some methods can be either *global* or *local* depending on how they are applied, for example Dougherty et al. [34] categorize the k-

Table 4.1: Summary of batch discretization methods, categorized in four axes.

method	global/ local	supervised/ unsupervised	static/ dynamic	parametric/ non-parametric
equal width	*global*	*unsupervised*	*static*	*parametric*
equal frequency	*global*	*unsupervised*	*static*	*parametric*
k-means clustering	either	*unsupervised*	either	*parametric*
Fayyad & Irani	either	*supervised*	*static*	*non-parametric*
C4.5	*local*	*supervised*	*static*	*non-parametric*

means clustering method as *local*, while Gama and Pinto [51] say it is *global*.

Discretization that is *supervised* is influenced by the class labels of the examples, whereas *unsupervised* discretization is not. Methods that are *supervised* can exploit class information to improve the effectiveness of discretization for classification algorithms.

Dougherty et al. also believe that the distinction between *static* and *dynamic* methods is important (otherwise known as *uni-variate* and *multi-variate*), although they do not consider *dynamic* methods in their survey, which are much less common than *static* ones. A *static* discretization treats each attribute independently. A *dynamic* discretization considers dependencies between attributes, for example a method that optimizes a parameter by searching for the best setting over all attributes simultaneously.

Gama and Pinto [51] add a fourth useful distinction: *parametric* vs *non-parametric*. Methods considered *parametric* require extra parameters from the user to operate, and *non-parametric* methods use the data alone.

The following subsections detail several well-known approaches to batch discretization. Table 4.1 summarizes the properties of each. All methods are *static*, apart from k-means clustering which is also capable of *dynamic* discretization.

4.1.1 Equal Width

This *global unsupervised parametric* method divides the continuous numeric range into k bins of equal width. There is no overlap between bins, so that any given value will lie in exactly one bin. Once the boundaries of the bins are determined, which is possible knowing only the minimum and maximum values, a single scan of the data can count the frequency of observations for each bin. This is perhaps the simplest method of approximating a distribution of numeric values, but is highly susceptible to problems caused by skewed distributions and outliers. A single outlier has potential to influence the approximation, as an extreme value can force a distribution that is otherwise reasonably approximated into representation with many values in a single bin, where most of the remaining bins are empty.

4.1.2 Equal Frequency

This method is similar to equal width, it is also a *global unsupervised parametric* method that divides the range of values into k bins. The difference is that the bin boundaries are positioned so that the frequency of values in each bin is equal. With n values, the count in each bin should be n/k, or as close to this as possible if duplicate values and uneven divisions cause complications. Computing the placement of bins is more costly than the equal width method because a straightforward algorithm needs the values to be sorted first.

4.1.3 k-means Clustering

This method is *unsupervised*, *parametric*, and can be either *local* or *global*. It is based on the well-known k-means clustering algorithm [64]. The clustering algorithm can work on multi-dimensional data and aims to minimize the distance within clusters while maximizing the distance between clusters. With an arbitrary starting point of k centers, the algorithm proceeds iteratively by assigning each point to its nearest center based on Euclidean distance, then recomputing the central points of each cluster. This continues until convergence, where every cluster assignment has stabilized. When used for *static* discretization the clustering will be performed on a single attribute at a time. *Dynamic* discretization is possible by clustering several attributes simultaneously.

4.1.4 Fayyad and Irani

This method [37] is quite different from those described above as it is *supervised* and *non-parametric*. The algorithm is categorized [34] as capable of both *global* and *local* operation. The values of an attribute are first sorted, then a cut-point between every adjacent pair of values is evaluated, with n values and no repeated values this involves $n - 1$ evaluations. The counts of class labels to either side of each split candidate determine the *information gain* in the same way that attributes are chosen during tree induction (Section 3.1). The information gain has been found to be a good heuristic for dividing values while taking class labels into consideration. The cut-point with the highest gain is used to divide the range into two sets, and the procedure continues by recursively cutting both sets. Fayyad and Irani show [37] that this procedure will never choose a cut-point between consecutive examples of the same class, leading to an optimization of the algorithm that avoids evaluation of such points.

A stopping criterion is necessary to avoid dividing the range too finely, failure to terminate the algorithm could potentially continue division until there is a unique interval per value. If an interval is *pure*, that is, has values all of the same class, then there is no reason to continue splitting. If an interval has a mixture of labels, Fayyad and Irani apply the principle of *minimum description length* (MDL) [106] to estimate when dividing the numeric range ceases to provide any benefit.

4.1.5 C4.5

The procedure for discretizing numeric values in Quinlan's C4.5 [104] decision tree learner is *local*, *supervised* and *non-parametric*. It essentially uses the same procedure described in Fayyad and Irani's method, only it is not recursively applied in the same manner. For the purposes of inducing a decision tree, a single two-way split is chosen and evaluated for every numeric attribute. The cut-points are decided locally on the respective subset of every node, in the same way as described above—scanning through the values in order and calculating the information gain of candidate cut-points to determine the point with the highest gain. The difference is that the scan is carried out only once per numeric attribute to find a single split into two subsets, and no further recursion is performed on those subsets at that point. The recursive nature of decision tree induction means however that numeric ranges can be cut again further down the tree, but it all depends on which attributes are selected during tree growth. Splits on different attributes will affect the subsets of examples that are subsequently evaluated.

Responding to results in [34] showing that *global* pre-discretization of data using Fayyad and Irani's method could produce better C4.5 trees than using C4.5's *local* method, Quinlan improved the method in C4.5 release 8 [105] by removing some bias in the way that numeric cut-points were chosen.

4.2 Data Stream Approaches

The first thing to consider are ways in which the batch methods from the previous section could be applied to the data stream setting.

The equal width method is simple to perform in a single pass and limited memory provided that the range of values is known in advance. This requirement could easily violate the requirements of a data stream scenario because unless domain knowledge is provided by the user the only way to determine the true range is to do an initial pass of the data, turning the solution into a two-pass operation. This thesis only considers solutions that work in a single pass. Conceivably an adaptive single-pass version of the algorithm could be used, such as described by Gama and Pinto [51], where any values outside of the known range would trigger a reorganization of the bins. However, even if a single-pass solution were available, it would still be prone to the problem of outliers.

Equal frequency appears more difficult to apply to data streams than equal width because the most straightforward implementation requires the data to be sorted. The field of database optimization has studied methods for constructing equal frequency intervals, or equivalently *equi-depth histograms* or computation of quantiles, from a single pass. The literature related to this is surveyed in Section 4.2.3.

A method similar to k-means discretization for data streams would require a clustering algorithm that is capable of working on a stream. Data stream clustering is outside the scope of this thesis, but the problem has been worked on by several researchers such as Guha et al. [59, 58].

Fayyad and Irani's discretization algorithm and the similar method built into C4.5 require the data to be sorted in order to search for the best cut-point.

A rare example of a discretization method specifically intended to operate on data streams for machine learning purposes is presented by Gama and Pinto [51]. It works by maintaining two layers—the first layer simplifies the data with an equal width summary that is incrementally updated and the second layer builds the final histogram, either equal width or equal frequency, and is only updated as needed.

Methods that are *global* are applied as a separate pre-processing step before learning begins, while *local* methods are integrated into a learning algorithm, where discretization happens during learning as needed. Since only single-pass solutions to learning are considered, straightforward implementation of *global* discretization is not viable, as this would require an initial pass to discretize the data, to be followed by a second pass for learning. Unfortunately this discounts direct application of all *global* solutions looked at thus far, leaving few options apart from C4.5. The attractive flexibility afforded in the batch setting by separating discretization as a pre-processing step does not transfer to the demands of the data stream setting.

For Hoeffding tree induction the discretization can be integrated with learning by performing *local* discretization on the subsets of data found at active leaves, those in search of splits. The number of examples that contribute to growth decisions at any one time are limited, depending on the total number of active leaves in the tree. A brute-force approach stores every example in a leaf until a split decision is made. Without memory management the number of active leaves in the tree can grow without bound, making it impossible to use brute force without a suitable memory management scheme.

The methods discussed in the following subsections represent various proposals from the literature for handling numeric values during Hoeffding tree induction. At a higher level they are all trying to reproduce the C4.5 discretization method in the stream setting, so in this sense they are all *supervised* methods. The difference between them is that they approximate the C4.5 process in different ways. The *exhaustive binary tree* approach (Section 4.2.2) represents the brute-force approach of remembering all values, thus is a recreation of the batch technique. Awareness of space-efficiency as a critical concern in processing data streams has led to other methods applying a two-stage approach. Discretization methods at the first stage are used to reduce space costs, intended to capture the sorted distribution of class label frequencies. These are then used as input for the second stage, which makes a *supervised* C4.5-style two-way split decision.

In addition to the distinctions introduced earlier, a final dimension is introduced to help distinguish between the methods: *all-class* vs *per-class*. The *all-class* methods produce a single approximation of the distribution of examples, such as a single set of boundaries, recording the frequency of all classes over one approximation. In contrast, the *per-class* methods produce a different approximation per class, so for example each class is represented by an independent set of boundaries. The *per-class* methods are *supervised* in the sense that the class labels influence the amount of attention given to certain details—by allocating the same amount of space to the approximation of each class the *per-class* methods studied here enforce equal attention to each class.

Table 4.2: Summary of stream discretization methods. All methods are *local*, *static*, and involve two stages, the second of which is *supervised*.

	per-class	*all-class*
parametric	quantile summaries Gaussian approx. (2nd stage)	VFML
non-parametric	Gaussian approx. (1st stage)	exhaustive binary tree

The discretization methods for Hoeffding trees are discussed next, with properties summarized in Table 4.2.

4.2.1 VFML Implementation

Although Domingos and Hulten have not published any literature describing a method for handling numeric attributes, they have released working source code in the form of their VFML package [72]. VFML is written in C and includes an implementation of VFDT that is capable of learning from streams with numeric attributes.

This method is *all-class* and *parametric*, although the original implementation hides the single parameter. Numeric attribute values are summarized by a set of ordered bins, creating a histogram. The range of values covered by each bin is fixed at creation and does not change as more examples are seen. The hidden parameter is a limit on the total number of bins allowed—in the VFML implementation this is hard-coded to allow a maximum of one thousand bins. Initially, for every new unique numeric value seen, a new bin is created. Once the fixed number of bins have been allocated, each subsequent value in the stream updates the counter of the nearest bin.

There are two potential issues with the approach. The first is that the method is sensitive to data order. If the first one thousand examples seen in a stream happen to be skewed to one side of the total range of values, then the final summary will be incapable of accurately representing the full range of values.

The other issue is estimating the optimal number of bins. Too few bins will mean the summary is small but inaccurate, and many bins will increase accuracy at the cost of space. In the experimental comparison the maximum number of bins is varied to test this effect.

4.2.2 Exhaustive Binary Tree

This method represents the case of achieving perfect accuracy at the necessary expense of storage space. It is *non-parametric* and *all-class*. The decisions made are the same that a batch method would make, because essentially it is a batch method—no information is discarded other than the observed order of values.

Gama et al. present this method in their *VFDTc* system [52]. It works by incrementally constructing a binary tree structure as values are observed. The path a value follows down the tree depends on whether it is less than, equal to or greater than the value at a particular node in the tree. The values are implicitly sorted as the tree is constructed.

This structure saves space over remembering every value observed at a leaf when a value that has already been recorded reappears in the stream. In most cases a new node will be introduced to the tree. If a value is repeated the counter in the binary tree node responsible for tracking that value can be incremented. Even then, the overhead of the tree structure will mean that space can only be saved if there are many repeated values. If the number of unique values were limited, as is the case in some data sets, then the storage requirements will be less intensive. In all of the synthetic data sets used for this study the numeric values are generated randomly across a continuous range, so the chance of repeated values is almost zero.

The primary function of the tree structure is to save time. It lowers the computational cost of remembering every value seen, but does little to reduce the space complexity. The computational considerations are important, because a slow learner can be even less desirable than one that consumes a large amount of memory. The impact of the space cost is measured in the experimental comparison (Section 4.3).

Beside memory cost, this method has other potential issues. Because every value is remembered, every possible threshold is also tested when the information gain of split points is evaluated. This makes the evaluation process more costly than more approximate methods.

This method is also prone to data order issues. The layout of the tree is established as the values arrive, such that the value at the root of the tree will be the first value seen. There is no attempt to balance the tree, so data order is able to affect the efficiency of the tree. In the worst case, an ordered sequence of values will cause the binary tree algorithm to construct a list, which will lose all the computational benefits compared to a well balanced binary tree.

4.2.3 Quantile Summaries

The field of database research is also concerned with the problem of summarizing the numeric distribution of a large data set in a single pass and limited space. The ability to do so can help to optimize queries over massive databases [110].

Researchers in the field of database research are concerned with accuracy guarantees associated with quantile estimates, helping to improve the quality of query optimizations. Random sampling is often considered as a solution to this problem. Vitter [120] shows how to randomly sample from a data stream, but the non-deterministic nature of random sampling and the lack of accuracy guarantees motivate search for other solutions. Munro and Paterson [98] show how an exact quantile can be deterministically computed from a single scan of the data, but that this requires memory proportional to the number of elements in the data. Using less memory means that quantiles must be approximated. Early work in quantile approximation includes the P^2 algorithm proposed by Jain and Chlamtac [76], which tracks five markers and updates them as values are observed via piecewise fitting to a parabolic curve. The method does not provide guarantees on the accuracy of the estimates. Agrawal and Swami [3] propose a method that adaptively adjusts the boundaries of a histogram, but it too fails to provide strong accuracy guarantees. More recently, the method of Alsabti et al. [5] provides guaranteed error bounds, continued by Manku et al. [92] who demonstrate an improved method with tighter bounds.

The quantile estimation algorithm of Manku et al. [92] was the best known method until Greenwald and Khanna [56] proposed a *quantile summary* method with even stronger accuracy guarantees, thus representing the best current known solution. The method works by maintaining an ordered set of tuples, each of which records a value from the input stream, along with implicit bounds for the range of each value's true rank. An operation for compressing the quantile summary is defined, guaranteeing that the error of the summary is kept within a desired bound. The quantile summary is said to be ϵ-approximate, after seeing N elements of a sequence any quantile estimate returned will not differ from the exact value by more than ϵN. The worst-case space requirement is shown by the authors to be $O(\frac{1}{\epsilon}\log(\epsilon N))$, with empirical evidence showing it to be even better than this in practice.

Greenwald and Khanna mention two variants of the algorithm. The *adaptive* variant is the basic form of the algorithm, that allocates more space only as error is about to exceed the desired ϵ. The other form, used by this thesis, is referred to as the *pre-allocated* variant, which imposes a fixed limit on the amount of memory used. Both variants are *parametric*—for *adaptive* the parameter is ϵ, for *pre-allocated* the parameter is a tuple limit. The *pre-allocated* method was chosen because it guarantees stable approximation sizes throughout the tree, and is consistent with the majority of other methods by placing upper bounds on the memory used per leaf.

When used to select numeric split points in Hoeffding trees, a *per-class* approach is used where a separate quantile summary is maintained per class label. When evaluating split decisions, all values stored in the tuples are tested as potential split points. Different limits on the maximum number of tuples per summary are examined later in Section 4.3.

4.2.4 Gaussian Approximation

This method approximates a numeric distribution on a *per-class* basis in small constant space, using a *Gaussian* (commonly known as *normal*) distribution. Such a distribution can be incrementally maintained by storing only three numbers in memory, and is completely insensitive to data order. A Gaussian distribution is essentially defined by its mean value, which is the center of the distribution, and standard deviation or variance, which is the spread of the distribution. The shape of the distribution is a classic bell-shaped curve that is known by scientists and statisticians to be a good representation of certain types of natural phenomena, such as the weight distribution of a population of organisms.

Algorithm 3 describes a method for incrementally computing the mean and variance of a stream of values. It is a method that can be derived from standard statistics textbooks. The method only requires three numbers to be remembered, but is susceptible to rounding errors that are a well-known limitation of computer number representation.

A more robust method that is less prone to numerical error is given as Algorithm 4. It also requires only three values in memory, but maintains them in a way that is less vulnerable to rounding error. This method was derived from the work of Welford [123], and its advantages are studied in [24]. This is the method used in the experimental implementation.

Algorithm 3 Textbook incremental Gaussian.

```
$weightSum = 0$
$valueSum = 0$
$valueSqSum = 0$
**for all** data points ($value$, $weight$) **do**
  $weightSum = weightSum + weight$
  $valueSum = valueSum + value \times weight$
  $valueSqSum = valueSqSum + value \times value \times weight$
**end for**

**anytime output:**
**return** $mean = \frac{valuesSum}{weightSum}$
**return** $variance = \frac{valueSqSum - mean \times valueSum}{weightSum - 1}$
```

Algorithm 4 Numerically robust incremental Gaussian.

```
$weightSum = weight_{first}$
$mean = value_{first}$
$varianceSum = 0$
**for all** data points ($value$, $weight$) after first **do**
  $weightSum = weightSum + weight$
  $lastMean = mean$
  $mean = mean + \frac{value - lastMean}{weightSum}$
  $varianceSum = varianceSum + (value - lastMean) \times (value - mean)$
**end for**

**anytime output:**
**return** $mean = mean$
**return** $variance = \frac{varianceSum}{weightSum - 1}$
```

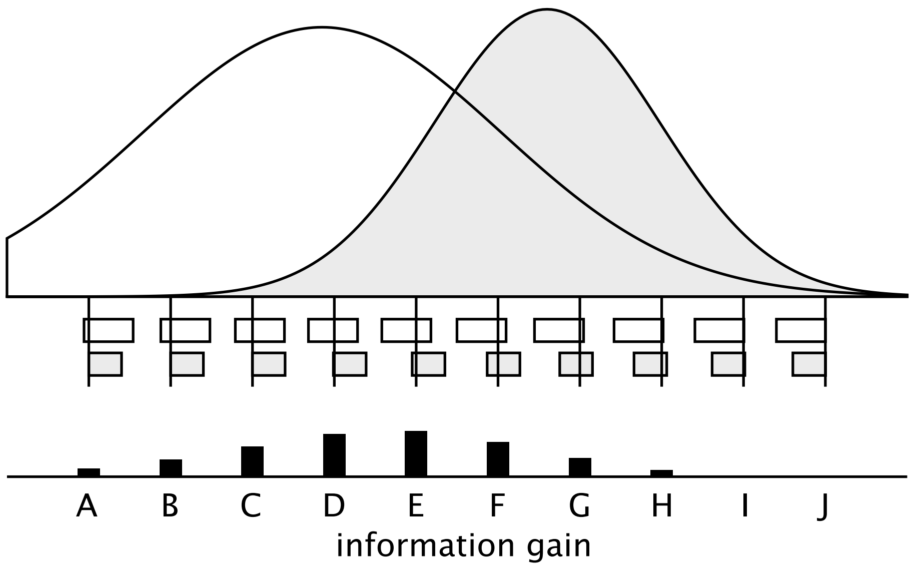

Figure 4.1: Gaussian approximation of 2 classes.

For each numeric attribute the numeric approximation procedure maintains a separate Gaussian distribution per class label. A method similar to this is described by Gama et al. in their UFFT system [50]. To handle more than two classes, the system builds a forest of trees, one tree for each possible pair of classes. When evaluating split points in that case, a single optimal point is computed as derived from the crossover point of two distributions. It is possible to extend the approach, however, to search for split points, allowing any number of classes to be handled by a single tree. The possible values are reduced to a set of points spread equally across the range, between the minimum and maximum values observed. The number of evaluation points is determined by a parameter, so the search for split points is *parametric*, even though the underlying Gaussian approximations are not. For each candidate point the weight of values to either side of the split can be approximated for each class, using their respective Gaussian curves, and the information gain is computed from these weights.

The process is illustrated in Figures 4.1-4.3. At the top of each figure are Gaussian curves, each curve approximates the distribution of values seen for a numeric attribute and labeled with a particular class. The curves can be described using three values; the mean value, the standard deviation or variance of values, and the total weight of examples. For example, in Figure 4.1 the class shown to the left has a lower mean, higher variance and higher ex-

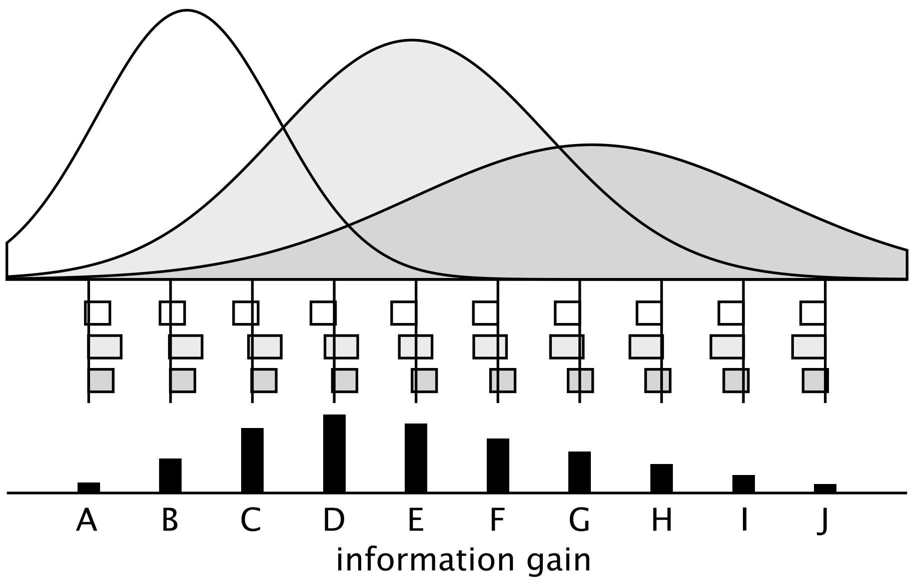

Figure 4.2: Gaussian approximation of 3 classes.

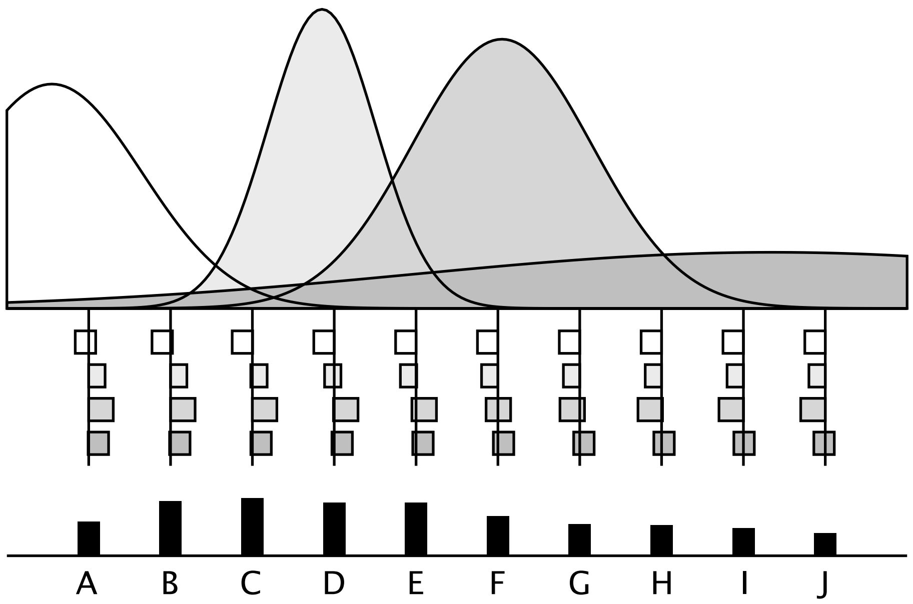

Figure 4.3: Gaussian approximation of 4 classes.

ample weight (larger area under the curve) than the other class. Below the curves the range of values has been divided into ten split points, labeled A to J. The horizontal bars show the proportion of values that are estimated to lie on either side of each split, and the vertical bar at the bottom displays the relative amount of information gain calculated for each split. For the two-class example (Figure 4.1), the split point that would be chosen as the best is point E, which according to the evaluation has the highest information gain. In the three-class example (Figure 4.2) the preferred split point is point D. In the four-class example (Figure 4.3) the split point C is chosen which nicely separates the first class from the others.

A refinement to this method, found to increase precision at low additional cost in early experiments, is used in the final implementation. It involves also tracking the minimum and maximum values of each class. This requires storing an extra two counts per class, but precisely maintaining these values is simple and fast. When evaluating split points the per-class minimum and maximum information is exploited to determine when class values lie completely to one side of a split, eliminating the small uncertainty otherwise present in the tails of the Gaussian curves. From the per-class minimum and maximum, the minimum and maximum of the entire range of values can be established, which helps to determine the position of split points to evaluate.

Intuitively it may seem that split points will only occur between the lowest and highest class mean, but this is not true. Consider searching for a split on the *age* attribute of the GENF1 data stream. The function is defined on page 38, where the first class has *age* values that are less than 40 and greater than 60, and the second class has *age* values between 40 and 60. Obviously either 40 or 60 are the optimal split points, but the means of both class distributions will lie somewhere between 40 and 60—the first class will have a large variance estimate, and the second will be much narrower. This motivates searching across the entire range of values for a split. Using the absolute minimum and maximum value makes the procedure susceptible to outliers similar to the weakness of equal width discretization. A more robust search may instead consider each mean plus or minus several standard deviations to determine the potential splitting range. This possibility is reserved for future work.

Simple Gaussian approximation will almost certainly not capture the full detail of an intricate numeric distribution, but is highly efficient in both computation and memory. Where the binary tree method uses extreme memory costs to be as accurate as possible, this method employs the opposite approach—using gross approximation to use as little memory as possible.

This simplified view of numeric distributions is not necessarily harmful to the accuracy of the trees it produces. There will be further opportunities to refine split decisions on a particular attribute by splitting again further down the tree. All methods have multiple attempts at finding values, where each subsequent attempt will be in a more focused range of values thus based on increasingly confident information. The more approximate Gaussian method relies on this more than the less approximate approaches. Also, the Gaussian approximation may prove more robust and resistant to noisy values than more complicated methods, which concentrate on finer details.

4.2.5 Numerical Interval Pruning

Another contribution is the work of Jin and Agrawal [77] who present an approach called *numerical interval pruning* (NIP). The authors claim it is an efficient method "which significantly reduces the processing time for numerical attributes, without loss in accuracy." Unfortunately insufficient details for reproducing the technique are provided in the paper. The numeric attribute range is divided into equal width intervals, and the intervals are then pruned if statistical tests determine that they are not likely to contain the best split point. Information is maintained in three ways—small class histograms, concise class histograms and detailed information (which can be in one of two formats, depending on which is most efficient). Without access to an actual implementation of their approach it is hard to be sure that any attempted reproduction of the technique, based on information provided in their paper, will be sufficiently faithful to the original. In their experimental evaluation, they found that the NIP method had more impact on computational cost than memory, in which they saw an average 39% reduction in runtime compared to an exhaustive search for split points. Based on these findings, it should be possible to relate NIP performance to that of the binary tree. Judging by the reports, both methods are highly accurate, but in terms of memory NIP would be more efficient than the binary tree by a small amount. The experimental results will show that even if a method is several times more space efficient than the exhaustive method it is still unlikely to compete with methods that use small and constant space per node.

4.3 Experimental Comparison of Methods

The numeric summarization methods could be evaluated several ways, for example, the amount of error in each approximation could be directly measured for several sample distributions, and the computational and memory costs compared. While such a study may be interesting, it does not necessarily give an overall impression of how the methods work inside Hoeffding tree induction. So instead of studying small individual cases, the methods are tested to see how well they perform in the final role of interest—how efficiently they help to produce Hoeffding trees and how accurate the resulting trees are.

The experiment is conducted as previously set out in Chapter 2. Note that the LED data set is omitted from this analysis because it does not contain numeric attributes. The tree induction algorithm has the following properties, with only the method for handling numeric attributes varied:

- split confidence $\delta = 10^{-7}$ (Section 3.2.1)
- grace period $n_{min} = 200$ (Section 3.2.3)
- pre-pruning *enabled* (Section 3.2.4)
- tie-breaking $\tau = 0.05$ (Section 3.2.5)
- skewed split prevention $p_{min} = 0.01$ (Section 3.2.6)
- memory managed with *mem-period*=10,000 for 100KB environment, and *mem-period*=100,000 for 32MB/400MB environments (Section 3.3)
- poor attribute removal *enabled* (Section 3.3.1)
- majority class prediction (Section 5.1)

These are the same basic settings used by HTMC in Chapter 5. The numeric method employed will also affect the behaviour of *Naive Bayes* enhanced predictions described in Chapter 5, which is avoided here by using majority class prediction to compare methods.

The numeric handling methods compared are listed in Table 4.3, including the memory limits imposed per numeric attribute per leaf, and with reference to the text explaining each method.

Table 4.4 lists the final results averaged over all 18 data sources, sorted by memory-limited environment. Detailed results per data source are available in Appendix A.1. The meaning of each column in the table is elaborated below:

Table 4.3: Final numeric methods compared.

name	description	memory limit	section
VFML10	VFML binning method	10 bins	4.2.1
VFML100	VFML binning method	100 bins	4.2.1
VFML1000	VFML binning method	1000 bins	4.2.1
BINTREE	exhaustive binary tree	none	4.2.2
GK100	Greenwald-Khanna quantile summary	100 tuples per class	4.2.3
GK1000	Greenwald-Khanna quantile summary	1000 tuples per class	4.2.3
GAUSS10	Gaussian approximation evaluating 10 split points	5 values per class	4.2.4
GAUSS100	Gaussian approximation evaluating 100 split points	5 values per class	4.2.4

accuracy the percentage of examples that the tree was able to correctly predict from the one million examples reserved for testing

training examples the total number of examples that were used to train the tree before evaluation was complete

active leaves the number of active leaves in the tree (those that are capable of further splitting)

inactive leaves the number of leaves that have been deactivated by the memory management scheme (these are no longer capable of splitting)

total nodes the total number of nodes in the tree, including internal decision nodes

tree depth the depth of the tree—the length of the longest path from the root to a leaf

training speed the speed that the tree was able to train, expressed as a percentage of the maximum speed that examples can be generated from the data source, as measured in Section 2.6

prediction speed the speed with which the tree could make predictions on the test data, again expressed as a percentage of maximum stream speed

The speeds achievable are quoted as percentages of the maximum speed that the streams can be generated by the experimental software/hardware. To complement the tables, Figures 4.5 and 4.6 display learning curves for most problems in the 32MB/handheld environment, which contains the most interesting results of the three environments. The results for GENF3 and GENF10 are omitted from the figures because these graphs are the least interesting, showing little visible separation between methods.

Reviewing the average accuracies, the four different approaches are easily ranked from best to worst. In all three memory environments, VFML10 is the most accurate on average over all data sources. The second most accurate method in every environment is GAUSS10. The GKx methods are generally third, and BINTREE is consistently the least accurate of the methods on average.

The default number of 1000 bins hard-coded in the original VFML implementation turns out to be the worst performer of the three VFML configurations tested. The trend is that smaller approximations, in this case smaller numbers of bins, that sacrifice accuracy for space per leaf, lead to more accurate trees overall. Requesting more space for numeric approximation reduces the numbers of active tree nodes that can reside in memory, slowing tree growth in a way that harms final tree accuracy.

The Gaussian method follows this trend, in that it is the smallest approximation tested, translating into the fastest tree growth and correspondingly accurate trees. Comparing the number of split evaluations tested, it is apparent that the finer grained exploration of GAUSS100 can be harmful. The GAUSS100 trees are on average much deeper than any of the other methods, suggesting that splits on certain numeric attributes are being repeated more often, because in many cases the tree depth exceeds the number of attributes available for splitting. These additional splits are probably very small and unnecessary refinements to previous split choices, and they may be very skewed. The skewed split parameter (Section 3.2.6) aims to reduce spurious splitting, but in this case the default setting is not enough to prevent poor choices. This is a symptom of trying to divide the range too finely based on approximation by a single smooth curve. The GAUSS10 method uses a suitably matched coarse division of only ten possibilities, which is far less susceptible to the same problem.

Comparing the quantile summary methods GK100 and GK1000, having 1000 tuples is helpful in the higher memory environments but harmful in 100KB of memory. Lower numbers of tuples can severely hinder the quantile summary method—a parameter setting of ten was tested but found to

Table 4.4: Final results averaged over all data sources comparing eight methods for handling numeric attributes.

method	accuracy (%)	training examples (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
100KB memory limit / sensor								
VFML10	87.70	25	0	8.29	10.8	11	70	83
VFML100	79.47	41	0	3.81	4.71	7	76	85
VFML1000	76.06	1	0	0.09	0.14	3	81	88
BINTREE	74.45	1	0	0.07	0.11	3	76	89
GK100	82.92	29	0	4.31	5.35	9	71	84
GK1000	74.65	1	0	0.08	0.13	3	59	88
GAUSS10	86.16	27	0	8.96	12.2	12	69	81
GAUSS100	85.33	28	0	8.33	11.9	20	64	79
32MB memory limit / handheld								
VFML10	91.53	901	31.8	674	1009	22	16	72
VFML100	90.97	941	5.98	480	703	24	17	73
VFML1000	90.97	952	4.28	412	604	27	17	73
BINTREE	90.48	835	3.67	373	541	22	15	73
GK100	89.96	962	6.88	531	777	34	17	73
GK1000	90.94	961	2.70	403	581	27	16	75
GAUSS10	91.35	892	94.8	683	1167	24	14	69
GAUSS100	90.91	853	92.6	639	1167	50	14	65
400MB memory limit / server								
VFML10	91.41	293	320	80.4	591	24	4	73
VFML100	91.19	142	73.9	143	316	23	4	74
VFML1000	91.12	108	19.0	127	206	22	3	78
BINTREE	90.50	60	13.7	92.9	147	19	2	80
GK100	89.88	158	84.0	145	346	32	4	75
GK1000	91.03	91	17.6	122	197	21	3	79
GAUSS10	91.21	518	540	26.8	891	28	6	73
GAUSS100	90.75	538	566	38.7	998	63	6	66

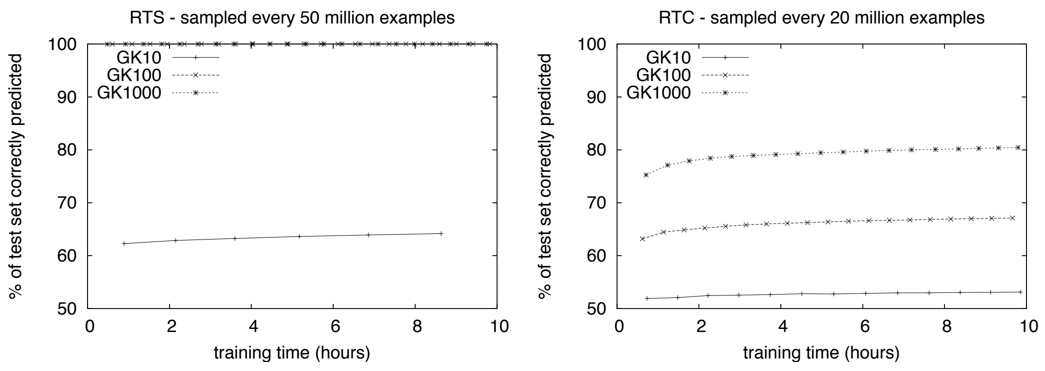

Figure 4.4: Examples of poor accuracy achieved by GK10 in 32MB.

be much worse than any other method, so is omitted from the final results. Figure 4.4 shows some examples of how much worse the ten-tuple summary can perform. In particular, the graph on RTS shows other settings getting very close to 100% accuracy in contrast to the ten-tuple variant achieving less than 65%. Like GAUSS100, GK10 suffers from symptoms of excessively deep trees which strongly indicates poor numeric split decisions. Larger quantile summaries perform well but the tradeoff between space and accuracy is not as effective as for the GAUSSx and VFMLx methods. The performance of GK1000 is similar to BINTREE in several situations, suggesting that it is highly accurate, while at the same time it manages to build larger trees, suggesting that it is more space efficient than BINTREE.

The poor performance of BINTREE shows that in limited memory situations, striving for perfect accuracy at the local level can result in lower accuracy globally. The problem is most pronounced in the 100KB sensor network environment, where tree growth for every data source was halted before the first evaluation took place, some time before one million training examples. Similar behaviour is evident in the other two most memory-intensive methods VFML1000 and GK1000, but BINTREE has the highest memory requirements of all, thus suffers the most in tree growth and accuracy. The method is simply too memory-hungry to support reasonable tree induction in this environment. In the other environments it fares better, but is not as successful on average as the more approximate methods.

Table 4.5 compares the individual final accuracies of the best two methods, VFML10 and GAUSS10. Bold figures indicate a better result, in this case both methods win 20 times each. GAUSS10 loses to VFML10 by a wide margin on RTCN in 400MB, although on this data set some of the other methods are not much better than GAUSS10 and some are worse still. Some of the worst losses

Table 4.5: VFML10 vs GAUSS10 accuracy (%).

method→	VFML10			GAUSS10		
	memory limit			memory limit		
dataset	100KB	32MB	400MB	100KB	32MB	400MB
RTS	96.49	99.99	99.98	**96.95**	99.99	**99.99**
RTSN	**75.80**	**78.54**	**78.53**	75.20	78.48	78.45
RTC	61.37	**83.58**	**83.87**	**62.49**	83.00	83.02
RTCN	53.63	**64.95**	**66.06**	53.63	62.45	61.87
RRBFS	87.69	93.13	92.43	**88.56**	**93.27**	**92.93**
RRBFC	87.84	98.61	97.41	**91.36**	**98.72**	**98.21**
WAVE21	80.80	84.20	83.50	**81.21**	**84.37**	**84.01**
WAVE40	80.28	84.00	83.31	**81.20**	**84.21**	**83.80**
GENF1	95.07	95.07	95.07	95.07	95.07	95.07
GENF2	**93.94**	**94.10**	**94.10**	78.46	94.03	94.00
GENF3	**97.52**	97.52	97.52	97.50	97.52	97.52
GENF4	**94.46**	94.67	**94.66**	93.68	94.67	94.65
GENF5	**92.45**	**92.89**	**92.84**	71.73	92.36	92.15
GENF6	89.70	**93.35**	93.28	**91.89**	93.31	93.28
GENF7	96.41	**96.82**	96.79	**96.51**	96.81	96.79
GENF8	99.40	99.42	99.42	**99.41**	99.42	99.42
GENF9	95.80	**96.81**	96.72	**96.07**	96.78	**96.74**
GENF10	**99.89**	99.89	99.89	99.88	99.89	99.89
average	87.70	91.53	91.41	86.16	91.35	91.21

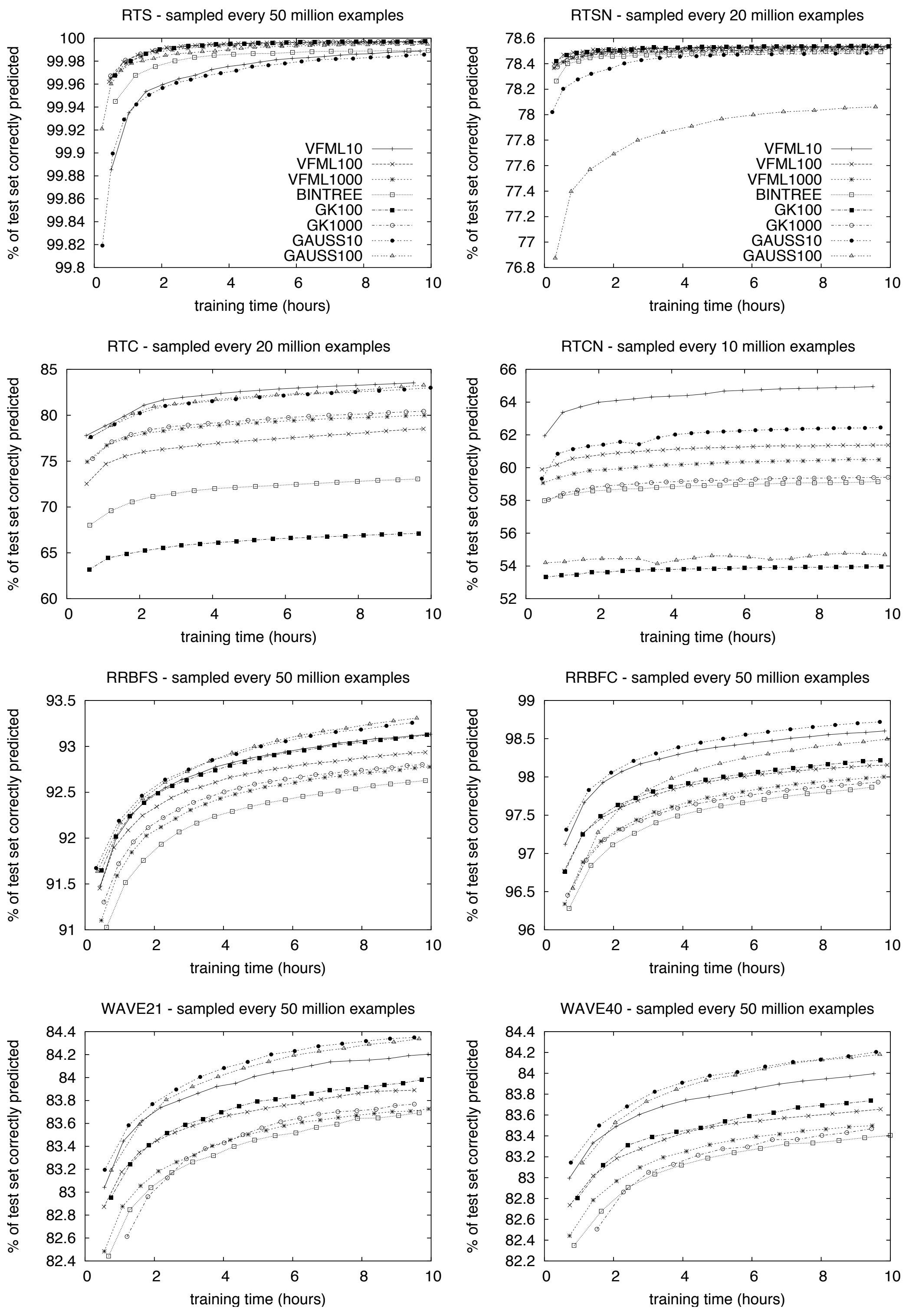

Figure 4.5: Part 1 of learning curves for numeric methods in 32MB memory limit.

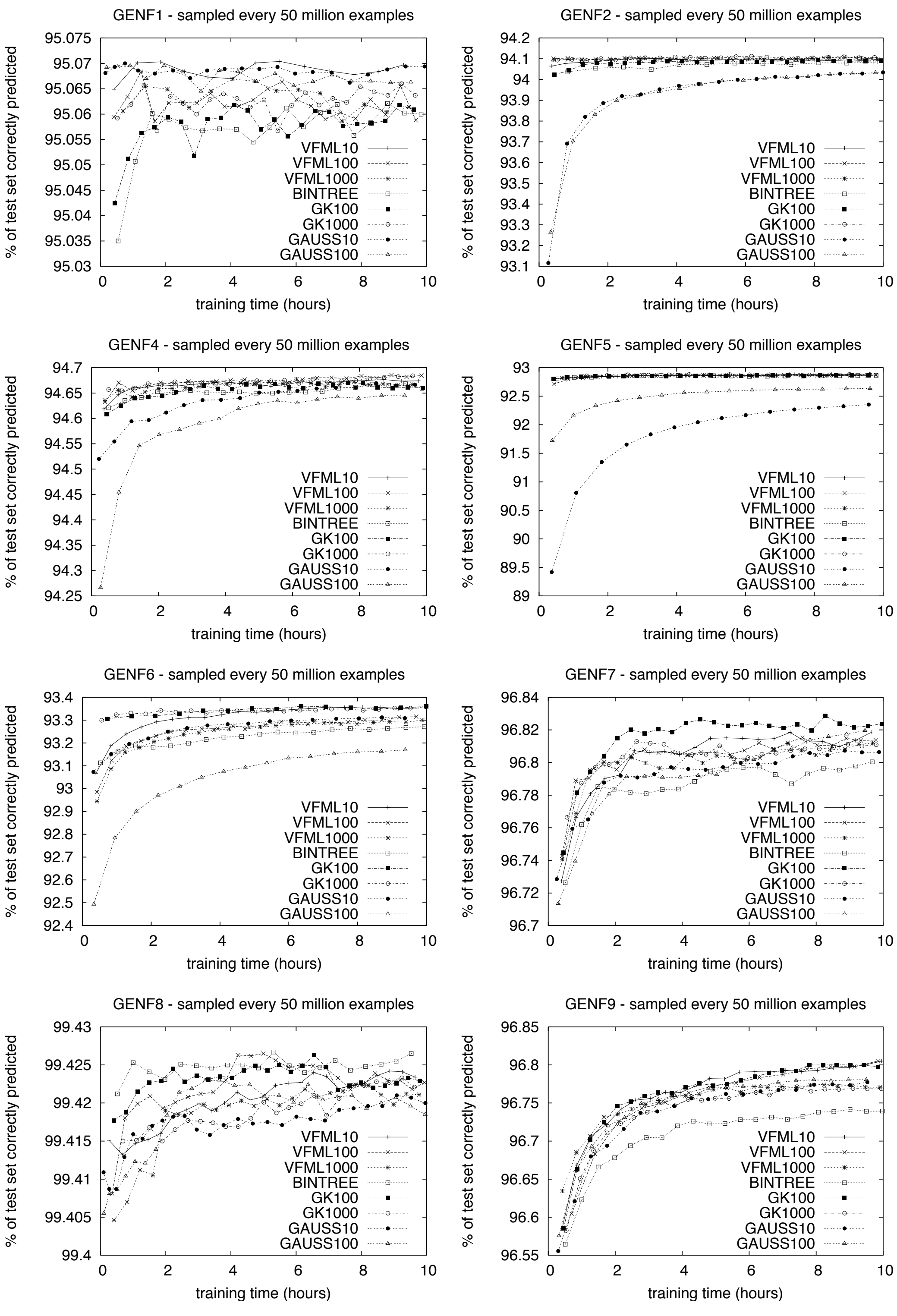

Figure 4.6: Part 2 of learning curves for numeric methods in 32MB memory limit.

for GAUSS10 occur on GENF2 and GENF5 in 100KB, where it is outperformed by all other methods. Referring back to the functions generating the underlying concept of these data streams (Figure 2.4, page 38), these functions are very similar. The function of GENF2 relies on the two numeric attributes *salary* and *age*, and GENF5 adds another layer of complexity by including dependency on a third numeric attribute, *loan*. From an inspection of the learning curves at the top right of Figure 4.6 it is clear that the Gaussian methods struggle with these concepts more than any of the other methods. The trees induced by the Gaussian method were inspected to find the cause of the problem. In this case the trees make the mistake of choosing a discrete attribute with many possible values that is completely irrelevant, *car*. After making this mistake the example space is highly segmented, so a lot of extra effort is required to correct the mistake further down the tree. The Gaussian methods slowly recover to come within reasonably close accuracy, besides the 100KB environment where the differences are exaggerated due to lack of space limiting opportunity to recover. This demonstrates a limitation of the Gaussian method, where the high level of approximation causes the best attributes to be underrated, although the true underlying cause of the issue is unknown, perhaps relating to an unintentional bias towards certain split types that could potentially be corrected in similar style to Quinlan's correction in [105]. This is reserved for future work.

Conversely, there are situations where the high level of approximation gives the Gaussian method an advantage over all others. The clearest cases of this are on the data sources RRBFS, RRBFC, WAVE21 and WAVE40. Such a bias may not be surprising when the generators responsible for these streams use numeric values drawn from random Gaussian distributions.

Analysing space complexity, the amount of memory required per leaf to track n numeric attributes and c classes is $10n + 10nc$ for VFML10 and $5nc$ for GAUSS10. For VFML10 the $10n$ term accounts for storage of the boundary positions, while the $10nc$ term accounts for the frequency counts. This simplified analysis underestimates the true cost of the VFML implementation, which also retains information about the class and frequency of values that lie exactly on the lower boundary of each bin, increasing the precision of decisions. For GAUSS10 the multiplying constant is five values per attribute and class because there are three values tracking the Gaussian curve and and additional two numbers tracking the minimum and maximum values. Clearly GAUSS10 requires the least memory.

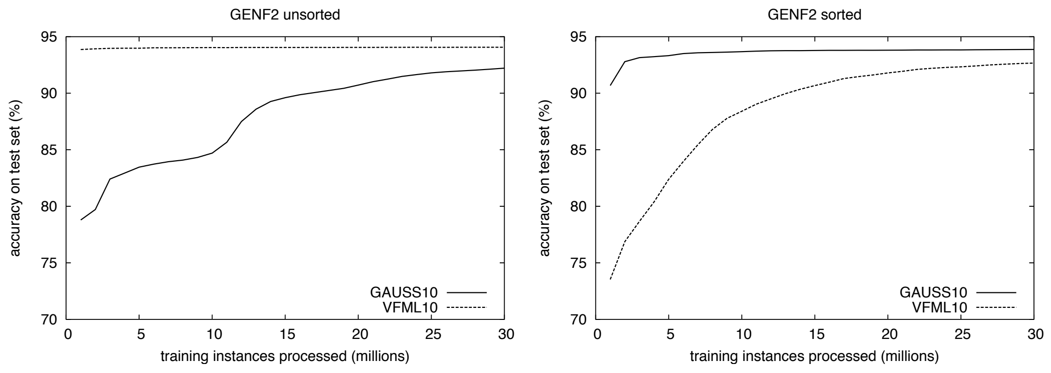

Figure 4.7: Effect that example ordering has on learning accuracy in 32MB on the GENF2 data. Left hand side: default random order. Right hand side: modified stream where every consecutive sequence of one million training examples has been sorted on the value of the *salary* attribute.

In theory, at the local level VFML10 should be very sensitive to data order, whereas GAUSS10 should not be sensitive at all. Whether this can translate into poorer global decisions during tree induction is not tested by the benchmark generators because all examples are randomly drawn somewhat uniformly from the space of possible examples. The right hand side of Figure 4.7 shows a constructed example where data order has been manipulated to expose VFML10's weakness. GENF2 has been modified so that every sequence of one million examples drawn from the stream has been sorted by the *salary* attribute. In this case the accuracy of GAUSS10 has improved while the early accuracy of VFML10 has dropped markedly. The ability of VFML10 to slowly recover may be partly due to additional tree structure increasing the dispersion of examples down different paths of the tree, reducing the degree to which values encountered at leaves are sorted.

The GAUSS10 method is highly competitive on most data sets besides being outperformed by VFML10 in a few cases. It uses less memory than VFML10 and is less susceptible to data order. For these reasons GAUSS10 is used as the default numeric handling method for the remainder of the thesis, used in the HTMC method of Section 5.4. Even though HTMC and GAUSS10 refer to the same algorithm they are kept separate for comparison purposes because the numeric experiments exclude the LED data source, causing the average of reported accuracies to differ.

On average, GAUSS10 trees reach much larger sizes than the other numeric methods in the same time and space, with many more active leaves. The lack of information during local decisions is made up for by increased tree structure, leading to trees that are more accurate overall. The general conclusion of this study of numeric handling techniques is that the most accurate methods for data streams are those that use very little space, but make up for loss of local accuracy by enabling tree growth to be much more productive.

4.4 Summary

This chapter studied the difficult challenge of managing continuous numeric attributes in data streams for the purposes of inducing decision trees. Five main approaches from the literature were discussed, and eight final configurations of algorithm were tested, ranging from perfectly accurate and memory intensive to highly approximate and lightweight. In experimental comparison, the most lightweight methods produced the most accurate trees, by virtue of being the lowest impediment to tree growth. The smallest approximation of all, the GAUSS10 method, was selected as the default numeric handling technique for the rest of the thesis. It is based on simple Gaussian approximation, similar to the method suggested by Gama et al. [50] but selects split points in a way that accommodates multiple class labels.

Before this investigation, VFML1000 was considered the default numeric handling strategy, as it is the strategy used by the public implementation of VFDT [72]. Averaged over all data sets and environments, and including accuracy on LED for consistency with the studies that follow, the accuracy of VFML1000 is 85.41%. The accuracy of GAUSS10, which will be referred to as HTMC in the next chapter, is 88.75%. The relative average improvement in accuracy, gained by selecting a more efficient numeric handling method, is 3.91%. This comes with an overall average training speed reduction of 11.88%, and an average prediction speed reduction of 6.75%.

Most of the accuracy gains were in the 100KB environment with a relative gain of 12.59%. In this environment training and prediction speed reduced by 13.75% and 7.95% respectively. In 32MB the relative accuracy gain was 0.41%, with training and prediction speed reducing by 17.65% and 4.17%. In 400MB where more expensive methods were able to compete better, the relative accuracy gain was only 0.10% on average. The training speed actually increased by half in this environment, while the prediction speed reduced by 7.79%.

Chapter 5

Prediction Strategies

The previous chapters have considered the induction of Hoeffding trees. Chapter 3 covered the basic induction of Hoeffding trees, followed by Chapter 4 which investigated the handling of continuous numeric attributes in the training data. This chapter focuses on the use of models once they have been induced—how predictions are made by the trees. Section 5.1 describes the standard *majority class* prediction method. Attempts to outperform this method are described in Section 5.2 and 5.3. The chapter concludes with an experiment in Section 5.4 to determine which method is best in practice.

5.1 Majority Class

Prediction using decision trees is straightforward. Examples with unknown label are filtered down the tree to a leaf, and the most likely class label retrieved from the leaf. An obvious and straightforward way of assigning labels to leaves is to determine the most frequent class of examples that were observed there during training. This method is used by the batch methods C4.5/CART and is naturally replicated in the stream setting by Hoeffding trees. If the likelihood of all class labels is desired, an immediate extension is to return a probability distribution of class labels, once again based on the distribution of classes observed from the training examples.

Table 5.1 is used to illustrate different prediction schemes throughout the chapter. In the case of majority class, the leaf will always predict class C_2 for every example, because most examples seen before have been of that class. There have been 60 examples of class C_2 versus 40 examples of class C_1, so the leaf will estimate for examples with unknown class that the probability of

Table 5.1: Example sufficient statistics in a leaf after 100 examples have been seen. There are two class labels: C_1 has been seen 40 times, and C_2 has been seen 60 times. There are three attributes: A_1 can either have value A or B, A_2 can be C, D or E, and A_3 can be F, G, H or I. The values in the table track how many times each of the values have been seen for each class label.

A_1		A_2			A_3					
A	B	C	D	E	F	G	H	I	class	total
12	28	5	10	25	13	9	3	15	C_1	40
34	26	21	8	31	11	21	19	9	C_2	60

class C_2 is 0.6, and the probability of C_1 is 0.4.

5.2 Naive Bayes Leaves

There is more information available during prediction in the leaves of a Hoeffding tree than is considered using majority class classification. The attribute values determine the path of each example down the tree, but once the appropriate leaf has been established it is possible to use the same attribute values to further refine the classification. Gama et al. call this enhancement *functional tree leaves* [50, 52].

If $P(C)$ is the probability of event C occurring, and $P(C|X)$ is the probability of event C given that X occurs, then from Bayes' theorem:

$$P(C|X) = \frac{P(X|C)P(C)}{P(X)} \tag{5.1}$$

This rule is the foundation of the Naive Bayes classifier [87]. The classifier is called 'naive' because it assumes independence of the attributes. The independence assumption means that for each example the value of any attribute will not have a bearing on the value of any other attribute. It is not realistic to expect such simplistic attribute relationships to be common in practice, but despite this the classifier works surprisingly well in general [33, 66].

By collecting the probabilities of each attribute value with respect to the class label from training examples, the probability of the class for unlabeled examples can be computed. Fortunately, the sufficient statistics being maintained in leaves of a Hoeffding tree for the purpose of choosing split tests are also the statistics required to perform Naive Bayes classification.

Returning to the example in Table 5.1, if an example being classified by the leaf has attribute values A_1=B, A_2=E and A_3=I then the likelihood of the class labels is calculated using Equation 5.1:

$$\begin{aligned}
P(C_1|X) &= \frac{P(X|C_1)P(C_1)}{P(X)} \\
&= \frac{[P(B|C_1) \times P(E|C_1) \times P(I|C_1)] \times P(C_1)}{P(X)} \\
&= \frac{\frac{28}{40} \times \frac{25}{40} \times \frac{15}{40} \times \frac{40}{100}}{P(X)} \\
&= \frac{0.065625}{P(X)}
\end{aligned}$$

$$\begin{aligned}
P(C_2|X) &= \frac{P(X|C_2)P(C_2)}{P(X)} \\
&= \frac{[P(B|C_2) \times P(E|C_2) \times P(I|C_2)] \times P(C_2)}{P(X)} \\
&= \frac{\frac{26}{60} \times \frac{31}{60} \times \frac{9}{60} \times \frac{60}{100}}{P(X)} \\
&= \frac{0.02015}{P(X)}
\end{aligned}$$

Normalizing the likelihoods means that the common $P(X)$ denominator is eliminated to reach the final probabilities:

$$\text{probability of } C_1 = \frac{0.065625}{0.065625 + 0.02015} = 0.77$$

$$\text{probability of } C_2 = \frac{0.02015}{0.065625 + 0.02015} = 0.23$$

So in this case the Naive Bayes prediction chooses class C_1, contrary to the majority class.

A technicality omitted from the example is the *zero frequency* problem that occurs if one of the counts in the table is zero. The Naive Bayes calculation cannot be performed with a probability of zero, so the final implementation overcomes this by using a *Laplace* estimator of 1. This adjustment, based on *Laplace's Law of Succession*, means for example that the class prior probabilities in the example above are instead treated as $\frac{41}{102}$ and $\frac{61}{102}$.

In batch learning the combination of decision trees and Naive Bayes classification has been explored by Kohavi in his work on *NBTrees* [83]. Kohavi's NBTrees are induced by specifically choosing tests that give an accuracy advantage to Naive Bayes leaves. In that setting he found that the hybrid trees could often outperform both single decision trees and single Naive Bayes models. He noted that the method performs well on large data sets, although the meaning of large in the batch setting can differ greatly from the modern stream context—most of the training sets he tested had less than 1,000 examples, with the largest set having under 50,000 examples.

Use of Naive Bayes models in Hoeffding tree induction has implications for the memory management strategy. Firstly, the act of deactivating leaf nodes is more significant, because throwing away the sufficient statistics will also eliminate a leaf's ability to make a Naive Bayes prediction. The heuristic used to select the most promising nodes does not take this into account, as it does not consider the possibility that a leaf may be capable of yielding better accuracy than majority class. For simplicity and consistency in the experimental implementation, the memory management strategy is not changed when Naive Bayes leaves are enabled. This makes sense if the use of Naive Bayes leaves are considered as a prediction-time enhancement to Hoeffding trees. Otherwise changes to memory management behaviour intended to better suit Naive Bayes prediction will significantly impact on overall tree induction, making it harder to interpret direct comparisons with majority class trees.

The outcome of this approach is that when memory gets tight and fewer of the leaves are allowed to remain active, then fewer of the leaves will be capable of Naive Bayes prediction. The fewer the active leaves, the closer the tree will behave to a tree that uses majority class only. By the time the tree is frozen and can no longer afford to hold any active leaves in memory then the tree will have completely reverted to majority class prediction. This behaviour is noted when looking at the experimental results.

The other issue is the secondary memory management strategy of removing poor attributes (Section 3.3.1). This too will alter the effectiveness of Naive Bayes models, because removing information about attributes removes some of the power that the Naive Bayes models use to predict, regardless of whether the attributes are deemed a poor candidate for splitting or not. As the removal strategy has been shown to not have a large bearing on final tree accuracy, whenever Naive Bayes leaves are employed the attribute removal strategy is not used.

Memory management issues aside, the Naive Bayes enhancement adds no cost to the induction of a Hoeffding tree, neither to the training speed nor memory usage. All of the extra work is done at prediction time. The amount of prediction-time overhead is quantified in the experimental comparison.

Early experimental work confirmed that Naive Bayes predictions are capable of increasing accuracy as Gama et al. observed [50, 52], but also exposed cases where Naive Bayes prediction fares worse than majority class. The first response to this problem was to suspect that some of the leaf models are immature. In the early stages of a leaf's development the probabilities estimated may be unreliable because they are based on very few observations. If that is the case then there are two possible remedies; either (1) give them a jump-start to make them more reliable from the beginning, or (2) wait until the models are more reliable before using them.

Previous work [68] has covered several attempts at option (1) of 'priming' new leaves with better information. One such attempt, suggested by Gama et al. [50] is to remember a limited number of the most recent examples from the stream, for the purposes of training new leaves as soon as they are created. A problem with this idea is that the number of examples able to be retrieved that apply to a particular leaf will diminish as the tree gets progressively deeper. Other attempts at priming involved trying to inject more of the information known prior to splitting into the resulting leaves of the split. Neither of these attempts were successful at overcoming the problem so are omitted from consideration.

The experimental results in Section 5.4 test two variations of option (2), waiting before trusting Naive Bayes models. The first is very simple—a fixed minimum number of examples are required at a leaf before Naive Bayes prediction is employed. The higher the threshold, the longer the tree will wait and the more often majority class prediction will be used. Setting it too high will not see any change from exclusive use of majority class, and setting it too low will permit premature use of Naive Bayes. The threshold used in the presented results is one thousand examples, which is not overly successful at solving the problem but was found in preliminary experimentation to be the best compromise.

It appears that a single fixed threshold is simply not sufficient to overcome the problem, motivating further exploration of methods. This led to the development and contribution of a second more sophisticated waiting strategy discussed next.

5.3 Adaptive Hybrid

Cases where Naive Bayes decisions are less accurate than majority class are a concern because more effort is being put in to improve predictions and instead the opposite occurs. In those cases it is better to use the standard majority class method, making it harder to recommend the use of Naive Bayes leaf predictions in all situations.

The method described here tries to make the use of Naive Bayes models more reliable, by only trusting them on a per-leaf basis when there is evidence that there is a true gain to be made. The *adaptive* method works by monitoring the error rate of majority class and Naive Bayes decisions in every leaf, and choosing to employ Naive Bayes decisions only where they have been more accurate in past cases. Unlike pure Naive Bayes prediction, this process *does* introduce an overhead during training. Extra time is spent per training example generating both prediction types and updating error estimates, and extra space per leaf is required for storing the estimates.

Algorithm 5 Adaptive prediction algorithm.

```
for all training examples do
    Sort example into leaf $l$ using $HT$
    if $majorityClass_l \neq$ true class of example then
        increment $mcError_l$
    end if
    if $NaiveBayesPrediction_l$(example) $\neq$ true class of example then
        increment $nbError_l$
    end if
    Update sufficient statistics in $l$
    ...
end for

for all examples requiring label prediction do
    Sort example into leaf $l$ using $HT$
    if $nbError_l < mcError_l$ then
        return $NaiveBayesPrediction_l$(example)
    else
        return $majorityClass_l$
    end if
end for
```

The pseudo-code listed in Algorithm 5 makes the process explicit. During training, once an example is filtered to a leaf but before the leaf is updated, both majority class prediction and Naive Bayes prediction are performed and both are compared with the true class of the example. Counters are incremented in the leaf to reflect how many errors the respective methods have made. At prediction time, a leaf will use majority class prediction unless the counters suggest that Naive Bayes prediction has made fewer errors, in which case Naive Bayes prediction is used instead.

In terms of the example in Table 5.1, the class predicted for new examples will depend on extra information. If previous training examples were more accurately classified by majority class than Naive Bayes then class C_2 will be returned, otherwise the attribute values will aid in Naive Bayes prediction as described in the previous subsection.

The accuracy gains afforded by this method and the extra costs involved are empirically quantified next.

5.4 Experimental Comparison of Methods

This section uses the testing framework established in Chapter 2 to compare four strategies for Hoeffding tree prediction. To ease reference to these methods in the text, each has been assigned a short name—the methods are called HTMC, HTNB, HTNB1K and HTNBA. Several elements to Hoeffding tree induction have been covered previously, so the following list summarizes the final properties of each method including references to explanatory text:

1. HTMC algorithm, properties:

   - split confidence $\delta = 10^{-7}$ (Section 3.2.1)
   - grace period $n_{min} = 200$ (Section 3.2.3)
   - pre-pruning *enabled* (Section 3.2.4)
   - tie-breaking $\tau = 0.05$ (Section 3.2.5)
   - skewed split prevention $p_{min} = 0.01$ (Section 3.2.6)
   - memory managed with *mem-period*=10,000 for 100KB environment, and *mem-period*=100,000 for 32MB/400MB environments (Section 3.3)
   - poor attribute removal *enabled* (Section 3.3.1)
   - numeric attributes handled with *Gaussian approximation* using 10 split evaluations (Section 4.2.4)
   - majority class prediction (Section 5.1)

2. HTNB algorithm, properties that differ from HTMC:
   - poor attribute removal *disabled* (Section 3.3.1 and 5.2)
   - Naive Bayes prediction (Section 5.2)
3. HTNB1K algorithm, properties that differ from HTNB:
   - majority class prediction in each leaf until 1000 examples seen, then Naive Bayes prediction afterwards (Section 5.2)
4. HTNBA algorithm, properties that differ from HTNB:
   - adaptive hybrid majority class/Naive Bayes prediction decided per leaf (Section 5.3)

HTMC is a re-implementation that for the most part is equivalent to the VFDT system, and as such can be considered the base Hoeffding tree method. Following the findings from the previous chapter, numeric attributes are handled using a Gaussian approximation that evaluates ten split points. In the current comparison the modifications tested involve changing the prediction strategies used by the tree, with HTMC representing the default majority class method.

Table 5.2 summarizes behaviour of the four methods for each of the three environments. The numbers have been averaged over all data sources. For a more detailed breakdown of the results per data source refer to the tables in Appendix A.2.

Recall from Chapter 2 that each method is allowed a total of ten hours training time. The results reported in the tables represent the final result recorded when ten hours of training were complete or earlier if the tree became frozen. Excessive evaluation overhead was avoided by only measuring and recording the properties of the trees after every ten million training examples in the 32MB/400MB environments, and more frequently in the 100KB environment where changes happen more rapidly, every one million examples.

First, looking at the properties of the trees besides accuracy, it is clear that the 100KB sensor environment strongly limits what the algorithms can achieve. In this environment fewer training examples are processed than possible in higher memory environments. This happened on every data set, and was caused by tree growth halting after all leaves have been deactivated well before the ten hour training limit. In fact, the training times in 100KB did not

Table 5.2: Final results averaged over all data sources comparing four methods for Hoeffding tree prediction.

method	accuracy (%)	training examples (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
100KB memory limit / sensor								
HTMC	85.51	27	0	8.64	11.9	12	69	81
HTNB	85.48	27	0	8.69	11.9	12	68	81
HTNB1K	85.48	27	0	8.69	11.9	12	68	81
HTNBA	85.44	29	0	8.65	11.9	12	67	82
32MB memory limit / handheld								
HTMC	90.44	902	92.3	659	1134	24	14	69
HTNB	90.48	825	75.1	643	1063	24	14	63
HTNB1K	90.48	905	72.5	691	1136	24	14	63
HTNBA	90.51	871	73.4	670	1106	24	14	65
400MB memory limit / server								
HTMC	90.30	525	522	25.4	864	28	6	71
HTNB	90.24	464	471	49.1	802	28	6	40
HTNB1K	90.34	450	494	49.1	847	28	6	42
HTNBA	90.70	463	489	46.7	828	28	6	53

exceed 30 minutes in any run. Because the final trees have been stripped of their active nodes they are effectively only capable of majority class prediction. This explains why the prediction speeds attained by the final trees hardly differ between prediction methods in this environment. One positive effect that the highly constrained memory limit has compared to the other environments is that it allows much higher training speeds to be attained, but this provides little consolation when only limited training is possible.

Looking at environments with higher memory, differences between the four methods begin to show and are the most pronounced in the server environment. It is interesting to see that the server environment is not capable of processing as many examples in the ten hour period as the 32MB handheld environment, neither is it able to grow as many nodes. This can be explained by looking at the extra amount of work involved in maintaining the trees in the largest memory environment. The trees are deeper and have many more active leaves to evaluate, slowing computation and limiting the number of examples that can be handled in a given time.

The average training speed is not significantly affected by the prediction method utilized, which is to be expected in the first three methods as they do not alter the amount of work performed during training, but this is a very positive sign for HTNBA which does in fact do some extra computation per training example. An explanation for this is that the extra processing is integrated with the induction process. The overhead of computing the local prediction accuracy is small when the appropriate data structures are already being updated.

The average prediction speeds of the trees seem to be related to their reliance on Naive Bayes leaves, a result that is understandable given that more computation is involved in making a Naive Bayes prediction than simply returning the majority class in a leaf. For this reason, HTNB and HTNB1K are the slowest at prediction because they respectively use Naive Bayes exclusively and almost exclusively, besides the 100KB case which as already explained is incapable of Naive Bayes at the final point. HTNBA lies between HTMC and the other two in terms of prediction speed, because it uses a mix of both prediction methods.

With regard to accuracy, based on the average results it appears that the prediction enhancements have little merit in the 100KB environment. Enabling Naive Bayes leaves in various forms has actually caused a decline in accuracy overall. The trees are quickly starved of memory and forced to revert fully to

Table 5.3: Modified HTNBA accuracy compared to HTMC, where HTNBA growth stops as soon as memory is full in 100KB, retaining all Naive Bayes models at the expense of tree size.

dataset	HTMC	fully active HTNBA
RTS	**96.95**	80.77
RTSN	**75.20**	70.11
RTC	**62.49**	58.41
RTCN	53.63	**54.34**
RRBFS	**88.56**	83.26
RRBFC	**91.36**	73.76
LED	73.94	**73.99**
WAVE21	81.21	**83.08**
WAVE40	81.20	**83.39**
GENF1	**95.07**	94.80
GENF2	**78.46**	74.84
GENF3	97.50	**97.52**
GENF4	**93.68**	89.80
GENF5	**71.73**	71.03
GENF6	**91.89**	90.75
GENF7	**96.51**	96.42
GENF8	**99.41**	99.40
GENF9	96.07	**96.08**
GENF10	**99.88**	99.87
average	85.51	82.72

majority class prediction. It is possible that Naive Bayes leaves could provide an advantage prior to deactivation.

To investigate this further, an experiment was conducted to test what would happen if Naive Bayes models are never deactivated. The only way to achieve this in limited memory is to stop growing the tree as soon as memory is full. As a result the trees end up being significantly smaller (in terms of average numbers of nodes, measured at over 24 times smaller), but the models in the leaves can continue to learn and refine their statistics with more examples. Each run was allowed to train for an hour, as experiments with this version of HTNBA showed that any benefit of additional learning after growth had ceased would level out very early, well before an hour of training was complete. Table 5.3 shows the resulting accuracy, which is on average worse than HTMC and also worse than the standard memory-managed HTNBA. Figures in bold represent superior accuracy on a particular data set. There are a few examples where a much smaller but Naive Bayes enhanced tree is more accurate than a

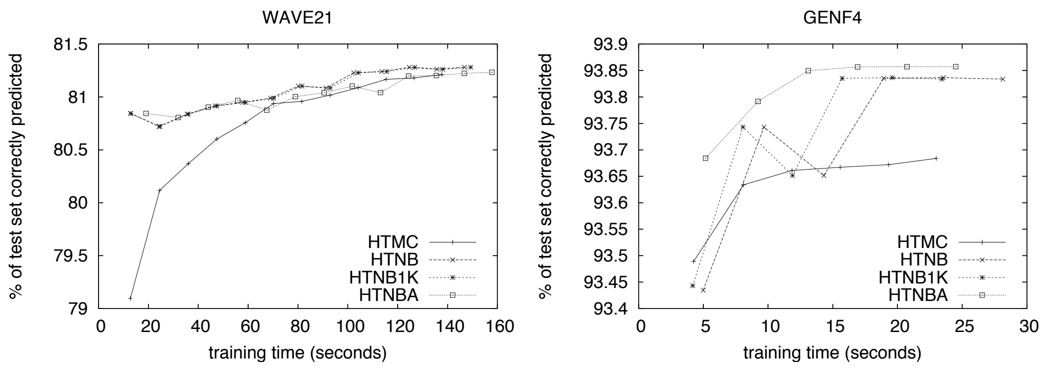

Figure 5.1: Two exceptional cases where Naive Bayes leaves perform better than majority class prediction in 100KB of memory.

larger tree relying on majority class prediction. It is not surprising that LED is one of those cases, as a single Naive Bayes model is capable of solving this particular problem very well. This and other cases demonstrate that more powerful leaf predictions can sometimes provide more benefit than additional tree structure. However, the cases where Naive Bayes models do not compensate for tree structure are more numerous, and some of the differences are very large.

In the main set of results where Naive Bayes nodes are being deactivated to allow further tree expansion, it looks as though the memory limit is too severe to see much evidence of an accuracy advantage from the Naive Bayes models prior to their deactivation. Figure 5.1 shows two cases against the trend where there are hints of this happening. On WAVE21 the Naive Bayes methods reach reasonable accuracy levels earlier than HTMC, but they all converge by the time the trees are frozen. GENF4 is a rare case where HTNBA actually looks best throughout in 100KB of memory, although the differences are only fractions of a percent. The fact that the final trees still differ in accuracy despite them all using majority class at that point suggests that another, stronger effect exists.

The reason why the final trees using alternate prediction methods do not behave the same as HTMC in the 100KB sensor environment despite them all being theoretically equivalent comes down to differences in memory management. HTMC saves memory via poor attribute removal where the other methods do not, and in this environment even the slightest difference in available memory can have a large effect on the final tree induced. This is reflected in HTNBA performing even worse still than HTNB/HTNB1K overall, which is due to it further increasing the storage requirements of active leaves by a small amount.

Table 5.4: HTMC vs HTNB accuracy (%).

method→	HTMC			HTNB		
	memory limit			memory limit		
dataset	100KB	32MB	400MB	100KB	32MB	400MB
RTS	**96.95**	99.99	99.99	96.87	99.99	99.99
RTSN	75.20	**78.48**	**78.45**	**75.21**	78.41	78.07
RTC	**62.49**	83.00	83.02	61.22	**83.16**	**83.78**
RTCN	53.63	**62.45**	61.87	53.63	62.32	**62.50**
RRBFS	**88.56**	93.27	92.93	88.51	**93.60**	**93.52**
RRBFC	**91.36**	98.72	98.21	91.24	**98.85**	**98.44**
LED	73.94	73.99	73.96	73.94	**74.02**	**73.99**
WAVE21	81.21	84.37	84.01	**81.28**	**84.82**	**85.21**
WAVE40	81.20	84.21	83.80	81.20	**84.55**	**84.89**
GENF1	95.07	**95.07**	**95.07**	95.07	94.99	94.80
GENF2	78.46	**94.03**	**94.00**	**78.84**	94.01	93.72
GENF3	**97.50**	**97.52**	**97.52**	97.49	97.48	97.36
GENF4	93.68	**94.67**	**94.65**	**93.83**	94.65	94.27
GENF5	71.73	**92.36**	**92.15**	**71.84**	92.27	91.67
GENF6	91.89	**93.31**	**93.28**	**92.08**	93.26	92.18
GENF7	96.51	**96.81**	**96.79**	**96.52**	96.77	95.49
GENF8	99.41	**99.42**	**99.42**	99.41	99.36	99.26
GENF9	**96.07**	**96.78**	**96.74**	95.97	96.77	95.64
GENF10	99.88	**99.89**	**99.89**	**99.89**	99.84	99.84
average	85.51	90.44	90.30	85.48	90.48	90.24

Table 5.5: HTNB vs HTNB1K accuracy (%).

method→	HTNB			HTNB1K		
	memory limit			memory limit		
dataset	100KB	32MB	400MB	100KB	32MB	400MB
RTS	96.87	99.99	99.99	96.87	99.99	99.99
RTSN	75.21	**78.41**	78.07	75.21	78.39	**78.36**
RTC	61.22	83.16	**83.78**	61.22	83.16	83.53
RTCN	53.63	**62.32**	**62.50**	53.63	62.24	62.49
RRBFS	88.51	93.60	93.52	88.51	**93.61**	**93.53**
RRBFC	91.24	**98.85**	**98.44**	91.24	98.84	98.15
LED	73.94	**74.02**	73.99	73.94	74.01	73.99
WAVE21	81.28	**84.82**	**85.21**	81.28	84.80	85.20
WAVE40	81.20	**84.55**	84.89	81.20	84.49	**84.92**
GENF1	95.07	94.99	94.80	95.07	**95.02**	94.80
GENF2	78.84	94.01	93.72	78.84	94.01	**93.81**
GENF3	97.49	**97.48**	97.36	97.49	97.47	**97.37**
GENF4	93.83	94.65	94.27	93.83	94.65	**94.38**
GENF5	71.84	92.27	91.67	71.84	**92.37**	**92.00**
GENF6	92.08	93.26	92.18	92.08	93.26	**92.74**
GENF7	96.52	96.77	95.49	96.52	96.77	**95.97**
GENF8	99.41	99.36	99.26	99.41	**99.37**	**99.30**
GENF9	95.97	96.77	95.64	95.97	**96.78**	**96.10**
GENF10	99.89	99.84	99.84	99.89	**99.85**	**99.86**
average	85.48	90.48	90.24	85.48	90.48	90.34

Table 5.6: HTMC vs HTNBA accuracy (%).

method→	HTMC			HTNBA		
	memory limit			memory limit		
dataset	100KB	32MB	400MB	100KB	32MB	400MB
RTS	**96.95**	99.99	99.99	96.92	99.99	99.99
RTSN	**75.20**	78.48	**78.45**	74.91	**78.49**	78.44
RTC	**62.49**	83.00	83.02	61.22	**83.10**	**83.84**
RTCN	**53.63**	**62.45**	61.87	53.60	62.26	**63.19**
RRBFS	**88.56**	93.27	92.93	88.43	**93.60**	**93.84**
RRBFC	**91.36**	98.72	98.21	91.19	**98.85**	**98.95**
LED	73.94	73.99	73.96	**73.96**	**74.02**	**73.98**
WAVE21	81.21	84.37	84.01	**81.23**	**84.80**	**85.66**
WAVE40	81.20	84.21	83.80	81.20	**84.52**	**85.52**
GENF1	95.07	**95.07**	**95.07**	95.07	95.06	95.05
GENF2	**78.46**	94.03	94.00	78.30	**94.05**	**94.05**
GENF3	**97.50**	97.52	**97.52**	97.49	97.52	97.51
GENF4	93.68	94.67	94.65	**93.86**	**94.68**	94.65
GENF5	71.73	92.36	92.15	**72.10**	**92.41**	**92.40**
GENF6	91.89	93.31	93.28	**92.09**	93.31	**93.29**
GENF7	96.51	96.81	96.79	**96.53**	**96.82**	**96.80**
GENF8	99.41	99.42	99.42	99.41	99.42	99.42
GENF9	**96.07**	96.78	96.74	95.98	**96.81**	**96.78**
GENF10	99.88	99.89	99.89	**99.89**	99.89	99.89
average	85.51	90.44	90.30	85.44	90.51	90.70

The final accuracy results are compared in Tables 5.4–5.6. Figures in bold indicate that a particular accuracy is higher for that method than its competitor. The larger memory limits allow active leaves to survive and bring with them results demonstrating a positive gain for the Naive Bayes methods. The trends are most evident in the 400MB case, where many active leaves expose the true merit of the alternative approaches to prediction.

Figures 5.2 & 5.3 show learning curves for most data sources in the 400MB environment. The rate of sampling the points for purposes of plotting has been varied to aid the readability of the graphs. The three cases missing from the graphs (GENF8, GENF10 and LED) add little information to that already shown—GENF8 looks similar to GENF7, and apart from many more examples being processed on GENF10 the relative accuracies look similar also. LED is not a good source of accuracy comparison because all of the methods fluctuate very close to the optimal Bayes accuracy, so that the methods are hard to visually separate when plotting accuracy over time.

Of the graphs displayed, RTCN, RRBFS, RRBFC, WAVE21, WAVE40, GENF2, GENF5 and GENF9 are all convincing cases for HTNBA, where the adaptive method dominates in accuracy across the entire evaluation. In the case of GENF1 and GENF3, HTMC emerges as the superior variant, but in these cases the difference between HTMC and HTNBA is much less significant than the poor performance of HTNB. There are in fact no convincing wins to HTNB.

Table 5.4 compares the final accuracy of HTMC with that of HTNB. In the 32MB case there are more data sets for which HTMC is more accurate, but despite this the overall average looks better for HTNB. In the 400MB case HTNB looks less accurate in both the number of wins and the overall average. These losses serve as examples of the problems that Naive Bayes models can have.

Comparing HTNB with HTNB1K in Table 5.5 looks at the difference afforded by waiting a fixed period before trusting Naive Bayes models. There is absolutely no difference to be noted in the 100KB case, and even in the 32MB environment it is difficult to determine a superior method from the results. Only the 400MB environment is able to expose a slight advantage to HTNB1K, although looking at the learning curves in Figures 5.2 & 5.3, for example on RTSN, GENF4, GENF6, GENF7 and GENF9, some of the gains do look significant. On RRBFC, the fixed threshold appears to have a detrimental effect.

HTNBA is a more convincing improvement over HTNB. In Table 5.6 its accuracy is compared with the base method HTMC. In the larger memory en-

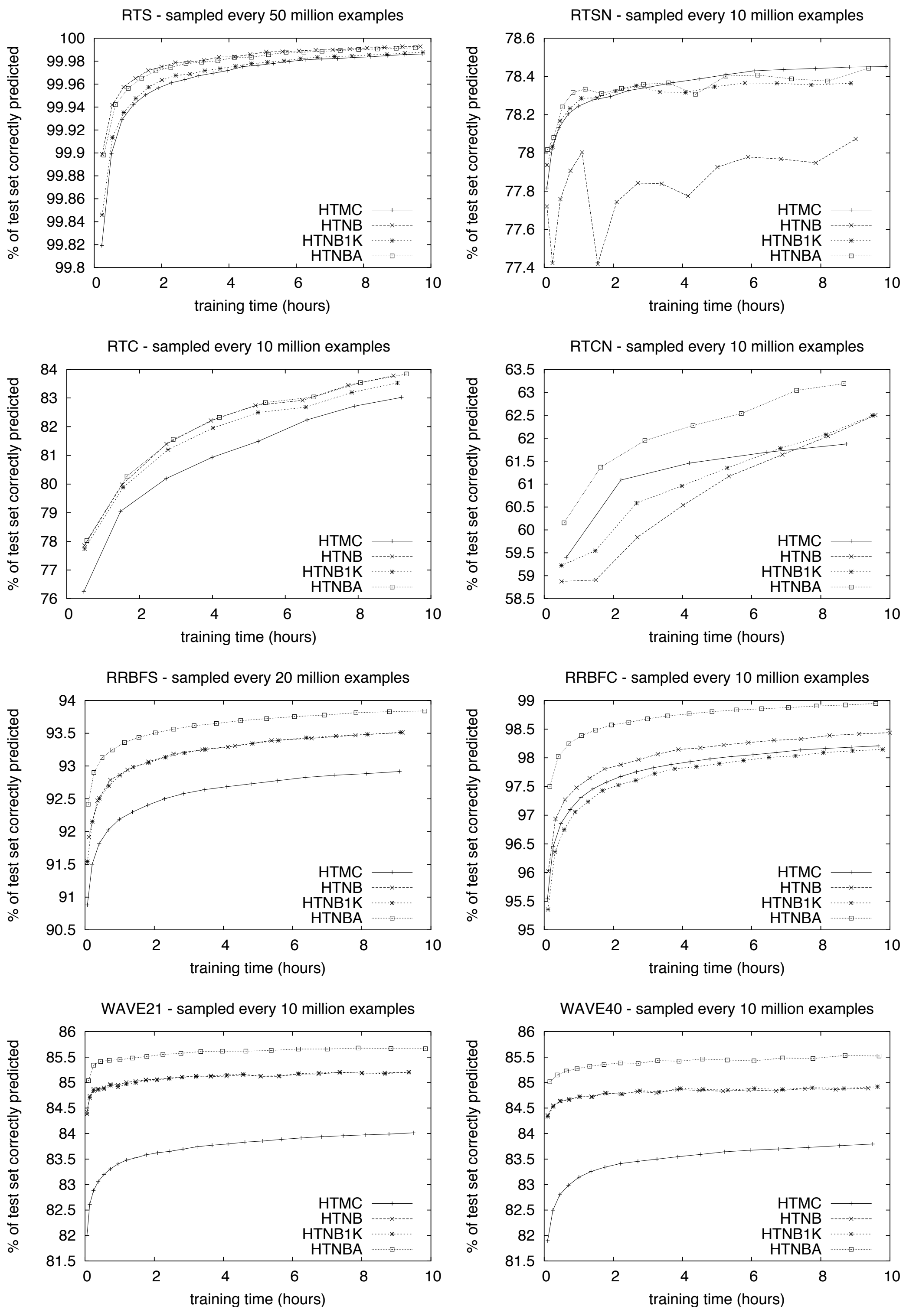

Figure 5.2: Part 1 of learning curves for prediction methods in 400MB memory limit.

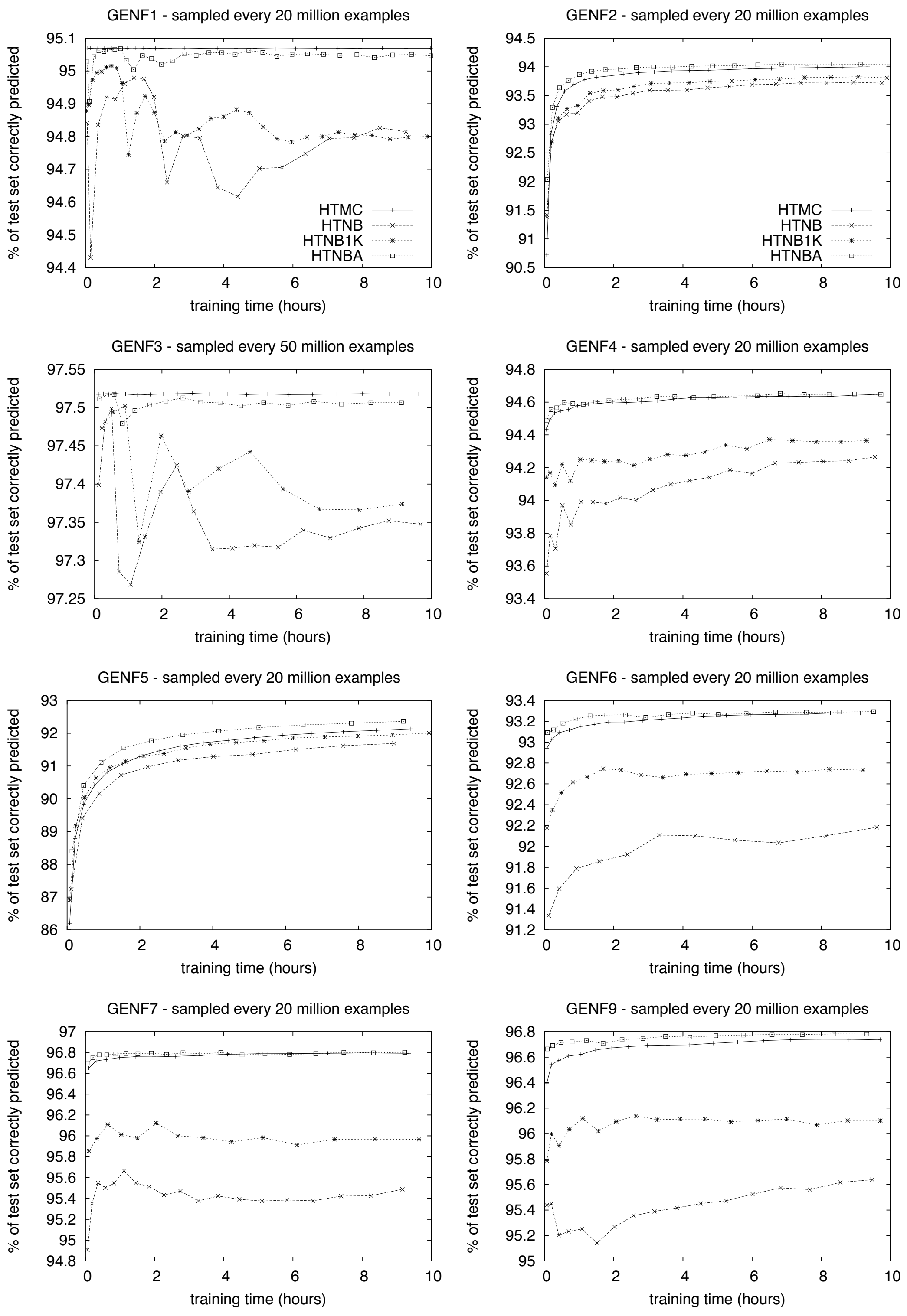

Figure 5.3: Part 2 of learning curves for prediction methods in 400MB memory limit.

vironments it makes a noticeable difference, outperforming all other methods on average. From these results, aside from its performance in 100KB of memory, the HTNBA method is the superior method of the four. A broad conclusion to be made is that the more memory available, the more benefit able to be provided by HTNBA. This is a sensible result considering how HTNBA's theoretical capacity to lift accuracy is directly impacted by deactivation of leaves, the direct consequence of limited memory. Due to this section concluding that HTNBA is an improvement on HTMC, it is the HTNBA algorithm, not HTMC, that is carried forward to the investigation in Chapter 7.

5.5 Summary

With an established induction method, a study of approaches to prediction was conducted. With the standard method of majority class prediction offering sometimes lower but more reliable accuracy compared to the enhancement of Naive Bayes leaves, a hybrid approach was introduced that adaptively chooses between the methods, and is shown to be the most accurate method overall when sufficient working memory is available.

The average accuracy of the base method HTMC, across all environments and data sets, was established at the end of the previous chapter as 88.75%. HTNBA has an average accuracy of 88.88%, representing an average relative improvement of 0.15%. Although not as significant overall as the numeric method, improving the prediction strategy has demonstrated improvement when sufficient memory is available. The improvement comes with an overall average training speed reduction of 2.25%, and an average prediction speed reduction of 9.50%.

In 100KB of memory the relative accuracy actually dropped by 0.08%, accompanied by a training speed reduction of 2.90%. In this environment HTMC is the recommended algorithm. In 32MB of memory HTNBA gained 0.08% accuracy relative to HTMC, with no change in training speed on average, and predictions that are 5.80% slower. HTNBA is marginally superior in this environment. The most accuracy gains were seen in 400MB, where the average relative gain was 0.44%. This also was without any training speed reduction on average, although it comes with a significant prediction speed reduction of 25.35% relative to HTMC. The best method in this environment is a choice between the faster predictions of HTMC or the more accurate predictions of HTNBA.

Chapter 6

Hoeffding Tree Ensembles

In machine learning classification, an *ensemble* of classifiers is a collection of several models combined together. Algorithm 6 lists a generic procedure for creating ensembles that is commonly used in batch learning.

Algorithm 6 Generic ensemble training algorithm.

```
1: Given a set of training examples $S$
2: for all models $h_m$ in the ensemble of $M$ models, $m \in \{1, 2, ..., M\}$ do
3:     Assign a weight to each example in $S$ to create weight distribution $D_m$
4:     Call base learner, providing it with $S$ modified by weight distribution
       $D_m$ to create hypothesis $h_m$
5: end for
```

This procedure requires three elements to create an ensemble:

1. A set S of training examples
2. A *base* learning algorithm
3. A method of assigning weights to examples (line 3 of the pseudo-code)

The third requirement, the weighting of examples, forms the major difference between ensemble methods. Another potential difference is the voting procedure. Typically each member of the ensemble votes towards the prediction of class labels, where voting is either *weighted* or *unweighted*. In weighted voting individual classifiers have varying influence on the final combined vote, the models that are believed to be more accurate will be trusted more than those that are less accurate on average. In unweighted voting all models have equal weight, and the final predicted class is the label chosen by the majority of ensemble members. Ensemble algorithms, algorithms responsible for inducing an ensemble of models, are sometimes known as *meta-learning* schemes. They perform a higher level of learning that relies on lower-level *base* methods to produce the individual models.

Ensemble methods are attractive because they can often be more accurate than a single classifier alone. The best ensemble methods are modular, capable of taking any base algorithm and improving its performance—any method can be taken off the shelf and 'plugged in'. Besides the added memory and computational cost of sustaining multiple models, the main drawback is loss of interpretability. A single decision tree may be easily interpreted by humans, whereas the combined vote of several trees will be difficult to interpret.

In batch learning, cases where the base model is already difficult to interpret or a *black-box* solution is acceptable, the potential for improved accuracy easily justifies the use of ensembles where possible. In the data stream setting, the memory and time requirements of multiple models need more serious consideration. The demands of a data stream application will be sensitive to the additive effect of introducing more models, and there will be limits to the numbers of models that can be supported. In limited memory learning—which is better, a single large model or many smaller models of equivalent combined size?

An ensemble of classifiers will be more accurate than any of its individual members if two conditions are met [63, 116]. Firstly, the models must be accurate, that is, they must do better than random guessing. Secondly, they must be diverse, that is, their errors should not be correlated. If these conditions are satisfied Hansen and Salamon [63] show that as the number of ensemble members increase, the error of the ensemble will tend towards zero in the limit. The accuracy of an ensemble in practice will fall short of the improvement that is theoretically possible, mostly because the members which have been trained on variations of the same training data will never be completely independent.

Dietterich surveys ensemble methods for machine learning classification [28]. He mentions three possible reasons for the success of ensemble methods in practice: statistical, computational and representational. The *statistical* contribution to success is that the risk of incorrect classification is shared among multiple members, so that the average risk is lower than relying on a single member alone. The *computational* contribution is that more than one attempt is made to learn what could be a computationally intractable problem, where each attempt guided by heuristics could get stuck in local minima, but the average of several different attempts could better solve the problem than any individual attempt. The *representational* contribution is that an ensemble of models may be more capable of representing the true function in the data than any single model in isolation.

Decision trees are the basic machine learning model being investigated by this thesis, and they make an excellent base for ensemble methods, for several reasons given below. This is verified in practice, where ensembles of decision trees are ranked among the best known methods in contemporary machine learning [23].

The statistical reasoning and the uncorrelated errors theories both suggest that member diversity is crucial to good ensemble performance. Ensemble methods typically exploit the lack of *stability* of a base learner to increase diversity, thus achieve the desired effect of better accuracy in combination. Breiman [12] defines *stable* methods as those not sensitive to small changes in training data—the more sensitive a method is to data changes the more *unstable* it is said to be. Decision trees are good candidates for ensembles because they are inherently unstable. The greedy local decisions can easily be influenced by only slight differences in training data, and the effect of a single change will be propagated to the entire subtree below. Naive Bayes in contrast is stable as it takes substantial differences in training data to modify its outcome.

In the stream setting, the nature of the Hoeffding bound decision might suggest that Hoeffding trees may not exhibit such instability. However, the bound only guides local per-split decisions—Domingos and Hulten [32] prove that overall the algorithm approximates the batch induction of decision trees, which is known to be unstable, so it follows that Hoeffding trees should also be unstable. Furthermore, the empirical results of bagged Hoeffding tree ensembles and Hoeffding option trees in the next chapter demonstrate that Hoeffding trees are indeed unstable.

Decision trees also suit the computational argument. Computing an optimal decision tree is an NP-complete problem [74], so out of necessity the algorithm performs a greedy local search. Several decision trees can approach the problem based on different searches involving local greedy decisions, which on average may form a better approximation of the true target than any single search.

It is interesting to consider the representational argument applied to ensembles of decision trees, especially where memory is limited. In theory an ensemble of decision trees can be represented by a single standard decision

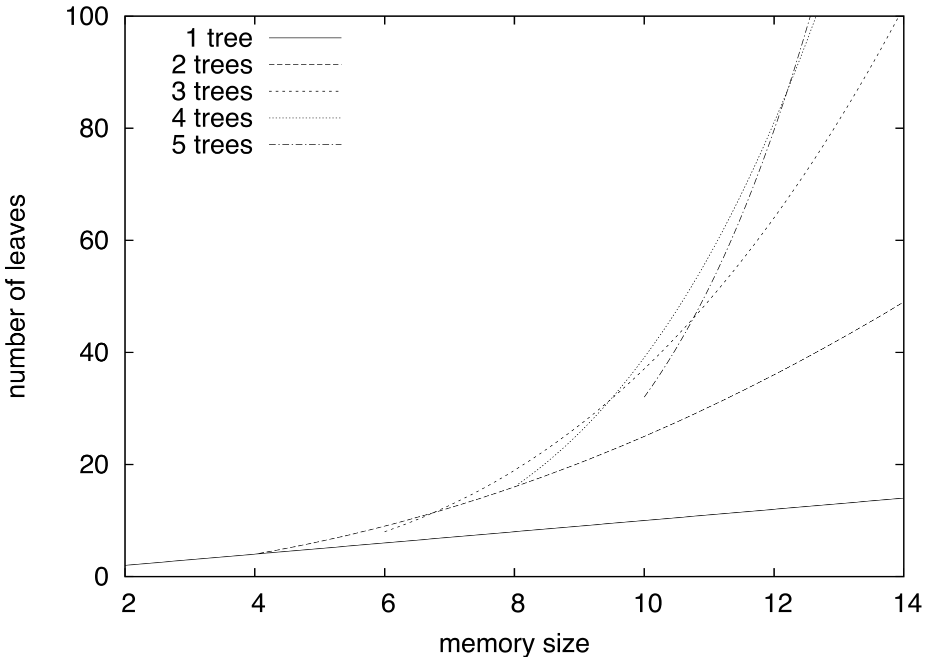

Figure 6.1: A simple model of the leaf count of combinations of decision trees as a function of total memory size.

tree, but the cost of constructing such a tree is expensive. Consider the process of combining two trees, A and B. This can be achieved by replacing every leaf of A with a subtree that is a complete copy of tree B, where the leaves copied from B are merged with the leaf of A that was replaced. Quinlan [103] shows that the procedure is multiplicative, with only limited opportunity to simplify the resulting combined tree in most cases.

Each leaf of a decision tree represents a particular region of the example space. The more leaves a tree has, the more regions are isolated by the tree, and the more potential it has to reduce error. So the number of leaves in a tree is in some way related to its "representational power". The tree multiplication argument above suggests that the effective number of leaves in an ensemble of n trees, each having l leaves, should be approximately l^n. Assume that the number of leaves that can be supported is a linear function of memory size m. Given that m is fixed, the average number of leaves in individual trees in an ensemble of n trees is at least two, and at most m/n. Combining these simplifying assumptions, the relative number of leaves in an ensemble of n trees can be modeled by the function $(m/n)^n$, where $m/n \geq 2$. Figure 6.1 plots this function for ensemble sizes one to five. The figure shows for example that at memory size ten, an ensemble of three or four trees will effectively have more leaves than an ensemble of five trees, providing superior representation capacity.

These assumptions are unlikely to hold in practice. The leaf count of a tree is unlikely to be a perfect linear function of m, and it is also questionable whether the leaf count of tree combinations is perfectly multiplicative. The number of leaves will not directly translate into accuracy, which is obviously bounded and influenced by other factors. Despite these flaws the exercise demonstrates something useful about the expected behaviour of tree ensembles in limited memory—that a given number of combined trees will only be beneficial when sufficient memory is available, and the more trees involved, the more memory required. Empirical evidence for this is found in Chapter 7.

A useful way to analyze ensemble behaviour is to consider the implications of *bias* and *variance* [55, 86]. The typical formulation breaks the error of an algorithm into three parts:

$$error = bias^2 + variance + noise \tag{6.1}$$

Given a fixed learning problem, the first component of error is the *bias* of the machine learning algorithm, that is, how closely it matches the true function of the data on average over all theoretically possible training sets of a given size. The second error component is the *variance*, that is, how much the algorithm varies its model on different training sets of the given size. The third component of error is the intrinsic noise of the data, which an algorithm will not be able to overcome, thus setting an upper bound on the achievable accuracy. There is often a tradeoff between bias and variance, where reducing one can come at the expense of the other. Bias-variance decomposition [86] is a tool that can help with understanding the behaviour of machine learning methods, and is discussed where appropriate throughout the chapter. The discussion of results in Chapter 7 uses empirical estimation of bias and variance to help with analysis.

The most common ensemble methods work like Algorithm 6, producing different models by manipulating the weight of each training example. This is why the stability of the base learner is significant. Other approaches that fall outside of this general model are not studied in this thesis. They include removing attributes, such as attempted by Tumer and Ghosh [116]; manipulating the class labels, as used by Dietterich and Bakiri [30] in their *error-correcting output codes* technique; and introducing randomness to the base model inducer, such as Breiman's popular *random forest* approach [17].

The following sections look at promising methods for improving accuracy, first in the batch setting (Section 6.1), followed by application to data streams (Section 6.2). First *bagging* (Sections 6.1.1 & 6.2.1) and *boosting* (Section 6.1.2 & 6.2.2) are investigated, then a third alternative, *option trees* (Section 6.1.3 & 6.2.3) are explored which offer a compromise between a single model and an ensemble of models. Section 6.3 considers the numbers of ensemble members that can be supported in the batch and stream settings.

6.1 Batch Setting

6.1.1 Bagging

Bagging (**b**ootstrap **agg**regat**ing**) was introduced by Breiman [12]. The procedure is simple—it combines the unweighted vote of multiple classifiers, each of which is trained on a different *bootstrap replicate* of the training set. A bootstrap replicate is a set of examples drawn randomly *with replacement* from the original training data, to match the size of the original training data. The probability that any particular example in the original training data will be chosen for a random bootstrap replicate is 0.632, so each model in the ensemble will be trained on roughly 63.2% of the full training set, and typically some examples are repeated several times to make the training size match the original.

Viewed in terms of the generic ensemble algorithm (Algorithm 6), in determining the weight distribution of examples for a particular model D_m the weights will correspond to the number of times that an example is randomly drawn—those examples that are *out-of-bag* will have a weight of zero, while the majority are likely to have a pre-normalized weight of one, with those examples randomly selected more than once having a pre-normalized weight of more than one.

Bagging works best when the base algorithm is unstable, because the models produced will differ greatly in response to only minor changes in the training data, thus increasing the diversity of the ensemble. In terms of bias and variance, bagging greatly reduces the variance of a method, by averaging the diverse models into a single result. It does not directly target the bias at all, so methods whose variance is a significant component of error have the most to gain from this method.

6.1.2 Boosting

Boosting emerged from the field of *Computational Learning Theory*, a field that attempts to do mathematical analysis of learning. It tries to extract and formalize the essence of machine learning endeavours, with the hope of producing mathematically sound proofs about learning algorithms. The discovery and success of boosting shows that such efforts can positively impact the practical application of learning algorithms.

A framework that has become one of the main influences in the field is *PAC Learning* (**P**robably **A**pproximately **C**orrect), proposed by Valiant [119]. Two concepts were defined by Kearns and Valiant [80], the *strong* learner and the *weak* learner. A strong learner is one that is highly accurate. Formally, this is defined as a learner that given training examples drawn randomly from a binary concept space is capable of outputting a hypothesis with error no more than ϵ, where $\epsilon > 0$, and does so with probability of at least $1 - \delta$, where $\delta > 0$. The defining requirement is that this must be achieved in runtime that is polynomial in all of $1/\epsilon$, $1/\delta$, and complexity of the target concept. A weak learner has the same requirements except that informally it needs only do slightly better than chance. Formally this means that in the weak case $\epsilon \leq 1/2 - \gamma$ where $0 < \gamma < 1/2$.

When these two notions of learning strength were introduced it was unknown whether the two are in fact equivalent, that is, whether a weak learner is also capable of strong learning. Schapire's paper "The Strength of Weak Learnability" [107] was the breakthrough that confirmed that this is indeed so. The paper shows that in the PAC framework weak learners are equivalent to strong learners by presenting an algorithm that is capable of "boosting" a weak learner in order to achieve high accuracy. Henceforth, the formal definition of a boosting algorithm is one that transforms a weak learner into a strong one.

Schapire's original hypothesis boosting algorithm works by combining many weak learners into a strong ensemble, as follows:

1. induce weak learner $h1$ as normal

2. train weak learner $h2$ with filtered examples, half of which $h1$ predicts correctly and the other half which $h1$ predicts incorrectly

3. train weak learner $h3$ only with examples that $h1$ and $h2$ disagree on

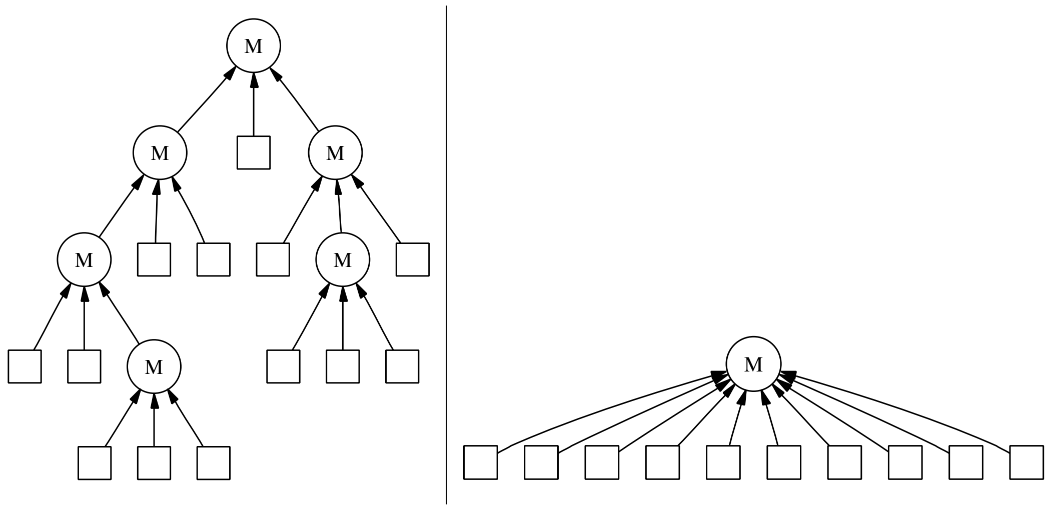

Figure 6.2: Left hand side: recursive tree structure used by original hypothesis boosting. Right hand side: flat structure of *boost-by-majority* and AdaBoost.

4. combine the predictions of $h1$, $h2$ and $h3$ by majority vote; if $h1$ and $h2$ agree then output the agreed upon class, otherwise output $h3$'s prediction

Schapire proves with certain guarantees in the PAC framework that the combined vote of the hypotheses will be more accurate than $h1$ alone. By recursively applying this process, effectively creating a tree of hypotheses with three-way branches that are combined by majority vote (see left hand side of Figure 6.2), the weak learners can be progressively improved to form a classifier of combined weak hypotheses that is just as powerful as a single strong hypothesis.

Schapire's work was improved by Freund [41], who presented an algorithm that is more efficient than Schapire's. In fact, Freund shows that the number of hypotheses needed for his *boost-by-majority* algorithm to reach a given accuracy is the smallest number possible. In this algorithm, the hypotheses are generated in a single flat level in sequence (see right hand side of Figure 6.2), as opposed to Schapire's recursive tree structure. The number of boosting iterations are fixed before induction begins, and there are two variants of the algorithm described, boosting by *sampling* and boosting by *filtering*.

Boosting by sampling is a method that suits the batch learning scenario. The first step of this process is to collect a training set from the concept space by requesting an entire batch of training examples, thus generating the set S. The goal of the process is then to create a hypothesis that is correct on all examples in S. To do so, the most straightforward approach is to retain the entire set of S in memory.

Boosting by filtering is a scenario that is more suited to incremental processing of data streams. There are interesting parallels between the theoretical learning situation proposed in the PAC framework and the challenges of data stream classification. This version of the algorithm works by selectively filtering the examples that are passed to the weak learners. The filter will either reject examples and discard them, or accept an example and pass it on to the appropriate weak learner. This variant has theoretical advantages over the sampling approach. In the filtering setting it is easier to analyze the expected amount of generalization that can be achieved, and the method can have superior space complexity due to not storing an entire training set.

The main problem with the *boost-by-majority* algorithm is that for it to work correctly the error (γ) of the weak learner must be known in advance. That is, how much better the base learner will perform over random guessing needs to be known before learning begins. This problem is serious enough to prevent the algorithm from being successful in practice.

The next advance in boosting algorithms was a combined effort of both Freund and Schapire [44]. The *AdaBoost* algorithm adjusts **ada**ptively to the errors of weak hypotheses, thus overcoming the problem of boosting by majority. The algorithm was introduced by taking an online allocation algorithm named *Hedge*, generalized from the *weighted majority algorithm* [91], and transforming it into a boosting algorithm. AdaBoost is a boosting by sampling method, and unfortunately a complementary filtering variant was not proposed. Limited to the sampling setting, it is shown that AdaBoost will drive training error exponentially towards zero in the number of iterations performed. AdaBoost is the most well known and successful boosting algorithm in practice. This is mostly due to it showing empirical evidence of accurately generalizing to data not included in the training set. Another reason for its popularity is that the algorithm is simple and intuitive. Freund and Schapire were awarded the 2003 Gödel Prize for their work in recognition of the significant influence AdaBoost has had on the machine learning field and science in general.

Algorithm 7 lists the AdaBoost pseudo-code. The intuitive understanding is that each new model in sequence will concentrate more on correctly classifying the examples that the previous models misclassified, and concentrate less on those that are already correct. To start with, all examples have equal weight (line 1). T is the number of boosting iterations preformed. Every iteration starts by inducing a model on the data based on the weights currently assigned to examples, and the error, ϵ, of the new model is estimated on the

Algorithm 7 AdaBoost. Input is a sequence of m examples, **WeakLearn** is the base weak learner and T is the number of iterations.

```
 1: Initialize $D_1(i) = 1/m$ for all $i \in \{1, 2, ..., m\}$
 2: for $t$ = 1,2,...$T$ do
 3:    Call WeakLearn, providing it with distribution $D_t$
 4:    Get back hypothesis $h_t : X \rightarrow Y$
 5:    Calculate error of $h_t : \epsilon_t = \sum_{i:h_t(x_i) \neq y_i} D_t(i)$
 6:    if $\epsilon_t \geq 1/2$ then
 7:       Set $T = t - 1$
 8:       Abort
 9:    end if
10:    Set $\beta_t = \epsilon_t/(1 - \epsilon_t)$
11:    Update distribution $D_t : D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} \beta_t & \text{if } h_t(x_i) = y_i \\ 1 & \text{otherwise} \end{cases}$
       where $Z_t$ is a normalization constant (chosen so $D_{t+1}$ is a probabilty
       distribution)
12: end for
13: return final hypothesis: $h_{fin}(x) = \arg\max_{y \in Y} \sum_{t:h_t(x)=y} \log 1/\beta_t$
```

training data (lines 3-5). Lines 6-9 handle the special case where the error of the weak learner is worse than random guessing. A situation like this will confuse the weighting process so the algorithm aborts to avoid invalid behaviour. Lines 10-11 reweight the examples—those that are incorrectly classified retain their weight relative to correctly classified examples whose weight is reduced by $\frac{\epsilon}{1-\epsilon}$, and the weights of all examples are normalized. The process repeats until a sequence of T models have been induced. To predict the class label of examples (line 13), each model in the final boosted sequence votes towards a classification, where each model's vote is weighted by $-\log \frac{\epsilon}{1-\epsilon}$ so that models with lower error have higher influence on the final result.

The weak learner is influenced either by *reweighting* or *resampling* the training examples. Reweighting the examples requires that the weak learner respond to different continuous weights associated with each example. Common learners such as C4.5 and Naive Bayes have this capability. Resampling involves randomly sampling a new training set from the original according to the weight distribution. The advantage of resampling is that the weak learner need not handle continuous example weights, but the resampling approach introduces a random element to the process. When Breiman [13] compared the approaches he found that accuracy did not significantly differ between the two.

Breiman looked at AdaBoost from a different perspective [13]. He introduced his own terminology, describing the procedure as *arcing* (**a**daptively

resample and **c**ombine), and refers to AdaBoost as *arc-fs* (for **F**reund and **S**chapire). He found arc-fs to be very promising and capable of significantly outperforming bagging. As an exercise to test his theory that AdaBoost's success is due to the general adaptive resampling approach and not dependent on the precise formulation of arc-fs, he introduced a simpler ad-hoc algorithm, *arc-x4*, listed in Algorithm 8. The main differences from AdaBoost are:

1. A simpler weight update step. Each example is relatively weighted by $1 + e^4$ where e is the number of misclassifications made on the example by the existing ensemble.

2. Voting is unweighted.

In experimental comparison [13], arc-x4 performed on a comparable level to arc-fs. Both were often more successful than bagging at reducing test set error.

Algorithm 8 Arc-x4, Breiman's ad-hoc boosting algorithm.

```
1: Initialize $D_1(i) = 1/m$ for all $i \in \{1, 2, ..., m\}$
2: for $t$ = 1,2,...$T$ do
3:     Call WeakLearn, providing it with distribution $D_t$
4:     Get back hypothesis $h_t : X \to Y$
5:     Count misclassifications: $e_i = \sum_{n=1}^{t} I(h_n(x_i) \neq y_i)$
6:     Update distribution $D_t$ : $D_{t+1}(i) = \frac{1+e_i^4}{\sum_{n=1}^{m} 1+e_n^4}$
7: end for
8: return final hypothesis: $h_{fin}(x) = \arg\max_{y \in Y} \sum_{t=1}^{T} I(h_t(x) = y)$
```

Breiman [13] employs bias and variance analysis as a tool to help explain how boosting works. Whereas bagging reduces mostly variance, boosting has the ability to reduce both bias and variance. An intuitive understanding of boosting's ability to reduce bias is that subsequent boosting iterations have an opportunity to correct the bias of previous models in the ensemble. Breiman was still puzzled by some aspects of the behaviour of boosting, prompting deeper investigation by Freund and Schapire [45].

It is accepted and understood how AdaBoost reduces error on the training set, but uncertainty remains as to why error on test examples, the *generalization* error, continues to decrease after training set error reaches zero. Researchers including Breiman were surprised at AdaBoost's ability to generalize well beyond any theoretical explanations and seemingly in contradiction to *Occam's razor*, which suggests that simpler solutions should be preferred over more complex ones. Schapire and Freund et al. [108] tried to explain this phenomenon in their paper "Boosting the Margin: A New Explanation for the Effectiveness of Voting Methods". They suggest that the *margin* of the predictions, the difference between the confidence of the true class and the confidence of the highest other class, continues to improve with additional boosting iterations, explaining the ability to improve generalization. However the uncertainty continues, as Breiman [14, 16] provides contrary evidence and claims that the margin explanation is incomplete. The true explanation for AdaBoost's generalization ability remains uncertain, but this of course does not prevent its use in practice.

The statisticians Friedman, Hastie and Tibshirani [46] point out the similarities between AdaBoost and *additive logistic regression*, a more established and commonly understood statistical technique. From an optimization point of view they show that AdaBoost performs gradient descent in function space that aims to minimise the following *exponential loss* function:

$$\exp\left(-y_i \sum_{t=1}^{T} \alpha h_t(x_i)\right) \tag{6.2}$$

Friedman et al. question why exponential loss is used, suggesting other possibilities, and propose a boosting variant named *LogitBoost*. Their insight has provided another perspective on AdaBoost, which further aids researchers in understanding its workings. Another contribution they provide is the idea of *weight trimming*—after several boosting iterations there can exist examples with very low weight, so low in fact that ignoring them will not harm accuracy while reducing computation.

Schapire and Singer [109] make further improvements to AdaBoost. The original formulation only used binary votes of ensemble members to reweight examples. The improved formulation generalizes the algorithm to make use of confidences output by base members, potentially improving performance. They show how to handle multiple class labels and also study how to best devise algorithms to output confidences suiting the improved AdaBoost. Additionally, they propose an alternative gain criterion, so that for example, decision trees can be induced that aim to directly improve boosting performance.

Early enthusiasm for AdaBoost sometimes overlooked its shortcomings. Observing its spectacular ability to generalize on many data sets it was sometimes suggested that AdaBoost is resistant to overfitting. Dietterich [29] addressed such claims by demonstrating that AdaBoost can suffer from problems with noise. In fact, his experiments show that with substantial classification noise, bagging is much better than boosting. The explanation for this behaviour is that AdaBoost will inevitably give higher weight to noisy examples, because they will consistently be misclassified.

An interesting piece of follow-up work was conducted by Freund [42], who revisited his *boost-by-majority* algorithm to make it adaptive like AdaBoost. The result is the *BrownBoost* algorithm. BrownBoost considers the limit in which each boosting iteration makes an infinitesimally small contribution to the whole process, which can be modelled with **Brown***ian motion*. As with the original boost-by-majority, the number of iterations, T, is fixed in advance. The algorithm is optimized to minimize the training error in the pre-assigned number of iterations—as the final iteration draws near, the algorithm gives up on examples that are consistently misclassified. This offers hope of overcoming the problems with noise that are suffered by AdaBoost. In fact, Freund shows that AdaBoost is a special case of BrownBoost, where $T \to \infty$. Despite the theoretical benefits of BrownBoost it has failed to draw much attention from the community, perhaps because it is much more complicated and less intuitive than AdaBoost. The study [95] found that BrownBoost is more robust than AdaBoost in class noise, but that LogitBoost performs at a comparable level.

6.1.3 Option Trees

Standard decision trees have only a single path that each example can follow[1], so any given example will apply to only one leaf in the tree. Option trees, introduced by Buntine [21] and further explored by Kohavi and Kunz [84], are more general, making it possible for an example to travel down multiple paths and arrive at multiple leaves. This is achieved by introducing the possibility of *option nodes* to the tree, alongside the standard decision nodes and leaf nodes. An option node splits the decision path several ways—when an option node is encountered several different subtrees are traversed, which could themselves contain more option nodes, thus the potential for reaching different leaves is multiplied by every option. Making a decision with an option tree involves combining the predictions of the applicable leaves into a final result.

Option trees are a single general structure that can represent anything from a single standard decision tree to an ensemble of standard decision trees

[1]For simplicity, this statement ignores missing value approaches such as C4.5 that send examples down several paths when testing on unknown attribute values.

to an even more general tree containing options within options. An option tree without any option nodes is clearly the same as a standard decision tree. An option tree with only a single option node at the root can represent an ensemble of standard trees. Option nodes below other option nodes take this a step further and have the combinational power to represent many possible decision trees in a single compact structure.

A potential benefit of option trees over a traditional ensemble is that the more flexible representation can save space—consider as an extreme example an ensemble of one hundred mostly identical large trees, where the only difference between each tree lies at a single leaf node, in the same position in each tree. The standard ensemble representation would require one hundred whole copies of the tree where only the leaf would differ. Efficiently represented as an option tree this would require almost a hundred times less space, where the varying leaf could be replaced by an option node splitting one hundred ways leading to the one hundred different leaf variants. The drive for diverse ensembles will of course make such a scenario unlikely, but the illustration serves the point that there are savings to be made by combining different tree permutations into a single structure. Essentially every path above an option node can be saved from repetition in memory, compared to explicitly storing every individual tree found in combination.

Another possible benefit offered by option trees is retention of interpretability. An accepted disadvantage of ensemble methods is that users will lose the ability to understand the models produced. An option tree is a single structure, and in some situations this can aid in interpreting the decision process, much more so than trying to determine the workings of several completely separate models in combination. An option tree containing many options at many levels can be complex, but humans may not be as confused by a limited number of option nodes in small and simple option trees. Arguments for the interpretability of option trees can be found in [84, 43].

In terms of accuracy, an option tree is just as capable of increasing accuracy as any ensemble technique. Depending on how it is induced, an option tree could represent the result of bagging or boosting trees, or something else entirely. An example application of boosting to option tree induction is the *alternating decision tree* (ADTree) learning algorithm by Freund and Mason [43].

The option tree induction approach by Kohavi and Kunz [84] seeks to explore and average additional split possibilities that a tree could make. Their approach is closer to bagging than boosting, because bagging also draws out different possibilities from the trees, but operates in a random and less direct fashion. It is less like boosting because it does not utilize classification performance as feedback for improving on previous decisions, but blindly tries out promising-looking paths that have previously not been explored. They provide two main reasons why such trees should outperform a single decision tree, *stability* and *limited lookahead.* The stability argument has already been discussed with other ensemble methods—a single tree can differ wildly on small changes in training data, but an option tree that combines several possibilities would vary less on average. The limited lookahead argument refers to the fact that standard decision trees make greedy local decisions, and do not consider better decisions that could be made if the search looked ahead before committing. Looking ahead is expensive and studies suggest that looking ahead a few steps is futile [99], but when the tree considers attributes in isolation it will be unaware of potential interactions between attributes that could greatly increase accuracy. By exploring several alternative options, an option tree increases the chance of utilizing rewarding attribute combinations.

A significant difficulty with inducing option trees is that they will grow very rapidly if not controlled. This is due to their powerful combinatorial nature which gives them high representational power, but which can also easily explore too many options if left unchecked. Kohavi and Kunz employ several strategies to prevent a combinatorial explosion of possibilities. Firstly, they set a parameter that controls how close other tests must be to the best test to be considered as extra options. The larger the *option factor*, the more potential there is for additional options to be explored. Secondly, they impose an arbitrary limit of five options per node. Because this is enforced locally per node, it will slow down but not completely prevent excessive combinations. Thirdly, they experimented with restricting splits to certain depths of the tree, either the lowest three levels or the top two levels, and also tried altering the frequency of options as a function number of supporting examples or tree depth.

Kohavi and Kunz [84] compared their option trees to bagged trees in experiments, showing that option trees are superior. Their hypothesis was that options nearer the root are more useful than options further down the tree, which was confirmed in their experiments. Interestingly, this opinion differs from that of Buntine [21], who argued that options are more effective nearer the leaves.

6.2 Data Stream Setting

6.2.1 Bagging

Bagging as formulated by Breiman does not seem immediately applicable to data streams, because it appears that the entire data set is needed in order to construct bootstrap replicates. Oza and Russell [102] show how the process of sampling bootstrap replicates from training data can be simulated in a data stream context. They observe that the probability that any individual example will be chosen for a replicate is governed by a *Binomial* distribution, so the sampling process can be approximated by considering each example in turn and randomly deciding with a Binomial probability distribution how many times to include the example in the formation of a replicate set. The difficulty with this solution is that the number of examples, N, needs to be known in advance. Oza and Russell get around this by considering what happens when $N \to \infty$, which is a reasonable assumption to make with data streams of arbitrary length, and conclude that the Binomial distribution tends to a *Poisson*(1) distribution. Following these observations the algorithm is straightforward to implement, listed in Algorithm 9. It requires a base learner that is also capable of processing data streams.

Algorithm 9 Oza and Russell's *Online Bagging.* M is the number of models in the ensemble and $I(\cdot)$ is the indicator function.

```
 1: Initialize base models $h_m$ for all $m \in \{1, 2, ..., M\}$
 2: for all training examples do
 3:    for $m = 1, 2, ..., M$ do
 4:       Set $k = Poisson(1)$
 5:       for $n = 1, 2, ..., k$ do
 6:          Update $h_m$ with the current example
 7:       end for
 8:    end for
 9: end for
10: anytime output:
11: return hypothesis: $h_{fin}(x) = \arg\max_{y \in Y} \sum_{t=1}^{T} I(h_t(x) = y)$
```

Oza and Russell construct proofs to demonstrate that their "online" bagging algorithm converges towards the original batch algorithm [102], and have collected experimental evidence to show this [101]. One issue they have not addressed is memory management, a consideration that is paramount in this thesis. Assuming that a memory-limited base algorithm is available, such as the Hoeffding tree algorithm studied in Chapter 3, then the memory requirements of an ensemble of trees can be controlled by limits on individual members. The experimental implementation of bagged Hoeffding trees takes a simple approach to controlling total memory usage—with an overall memory limit of m and a total of n trees, each tree is limited to a maximum size of m/n.

6.2.2 Boosting

Existing literature for the problem of applying boosting to data stream classification has mainly focussed on modifying AdaBoost, and often under the guise of *online* learning [102, 36]. The term "online" is slightly ambiguous, as researchers have used it in the past to refer to varying goals. Typically the focus is on processing a single example at a time, but sometimes without emphasis on other factors believed important in this thesis such as memory and time requirements. For the most part however the online methods reviewed here are directly applicable to the data stream setting. Attempts at modifying boosting to work in an "online" fashion can be divided into two main approaches: *block* boosting and *parallel* boosting.

Block boosting involves collecting data from the stream into sequential blocks, reweighting the examples in each block in the spirit of AdaBoost, and then training a batch learner to produce new models for inclusion in the ensemble. An advantage of this approach is that specialized data stream based learners are not required, the boosting "wrapper" algorithm handles the data stream. Memory management by such a method can be achieved by discarding weaker models, but this raises interesting questions—traditional AdaBoost depends on the previous set of ensemble members remaining constant, the effect of removing arbitrary models from an ensemble during learning is unknown. Margineantu and Dietterich [93] look at pruning models from AdaBoost ensembles and find it effective, although they do this after learning is complete. A significant difficulty with block boosting is deciding how large the blocks should be. A demanding batch learner and/or overly large block sizes will limit the rate at which examples can be processed, and block sizes that are too small will limit accuracy. Examples of block boosting include Breiman's pasting of 'bites' [15], and the study by Fan et al. [36].

Parallel boosting involves feeding examples as they arrive into a base data stream algorithm that builds each ensemble member in parallel. A difficulty is that AdaBoost was conceived as a sequential process. Sequential weighting of training examples can be simulated by feeding and reweighting examples through each member in sequence. This does not directly emulate the strictly sequential process of AdaBoost, as models further down the sequence will start learning from weights that depend on earlier models that are also evolving over time, but the hope is that in time the models will converge towards an AdaBoost-like configuration. An advantage of using data stream algorithms as base learners is that the ensemble algorithm can inherit the ability to adequately handle data stream demands.

Examples of parallel boosting include Fern and Givan's [38] online adaptation of arc-x4, and Domingo and Watanabe's [31] *MadaBoost*. Domingo and Watanabe describe difficulties with correctly weighting examples when applying AdaBoost to the online setting. Their solution is essentially to put a limit on the highest weight that can be assigned to an example. They try to prove that their modification is still boosting in the formal PAC-learning sense but face theoretical problems that prevent them from providing a full proof. Bshouty and Gavinsky [20] present a more theoretically sound solution to the problem, performing *polynomially smooth* boosting, but the result is far less intuitive than most AdaBoost-like algorithms.

The data stream boosting approach adopted by this thesis is a parallel boosting algorithm that was developed by Oza and Russell [102], who compliment their online bagging algorithm (Section 6.2.1) with a similar approach to online boosting. The pseudo-code is listed in Algorithm 10. Oza and Russell note that the weighting procedure of AdaBoost actually divides the total example weight into two halves—half of the weight is assigned to the correctly classified examples, and the other half goes to the misclassified examples. As the ensemble gets more accurate the number of misclassified examples should progressively get less and less relative to the number of correct classifications. In turn, the misclassified set gets more weight per example than the correctly classified set. This motivates the weight update procedure in lines 9-15, which is intended to simulate the batch weight behaviour in a streaming environment, much like the online bagging algorithm is designed to simulate the creation of bootstrap replicates. Once again they utilize the Poisson distribution for deciding the random probability that an example is used for training, only this time the parameter (λ_d) changes according to the boosting weight of the example as it is passed through each model in sequence. The use of random Poisson is well founded for bagging, but motivation for it in the boosting situation is less

Algorithm 10 Oza and Russell's *Online Boosting.* N is the number of examples seen.

```
1: Initialize base models $h_m$ for all $m \in \{1, 2, ..., M\}$, $\lambda_m^{sc} = 0, \lambda_m^{sw} = 0$
2: for all training examples do
3:    Set "weight" of example $\lambda_d = 1$
4:    for $m = 1, 2, ..., M$ do
5:       Set $k = Poisson(\lambda_d)$
6:       for $n = 1, 2, ..., k$ do
7:          Update $h_m$ with the current example
8:       end for
9:       if $h_m$ correctly classifies the example then
10:         $\lambda_m^{sc} \leftarrow \lambda_m^{sc} + \lambda_d$
11:         $\lambda_d \leftarrow \lambda_d \left(\frac{N}{2\lambda_m^{sc}}\right)$
12:      else
13:         $\lambda_m^{sw} \leftarrow \lambda_m^{sw} + \lambda_d$
14:         $\lambda_d \leftarrow \lambda_d \left(\frac{N}{2\lambda_m^{sw}}\right)$
15:      end if
16:   end for
17: end for
18: anytime output:
19: Calculate $\epsilon_m = \frac{\lambda_m^{sw}}{\lambda_m^{sc} + \lambda_m^{sw}}$ and $\beta_m = \epsilon_m/(1 - \epsilon_m)$ for all $m \in \{1, 2, ..., M\}$
20: return hypothesis: $h_{fin}(x) = \arg\max_{y \in Y} \sum_{m:h_m(x)=y} \log 1/\beta_m$
```

clear. Preliminary experimental work for this thesis involved testing several boosting algorithms, including a modification of Oza and Russell's boosting algorithm that applied example weights directly instead of relying on random drawings from Poisson. Overall the algorithm performed better *with* the random Poisson element than without it, so it seems to be a worthwhile approach even if the theoretical underpinning is weak.

Of the boosting strategies tested in preliminary experimentation, Oza and Russell's approach represents the best found. More discussion about the performance of boosting in data streams is reserved for Section 7.2. As with bagging, the memory management strategy tested in Chapter 7 uses the simple approach of controlling total ensemble size by enforcing uniform memory limits to each base tree.

6.2.3 Option Trees

A new algorithm for inducing option trees from data streams was devised for this thesis. It is based on the Hoeffding tree induction algorithm, but generalized to explore additional options in a manner similar to the option trees of Kohavi and Kunz [84]. The pseudo-code is listed in Algorithm 11. The algorithm is an extension of the basic decision tree inducer listed in Algorithm 2 on page 47. The differences are, firstly, that each training example can update a set of option nodes rather than just a single leaf, and secondly, lines 20-32 constitute new logic that applies when a split has already been chosen but extra options are still being considered.

A minor point of difference between Kohavi and Kunz's option tree and the Hoeffding option trees implemented for this study is that the Hoeffding option tree uses the class confidences in each leaf to help form the majority decision. The observed probabilities of each class are added together, rather than the binary voting process used by Kohavi and Kunz where each leaf votes completely towards the local majority in determining the overall majority label. Preliminary experiments suggested that using confidences in voting can slightly improve accuracy.

Just as Kohavi and Kunz needed to restrict the growth of their option trees in the batch setting, strategies are required to control the growth of Hoeffding option trees. Without any restrictions the tree will try to explore all possible tree configurations simultaneously which is clearly not feasible.

The first main approach to controlling growth involves the initial decision of when to introduce new options to the tree. Kohavi and Kunz have an *option factor* parameter that controls the inducer's eagerness to add new options. The Hoeffding option tree algorithm has a similar parameter, δ', which can be thought of as a secondary Hoeffding bound confidence. The δ' parameter controls the confidence of secondary split decisions much like δ influences the initial split decision, only for additional splits the test is whether the information gain of the best candidate exceeds the information gain of the best split already present in the tree. For the initial split, the decision process searches for the best attribute overall, but for subsequent splits, the search is for attributes that are superior to existing splits. It is very unlikely that any other attribute could compete so well with the best attribute already chosen that it could beat it by the same initial margin. Recall, the original Hoeffding bound decision is designed to guarantee that the other attributes should not be any better. For this reason, the secondary bound δ' needs to be much looser than the initial bound δ for there to be any chance of additional attribute choices. It seems a contradiction to require other attributes to be better when the Hoeffding bound has already guaranteed with high confidence that they are not as good, but the guarantees are much weaker when tie breaking has been

Algorithm 11 Hoeffding option tree induction algorithm. δ' is the confidence for additional splits and *maxOptions* is the maximum number of options that should be reachable by any single example.

```
1: Let $HOT$ be an option tree with a single leaf (the root)
2: for all training examples do
3:    Sort example into leaves/option nodes $L$ using $HOT$
4:    for all option nodes $l$ of the set $L$ do
5:       Update sufficient statistics in $l$
6:       Increment $n_l$, the number of examples seen at $l$
7:       if $n_l \ mod \ n_{min} = 0$ and examples seen at $l$ not all of same class then
8:          if $l$ has no children then
9:             Compute $\overline{G}_l(X_i)$ for each attribute
10:            Let $X_a$ be attribute with highest $\overline{G}_l$
11:            Let $X_b$ be attribute with second-highest $\overline{G}_l$
12:            Compute Hoeffding bound $\epsilon = \sqrt{\frac{R^2 ln(1/\delta)}{2n_l}}$
13:            if $X_a \neq X_\emptyset$ and $(\overline{G}_l(X_a) - \overline{G}_l(X_b) > \epsilon$ or $\epsilon < \tau)$ then
14:               Add a node below $l$ that splits on $X_a$
15:               for all branches of the split do
16:                  Add a new option leaf with initialized sufficient statistics
17:               end for
18:            end if
19:         else
20:            if $optionCount_l < maxOptions$ then
21:               Compute $\overline{G}_l(X_i)$ for existing splits and (non-used) attributes
22:               Let $S$ be existing child split with highest $\overline{G}_l$
23:               Let $X$ be (non-used) attribute with highest $\overline{G}_l$
24:               Compute Hoeffding bound $\epsilon = \sqrt{\frac{R^2 ln(1/\delta')}{2n_l}}$
25:               if $\overline{G}_l(X) - \overline{G}_l(S) > \epsilon$ then
26:                  Add an additional child option to $l$ that splits on $X$
27:                  for all branches of the split do
28:                     Add a new option leaf with initialized sufficient statistics
29:                  end for
30:               end if
31:            else
32:               Remove attribute statistics stored at $l$
33:            end if
34:         end if
35:      end if
36:   end for
37: end for
```

employed, and setting a sufficiently loose secondary bound does indeed create further split options. δ' can be expressed in terms of a multiplication factor α, specifying a fraction of the original Hoeffding bound:

$$\delta' = e^{\alpha^2 \ln(\delta)} \tag{6.3}$$

Or equivalently:

$$\alpha = \sqrt{\frac{\ln(\delta')}{\ln(\delta)}} \tag{6.4}$$

For example, with the default parameter settings of $\delta = 10^{-7}$ and $\delta' = 0.999$ then from Equation 6.4, $\alpha \approx 0.0079$, that is, decisions to explore additional attributes are approximately 126 times more eager than the initial choice of attribute. Preliminary experiments found that a setting of $\delta' = 0.999$ works well in combination with the second overriding growth strategy discussed next.

The second main approach to controlling tree growth involves directly limiting the number of options that the tree will explore. Kohavi and Kunz tried a local limit, where they would not allow an option node to split more than five different ways. This thesis introduces a global limiting strategy that instead of limiting options *per node* it limits options *per example*. A combinatorial explosion is still possible with a locally applied limit, whereas a global limit prevents it altogether. Doing so requires more work to compute how many leaves are reachable at a particular point in the tree, and only allowing more options if it does not exceed the maximum allowed. Line 20 of Algorithm 11 performs the test that determines whether more options are possible, and assumes that the maximum reachability count ($optionCount_l$) of the node is available. Computing maximum reachability counts in an option tree is complicated—it not only depends on the descendants of a node but also on its ancestors, and their descendants too. To solve this problem an incremental algorithm was devised to keep track of maximum counts in each option node in the tree, listed in Algorithm 12. The worst-case complexity of this algorithm is linear in the number of nodes in the tree, as it could potentially visit every node once. All nodes start with an *optionCount* of 1. The operation for updating counts is employed every time the tree grows. Nodes are not removed from the tree, as pruning is not considered in this thesis. However, an operation for removing option nodes, that would be needed if the option trees were pruned, is included for completeness.

Interestingly, the arbitrary local limit of five options employed by Kohavi

Algorithm 12 Option counter update, for adding and removing options.

```
Procedure $AddOption(node, newOption)$:
$max \leftarrow node.optionCount$
if $node$ has children then
   $max \leftarrow max + 1$
end if
for all children of $newOption$ do
   $child.optionCount \leftarrow max$
end for
add $newOption$ as child of $node$
call UpdateOptionCount($node$,$newOption$)

Procedure $RemoveOption(node, index)$:
while there are options below $node$ do
   remove deepest option
end while
remove child at $index$ from node
call UpdateOptionCountBelow($node$,-1)
if $node$ has parent then
   call UpdateOptionCount($parent$,$node$)
end if

Procedure $UpdateOptionCount(node, source)$:
$max \leftarrow$ maximum $optionCount$ of $node$ children
$\delta \leftarrow max - node.optionCount$
if $\delta \neq 0$ then
   $node.optionCount \leftarrow node.optionCount + \delta$
   for all children of $node$ such that $child \neq source$ do
      call UpdateOptionCountBelow($child$,$\delta$)
   end for
   if $node$ has parent then
      call UpdateOptionCount($parent$,$node$)
   end if
end if

Procedure $UpdateOptionCountBelow(node, \delta)$:
$node.optionCount \leftarrow node.optionCount + \delta$
for all children of $node$ do
   call UpdateOptionCountBelow($child$, $\delta$)
end for
```

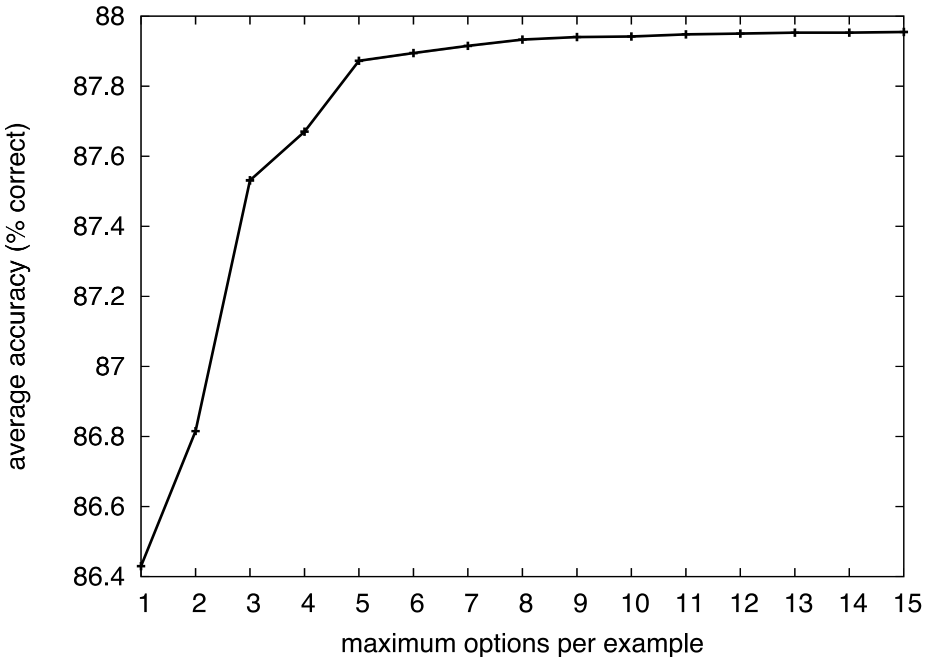

Figure 6.3: Average accuracy of Hoeffding option tree over many data sets versus the maximum number of options per example. Accuracies were estimated in unbounded memory.

and Kunz also seems to be a good choice for the global limit suggested here, at least when memory restrictions are not considered. Early experiments looked at the effect of the *maxOptions* parameter on smaller scale runs in unbounded memory. The average accuracy across many data sets is plotted in Figure 6.3, showing that prior to a maximum of five options there are significant accuracy gains, but after that point the accuracy gains diminish. The computational demands continue to rise with each additional option.

Other design decisions also help to limit tree growth. When searching for additional options, ties are not broken in the same manner as during the initial decision. Doing so would inevitably force options to be introduced when the bound is sufficiently small. Instead, this cuts down excessive options by only considering genuine contenders that emerge, those with positive gains. Another restriction is that an attribute will only be used once per option node, which reduces the possibility of multiple redundant splits, especially where numeric attributes are concerned.

Having internal nodes that also act like active leaves because they record sufficient statistics has serious implications for memory management. To reduce the memory requirements of internal nodes, they are deactivated as soon as further options are prohibited (line 32 of Algorithm 11). The memory management strategy needs to be adapted to cope with these costly nodes when memory is exhausted. Recall from Section 3.3 that each leaf has a measure of 'promise'. The least promising nodes are deactivated first, while the most promising are retained in memory the longest. The 'promise' is measured by the numbers of examples seen by the node since creation that are not correctly classified by the majority observed class. Three straightforward approaches were compared to decide the final method for experimental comparison:

1. Each active node is treated equally, regardless of whether it is internal or a leaf. This has the most potential to stall growth because too many internal leaves that are highly promising will prevent the tree from growing at the leaves. Also, nodes near the top of the tree such as the root node are likely to always look promising due to high numbers of misclassified examples at that point.

2. Internal nodes are penalized by dividing their promise by the number of local options. This reduces focus on areas that have already explored several options.

3. Leaf nodes are always more promising than internal nodes. This means that when reclaiming memory the internal nodes are always the first to be removed, in order of promise. Once all internal nodes are removed, the leaf nodes will start being deactivated, in the same manner as standard Hoeffding trees.

The third option fared the best, and is the strategy used in experiments. It appears that seeking more options at the expense of pursuing the existing ones is harmful overall to tree accuracy.

The *maxOptions* parameter plays a significant role overall when memory is limited, because it adjusts the tradeoff between the numbers of options that are explored and the depth at which they can be explored. For the final experiments in Chapter 7 *maxOptions* was altered to be consistent in comparison to the other methods, where a setting of m is functionally equivalent to an ensemble of m distinct trees.

6.3 Realistic Ensemble Sizes

In the batch setting, a typical ensemble may consist of one hundred or more base models. This is because the primary goal is increasing accuracy, and the more models combined the greater the potential for this to occur. The memory and computational implications of combining this many models in the batch setting are not a major concern, as typically training sizes are small, and prolonged training and testing times are acceptable. If it were too demanding, there is always the option to use a simpler base algorithm or reduce the ensemble size.

In the data stream setting the extra demands of combining models must be carefully considered. If a particular algorithm is barely able to cope with a high-speed stream, then even adding a second model in parallel will not be feasible. For every additional model included, the gains must be weighed against the cost of supporting them. An ensemble of one hundred models would process examples approximately one hundred times slower, and memory restrictions would require each model to be one hundred times as small. In 100KB of memory this would rely heavily on memory management, which would endeavour to keep each model under 1KB in size. If sensible models can be induced in such conditions, the simplistic models may not work as effectively in combination than more sophisticated ones. For these reasons, the ensemble sizes experimented with in the next chapter are restricted to smaller values of ten, five or three models.

Such small ensemble sizes are not necessarily a hindrance to ensemble performance. The preliminary investigation into restricting the effective number of models in option trees, Figure 6.3, suggested that anything on the order of five models is close to optimal, and that higher numbers of models show limited improvement. This was without considering the added pressure of memory limits, which in some conditions may favour even smaller combinations. The ensemble size of three especially will be an interesting study of whether small combinations can show a useful effect.

6.4 Summary

The theory behind successful methods of improving accuracy in the batch setting have been reviewed—the ensemble methods bagging and boosting, and a generalized tree representation offering similar benefits, the option tree. Application of these methods to the data stream setting has been considered. The following chapter compares three methods experimentally, to see whether they can outperform a single Hoeffding tree. Bagging and boosting have been implemented as suggested by Oza and Russell [102], and are compared to a novel algorithm for inducing option trees using Hoeffding bounds. Three ensemble configurations have been chosen for testing each method; three, five or ten trees in combination.

Chapter 7

Ensemble Evaluation

The same evaluation methodology from Chapter 2 used throughout the thesis is employed to compare the candidate methods of improving decision tree accuracy. A total of ten algorithm variations are compared:

HTNBA Single Hoeffding tree with adaptive Naive Bayes prediction (Section 5.4).
Used as the base method for bagging and boosting.

BAG3/5/10 Oza and Russell's online bagging (Section 6.2.1).
Ensemble of three, five or ten HTNBA trees.

BOOST3/5/10 Oza and Russell's online boosting (Section 6.2.2).
Ensemble of three, five or ten HTNBA trees.

HOT3/5/10 Hoeffding option tree (Section 6.2.3).
Maximum of three, five or ten options per example.
Settings same as HTNBA including adaptive Naive Bayes prediction.
Secondary split confidence $\delta' = 0.999$

As previously, there are three memory-limited environments (100KB/sensor, 32MB/handheld, 400MB/server), 19 synthetic data sets, a limit of ten hours per training run, and testing via one million test examples. Training and testing speeds are listed as percentages of the full generation speeds unique to each data set as measured in Section 2.6.

7.1 Results

Detailed per-data set results of the experiments are available in Appendix A.3. To simplify comparison, averages across all data sets are compiled in Table 7.1.

Table 7.1: Final results averaged over all data sources comparing ensemble methods.

method	accuracy (%)	training examples (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	(average) tree depth	training speed (%)	prediction speed (%)
100KB memory limit / sensor								
HTNBA	85.44	29	0	8.65	11.9	12	67	82
BAG3	85.82	6	0	7.66	10.4	7.8	54	67
BAG5	84.02	4	0	6.78	9.10	6.3	45	60
BAG10	78.34	2	0	5.49	7.40	4.6	37	52
BOOST3	84.63	7	0	7.61	10.4	8.0	46	61
BOOST5	83.67	4	0	6.82	9.30	6.5	37	53
BOOST10	77.14	2	0	6.97	9.20	4.6	31	44
HOT3	86.34	15	0	8.83	11.5	12	54	68
HOT5	86.28	14	0	9.15	11.8	11	52	65
HOT10	85.90	14	0	9.60	12.1	11	52	63
32MB memory limit / handheld								
HTNBA	90.51	871	73.4	670	1106	24	14	65
BAG3	90.48	1025	34.1	1714	2539	20.0	16	50
BAG5	90.33	998	21.2	2064	3017	18.9	15	41
BAG10	90.34	825	13.0	2380	3458	17.6	16	30
BOOST3	89.38	948	15.9	1994	3085	21.3	14	43
BOOST5	89.60	1015	6.07	2240	3500	20.9	15	34
BOOST10	89.33	814	1.81	2445	3774	19.4	14	23
HOT3	90.66	792	64.0	944	1481	23	13	55
HOT5	90.72	750	61.5	1005	1575	23	12	51
HOT10	90.70	691	59.6	1081	1687	22	11	49
400MB memory limit / server								
HTNBA	90.70	463	489	46.7	828	28	6	53
BAG3	90.73	332	917	252	1772	23.7	5	34
BAG5	90.68	257	998	672	2482	22.2	4	27
BAG10	90.79	173	1046	1476	3678	20.6	3	19
BOOST3	89.77	156	985	869	2987	24.8	4	34
BOOST5	90.01	109	1062	1492	4188	23.2	3	25
BOOST10	89.93	79	1077	2994	6550	21.3	3	18
HOT3	90.85	377	580	73.9	1028	26	5	40
HOT5	90.94	344	608	86.1	1092	25	4	37
HOT10	90.96	292	609	133	1177	25	4	34

The general differences between memory environments follow the same trends seen in previous experiments—the most training examples are processed in 32MB of memory because once again the 100KB environment stops early once there is insufficient memory to continue growth, and the 400MB environment is slower due to deeper trees with many more active nodes to maintain.

The average accuracy figures clearly separate the methods. The HOT methods are the most accurate on average across all three environments, followed by the BAG methods. The BOOST methods are the least accurate on average, in all cases worse than the average accuracy of a single HTNBA tree.

It is interesting to study the accuracy trends when the number of trees, or equivalently the number of options, is increased in fixed memory limits. In the 100KB/sensor environment, higher ensemble sizes tend to be lower in accuracy, with ensembles of three trees being the most accurate size regardless of ensemble type. This trend is perhaps reflective of the size/accuracy tradeoff predicted in very limited memory. The 100KB is simply not enough to support accurate combinations of more than just a few trees. In the 32MB/handheld environment the middle of the range size of five trees tends to be best, the most accurate size for boosted and option trees, and of the BAG methods only marginally worse than ten bagged trees. In the 400MB/server environment the generous memory allowance creates opportunities for the ten tree ensembles to do best, apart from boosting where BOOST5 is more accurate than BOOST10 on average.

In the 32MB and 400MB environments, the HOT methods tend to have smaller models than the other ensemble methods, with fewer tree nodes in total than either bagged or boosted trees. This is a positive result for the option tree representation. Boosted trees tend to have the most nodes on average.

The average accuracy figures do not, however, convey the whole situation. Tables 7.2-7.4 give a detailed per-data set accuracy breakdown of a single HTNBA tree against the mid-range ensembles (five trees/options). In Table 7.2 the BAG5 method does not compete well in 100KB of memory, but wins many times in the other environments. In Table 7.3 the BOOST5 method struggles to show any gain over a single tree. In Table 7.4, HOT5 generally outperforms HTNBA, but is not so convincing in the 100KB environment.

Table 7.5 directly compares BAG5 with HOT5. In 100KB of memory the option tree method is clearly superior. In the higher memory environments, HOT5 is actually worse than BAG5 in most cases. The main exception is the

Table 7.2: HTNBA vs BAG5 accuracy (%).

method→	HTNBA			BAG5		
	memory limit			memory limit		
dataset	100KB	32MB	400MB	100KB	32MB	400MB
RTS	**96.92**	99.99	99.99	84.84	**100.00**	99.99
RTSN	**74.91**	**78.49**	78.44	69.82	78.48	**78.49**
RTC	**61.22**	**83.10**	**83.84**	54.13	79.66	81.90
RTCN	**53.60**	**62.26**	**63.19**	52.96	60.06	62.29
RRBFS	**88.43**	93.60	93.84	87.68	**93.93**	**94.29**
RRBFC	**91.19**	98.85	98.95	76.66	**99.47**	**99.56**
WAVE21	**81.23**	84.80	85.66	81.01	**85.19**	**86.14**
WAVE40	**81.20**	84.52	85.52	80.30	**85.06**	**85.98**
LED	**73.96**	**74.02**	**73.98**	73.35	73.98	73.97
GENF1	95.07	95.06	**95.05**	95.07	**95.07**	95.03
GENF2	78.30	94.05	94.05	**92.18**	**94.11**	**94.09**
GENF3	97.49	**97.52**	**97.51**	**97.51**	97.51	97.50
GENF4	**93.86**	94.68	94.65	91.61	94.68	**94.66**
GENF5	72.10	92.41	92.40	**78.36**	**92.83**	**92.83**
GENF6	**92.09**	93.31	93.29	90.64	**93.34**	**93.32**
GENF7	**96.53**	96.82	96.80	96.18	**96.84**	**96.83**
GENF8	**99.41**	99.42	99.42	99.40	**99.43**	99.42
GENF9	**95.98**	96.81	96.78	94.86	**96.82**	**96.83**
GENF10	**99.89**	99.89	99.89	99.88	99.89	99.89
average	85.44	90.51	90.70	84.02	90.33	90.68

Table 7.3: HTNBA vs BOOST5 accuracy (%).

method→	HTNBA			BOOST5		
	memory limit			memory limit		
dataset	100KB	32MB	400MB	100KB	32MB	400MB
RTS	**96.92**	99.99	**99.99**	88.38	99.99	99.98
RTSN	**74.91**	**78.49**	**78.44**	68.59	78.39	78.34
RTC	**61.22**	**83.10**	83.84	59.23	82.39	**84.24**
RTCN	**53.60**	**62.26**	**63.19**	53.55	59.27	60.12
RRBFS	**88.43**	**93.60**	**93.84**	87.13	93.01	93.30
RRBFC	**91.19**	98.85	98.95	79.87	**99.19**	**99.30**
WAVE21	**81.23**	**84.80**	**85.66**	80.93	84.49	85.37
WAVE40	**81.20**	**84.52**	**85.52**	80.48	84.32	85.06
LED	**73.96**	**74.02**	**73.98**	73.87	73.97	73.92
GENF1	**95.07**	**95.06**	**95.05**	93.72	90.93	93.23
GENF2	78.30	**94.05**	**94.05**	**86.18**	92.01	92.48
GENF3	**97.49**	**97.52**	**97.51**	96.40	95.47	96.41
GENF4	**93.86**	**94.68**	**94.65**	93.26	93.17	93.31
GENF5	**72.10**	**92.41**	**92.40**	68.72	91.80	91.66
GENF6	**92.09**	**93.31**	**93.29**	89.24	92.19	92.07
GENF7	**96.53**	**96.82**	**96.80**	95.93	96.14	96.02
GENF8	**99.41**	**99.42**	**99.42**	99.36	99.34	99.28
GENF9	**95.98**	**96.81**	**96.78**	94.93	96.43	96.21
GENF10	**99.89**	**99.89**	**99.89**	99.88	99.87	99.87
average	85.44	90.51	90.70	83.67	89.60	90.01

Table 7.4: HTNBA vs HOT5 accuracy (%).

method→	HTNBA			HOT5		
	memory limit			memory limit		
dataset	100KB	32MB	400MB	100KB	32MB	400MB
RTS	**96.92**	99.99	99.99	95.85	99.99	99.99
RTSN	**74.91**	**78.49**	**78.44**	73.71	78.48	78.38
RTC	61.22	83.10	83.84	**64.79**	**84.22**	**84.92**
RTCN	53.60	62.26	63.19	**54.64**	**63.88**	**65.61**
RRBFS	**88.43**	93.60	93.84	87.93	**93.83**	**94.18**
RRBFC	**91.19**	98.85	98.95	78.63	**99.20**	**99.22**
WAVE21	81.23	84.80	85.66	81.23	**85.12**	**86.03**
WAVE40	**81.20**	84.52	85.52	81.14	**84.95**	**85.86**
LED	**73.96**	**74.02**	**73.98**	73.91	73.96	73.94
GENF1	**95.07**	95.06	95.05	95.06	95.06	95.05
GENF2	78.30	94.05	94.05	**93.47**	**94.09**	**94.06**
GENF3	**97.49**	**97.52**	**97.51**	97.48	97.51	97.50
GENF4	**93.86**	**94.68**	**94.65**	93.80	94.67	94.63
GENF5	72.10	92.41	92.40	**84.25**	**92.71**	**92.80**
GENF6	**92.09**	93.31	**93.29**	91.95	**93.33**	93.28
GENF7	**96.53**	96.82	**96.80**	96.38	**96.83**	96.79
GENF8	**99.41**	99.42	99.42	99.40	99.42	99.42
GENF9	**95.98**	96.81	96.78	95.77	**96.82**	**96.79**
GENF10	**99.89**	99.89	99.89	99.88	99.89	99.89
average	85.44	90.51	90.70	86.28	90.74	90.97

Table 7.5: BAG5 vs HOT5 accuracy (%).

method→	BAG5			HOT5		
	memory limit			memory limit		
dataset	100KB	32MB	400MB	100KB	32MB	400MB
RTS	84.84	**100.00**	99.99	**95.85**	99.99	99.99
RTSN	69.82	78.48	**78.49**	**73.71**	78.48	78.38
RTC	54.13	79.66	81.90	**64.79**	**84.22**	**84.92**
RTCN	52.96	60.06	62.29	**54.64**	**63.88**	**65.61**
RRBFS	87.68	**93.93**	**94.29**	**87.93**	93.83	94.18
RRBFC	76.66	**99.47**	**99.56**	**78.63**	99.20	99.22
WAVE21	81.01	**85.19**	**86.14**	**81.23**	85.12	86.03
WAVE40	80.30	**85.06**	**85.98**	**81.14**	84.95	85.86
LED	73.35	**73.98**	**73.97**	**73.91**	73.96	73.94
GENF1	**95.07**	**95.07**	95.03	95.06	95.06	**95.05**
GENF2	92.18	**94.11**	**94.09**	**93.47**	94.09	94.06
GENF3	**97.51**	97.51	97.50	97.48	97.51	97.50
GENF4	91.61	**94.68**	**94.66**	**93.80**	94.67	94.63
GENF5	78.36	**92.83**	**92.83**	**84.25**	92.71	92.80
GENF6	90.64	**93.34**	**93.32**	**91.95**	93.33	93.28
GENF7	96.18	**96.84**	**96.83**	**96.38**	96.83	96.79
GENF8	99.40	**99.43**	99.42	99.40	99.42	99.42
GENF9	94.86	96.82	**96.83**	**95.77**	96.82	96.79
GENF10	99.88	99.89	99.89	99.88	99.89	99.89
average	84.02	90.33	90.68	86.28	90.74	90.97

RTC/RTCN data sets where HOT5 is ahead by such a margin that overall the average accuracy is higher (see Section 7.2 for detailed analysis). This is a case where the average values are misleading. On the whole BAG and HOT are fairly similar with different strengths and weaknesses. One of the option trees clear strengths is memory efficiency—it is better than bagging in the most limited memory environment, and in general it uses fewer tree nodes while achieving similar accuracy.

Learning curves for all methods in 400MB are plotted on the left side of Figures 7.1-7.5. The poor performance of boosting stands out most on the GENF1-GENF10 data.

Alongside each learning curve plot are three plots displaying the distribution of extra options present in the final option trees. Plotted on the y-axis is the number of additional options introduced, and on the x-axis is the depth of the options. The depth of zero represents the root of the tree, and the x-axis is scaled to accommodate the full depth of the tree. There are several cases where options were added at the root: RTC, RTCN, LED, WAVE21 and WAVE40. In all other cases there were no additional options at the root. It is uncertain why this is the case, the maximum number of options are exceeded at the root before it is ready to introduce options. This may suggest that estimation at the root tends to be more reliable, or is perhaps due to the data sets having only a single attribute that splits well at the root.

Recall that Kohavi and Kunz [84] concluded that options nearer the root are more valuable. In general the option trees induced had their options present in the upper half of the tree. This is a side effect of the option limits and memory management. Options are allowed during early growth while the tree is shallow, but later when option limits and memory management prohibit introduction of further options the trees will continue to deepen, so it is not surprising that they tend to occur nearer the root of the tree.

7.2 Discussion

Some of the results are puzzling—why does boosting do so poorly, and why does the option tree stand out against bagging on the RTCN data source? To help gain greater understanding of differences between the methods, a bias/variance decomposition analysis is employed. The bias and variance components of the error were estimated on two selected data sets in 400MB of memory. This involved training the algorithms ten times on independent streams of

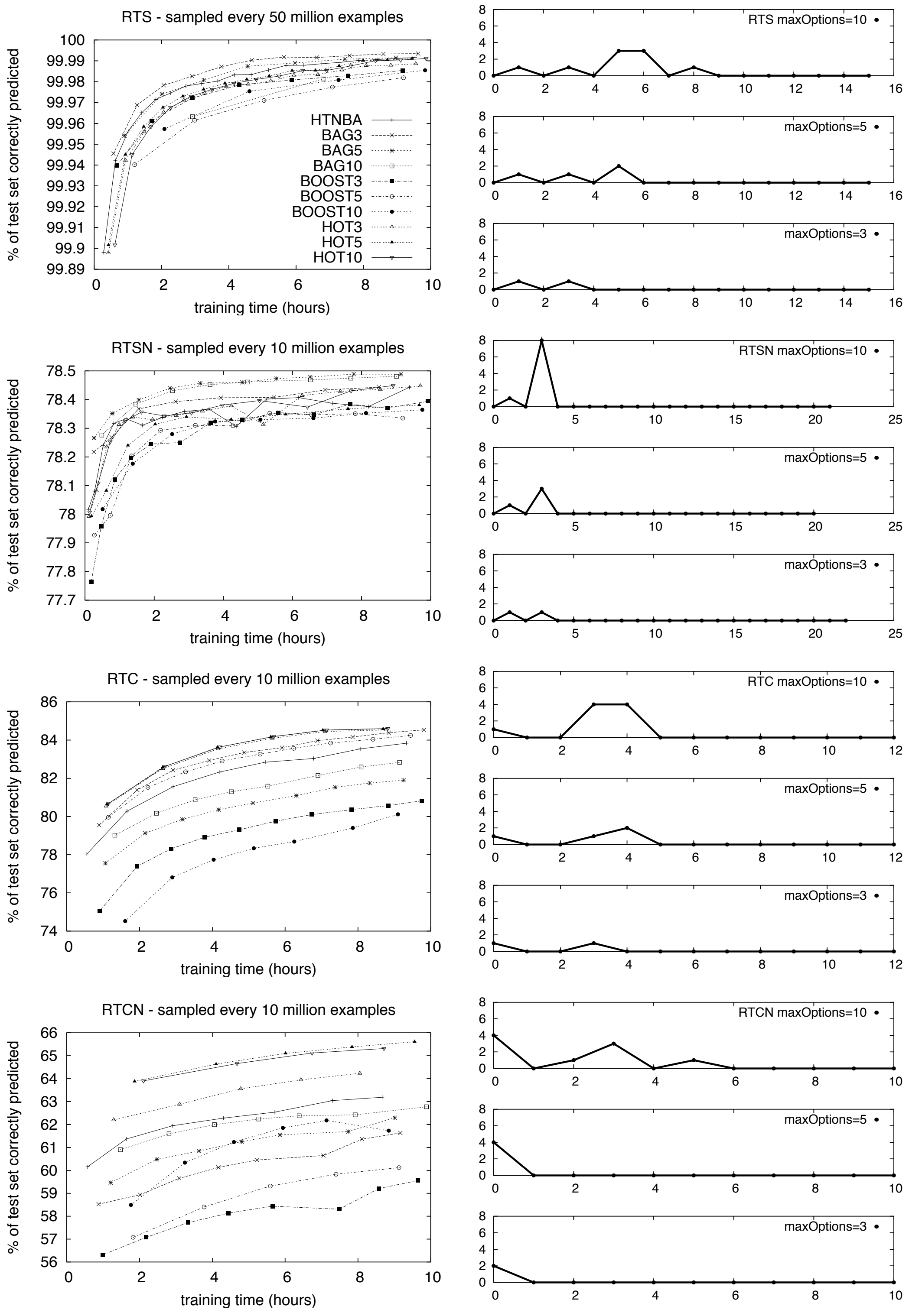

Figure 7.1: Part 1 of learning curves for ensemble methods (left) and HOT option distribution (right) in 400MB memory limit.

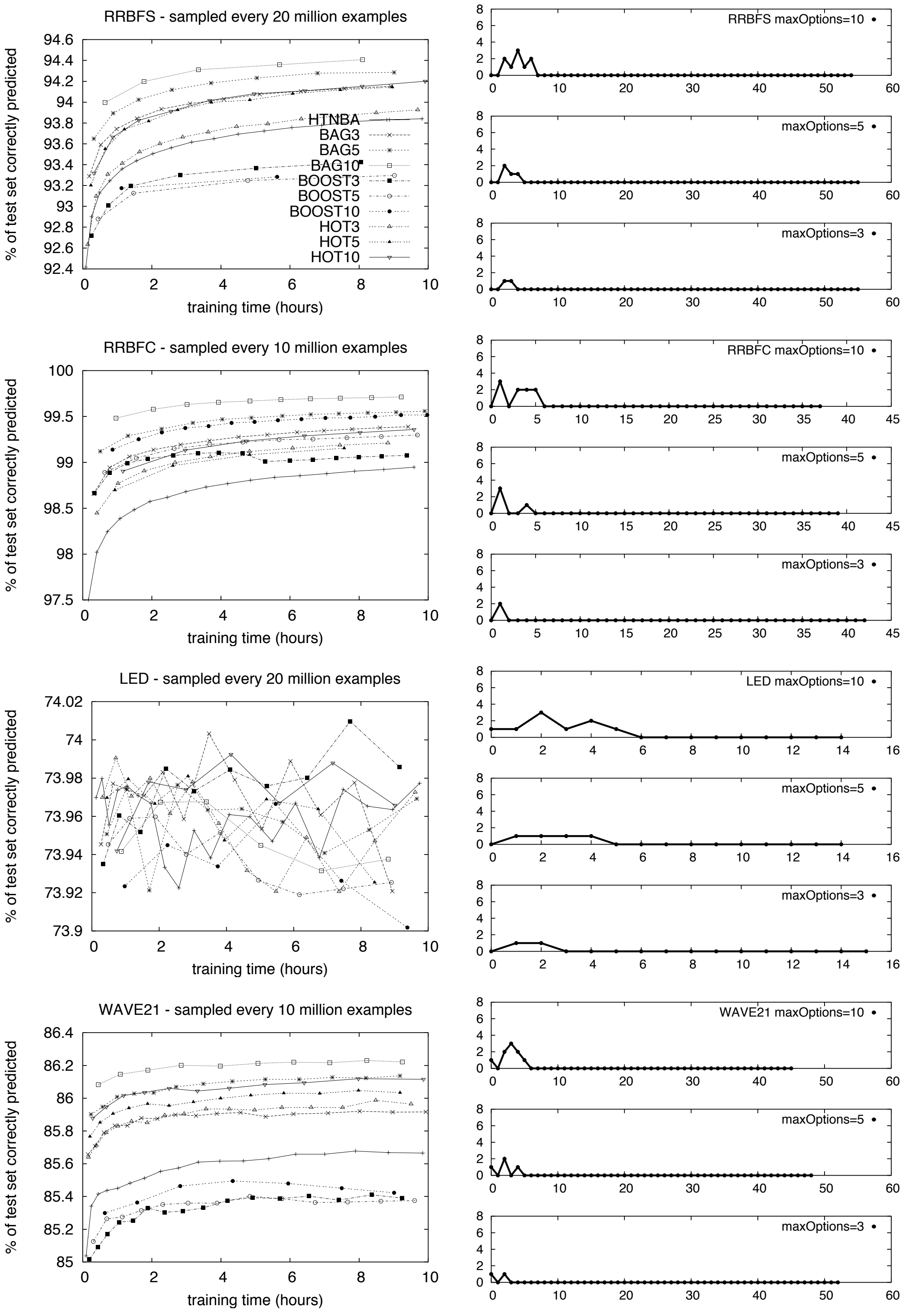

Figure 7.2: Part 2 of learning curves for ensemble methods (left) and HOT option distribution (right) in 400MB memory limit.

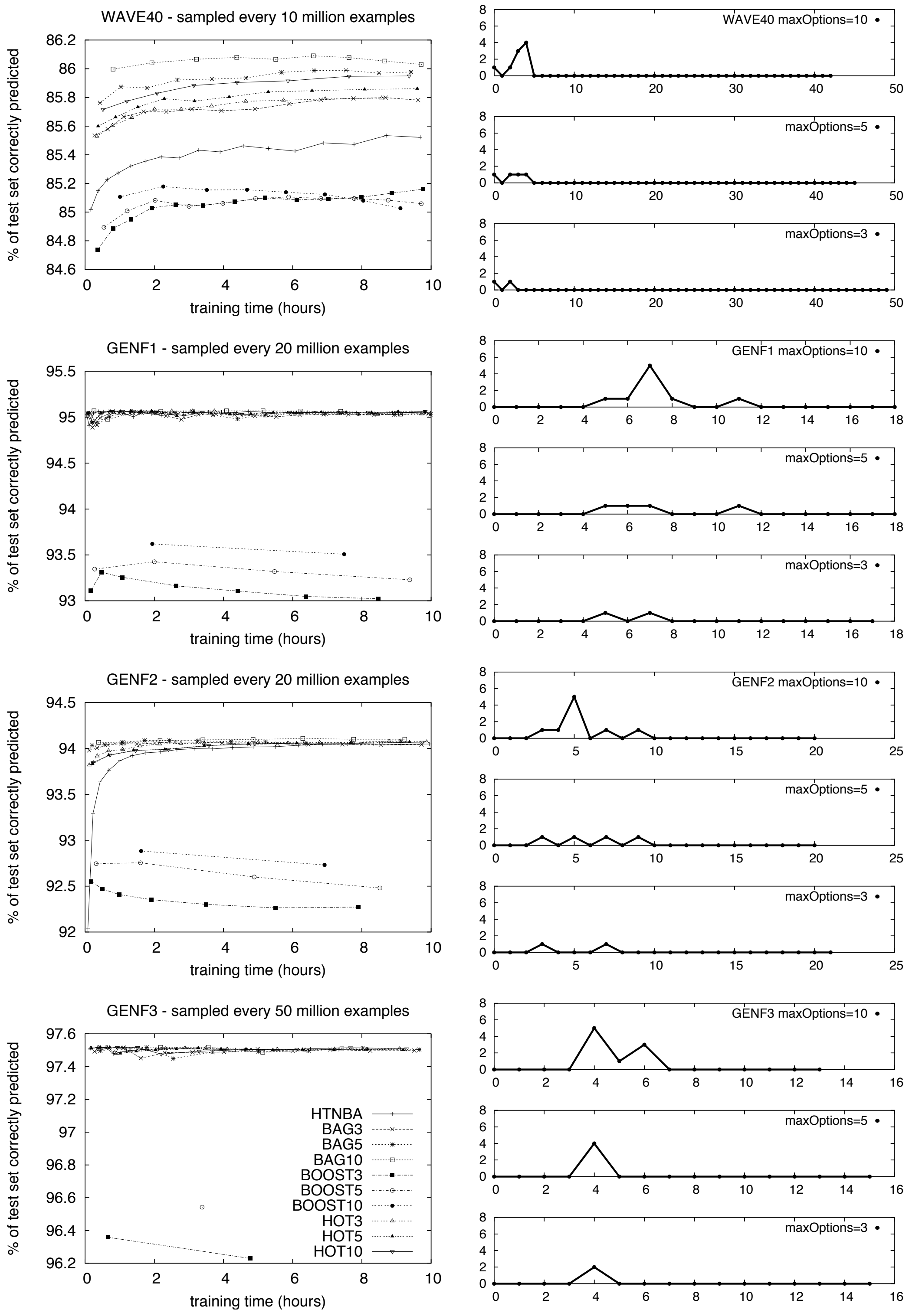

Figure 7.3: Part 3 of learning curves for ensemble methods (left) and HOT option distribution (right) in 400MB memory limit.

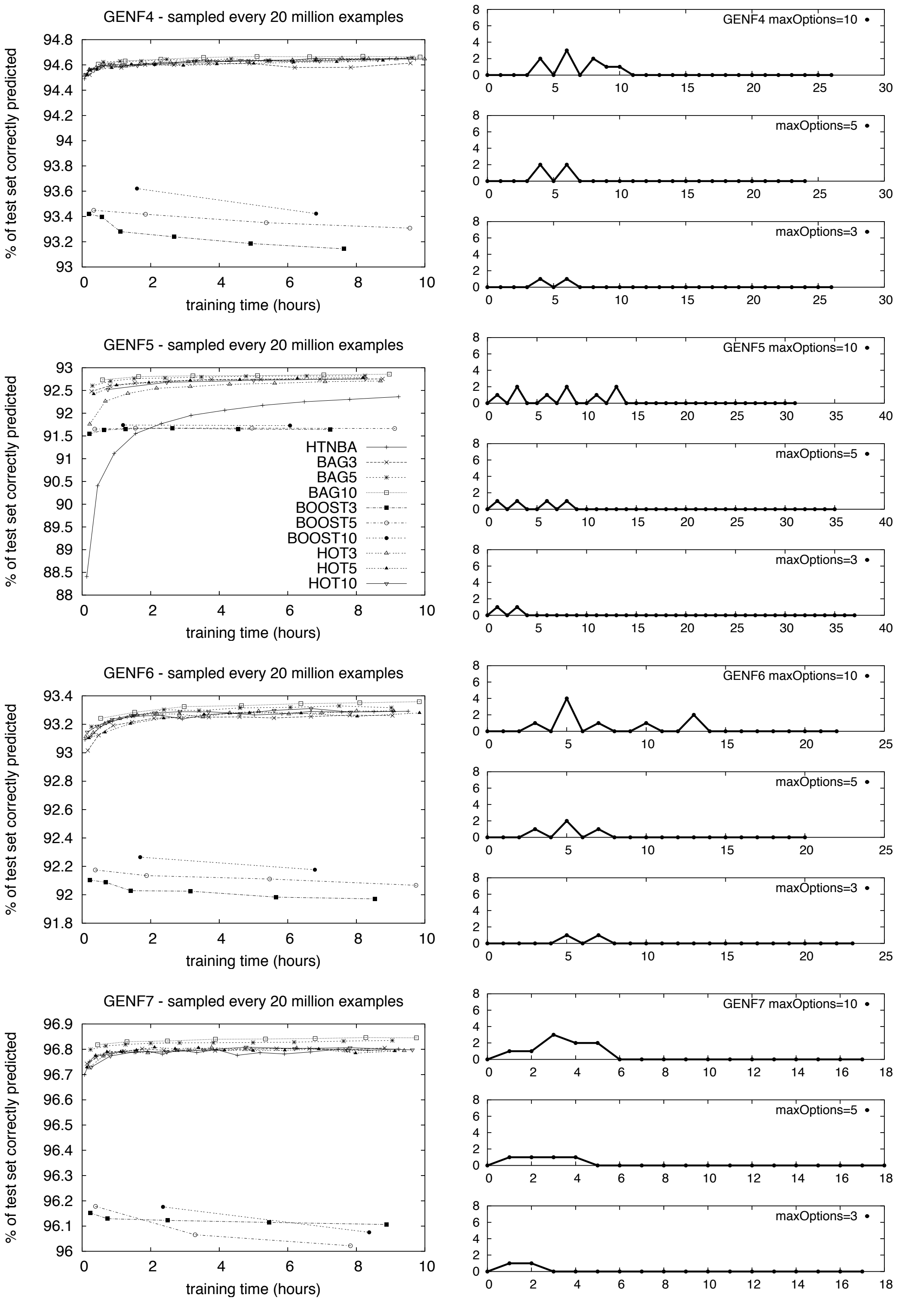

Figure 7.4: Part 4 of learning curves for ensemble methods (left) and HOT option distribution (right) in 400MB memory limit.

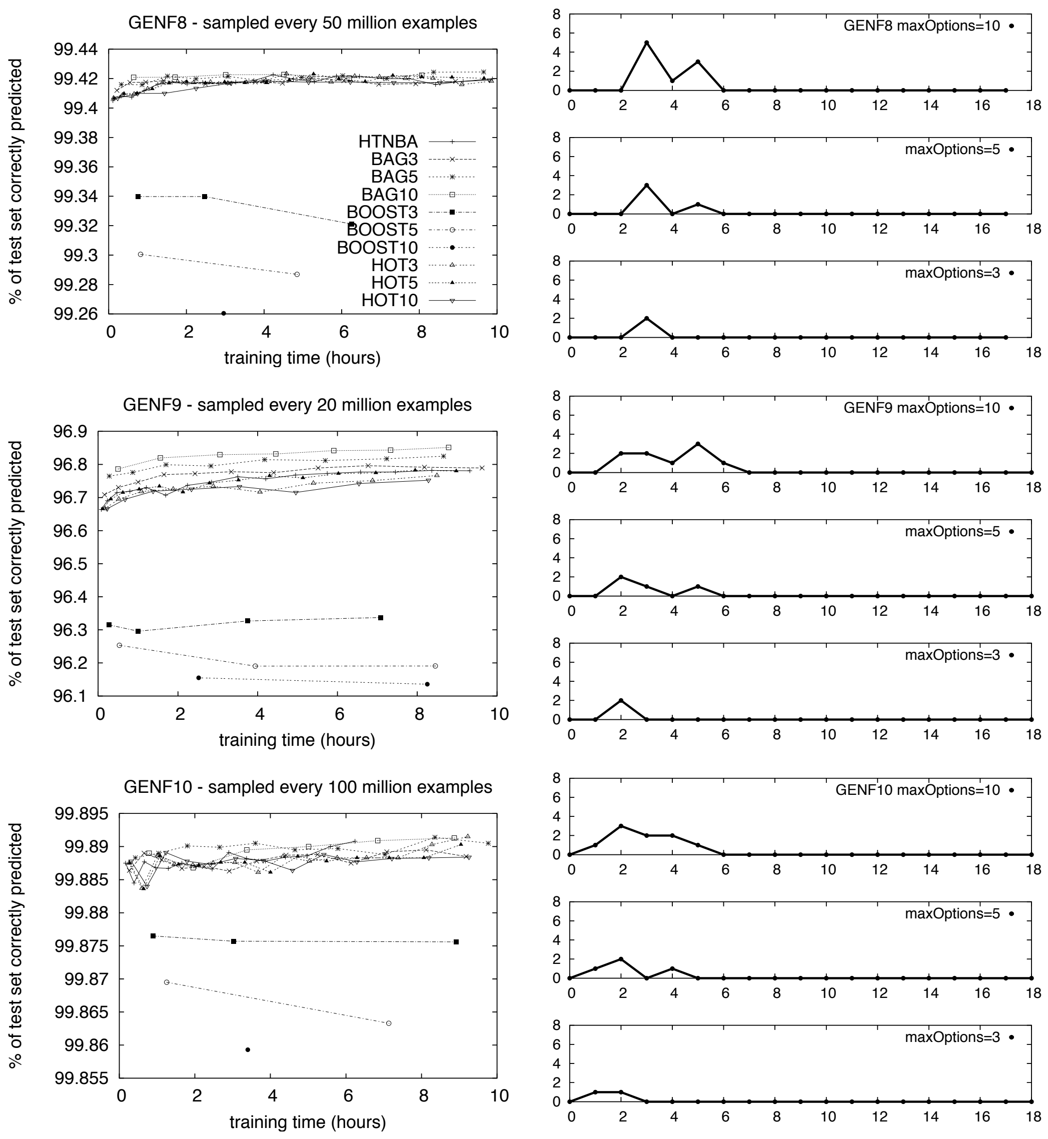

Figure 7.5: Part 5 of learning curves for ensemble methods (left) and HOT option distribution (right) in 400MB memory limit.

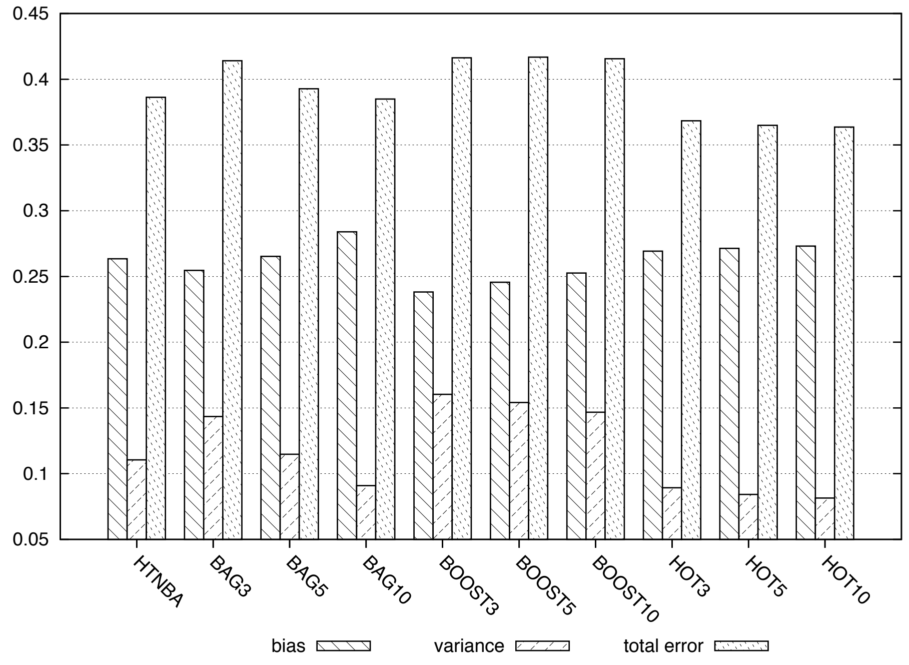

Figure 7.6: Bias variance decomposition on RTCN.

ten million examples each, and testing on ten thousand held-out test examples. From this procedure the bias and variance were computed according to Kohavi and Wolpert [86]. A smaller scale experiment was required to collect these results. Although not training each model as much as in the final experiments, repeating the procedure ten times makes it a time demanding process. Even though the results are based on smaller training sets they are still expected to expose meaningful differences between the methods.

Figure 7.6 shows the bias/variance decomposition estimates on the RTCN data. On this particular data set in Section 7.1, BAG did substantially worse than HOT. The bias of BAG and HOT is similar on this problem, close to the bias of a single tree, although the bias is relatively constant between the HOT sizes whereas it noticeably increases with the size of the BAG ensembles. The main difference lies in the estimated variance, which is reduced the most by HOT and also substantially reduces with larger BAG ensembles while not dropping quite as low. The combined effect of the components of error leads to an overall error in favour of HOT, which has the lowest error of all methods. BAG error falls closer to that of a single HTNBA tree as more trees are combined, but as reflected in the final accuracy figures in Table 7.2 it is not as accurate. On this particular data set, which is one of the most complex benchmark data sets tested, bagging has high variance with three trees and high bias with

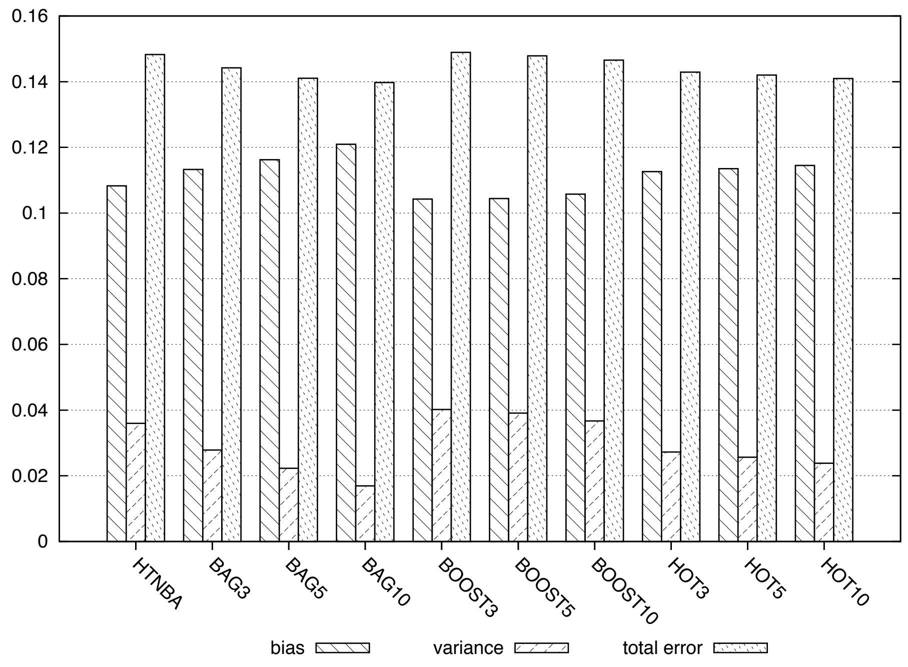

Figure 7.7: Bias variance decomposition on WAVE21.

ten. Either way it does not compete with the Hoeffding option tree which maintains a steady variance, and significantly lower bias that reduces slightly with additional options. Boosting manages to substantially reduce bias, much more than the other methods, but has consistently high variance, with overall error that is worse than the other methods.

Figure 7.7 displays the estimated bias/variance on the WAVE21 data. In experiments involving 32 and 400 megabytes of memory, BAG was generally superior to HOT on this problem (Table 7.5). Once again the bias of BAG increases with extra trees compared to the steady bias of HOT, both of which have more bias than a single tree. This time as the number of trees in the bag increases, BAG manages to reduce variance more than any other method, which results in the lowest errors overall when five or ten trees are combined. HOT is competitive but does not reduce the variance as significantly. Boosting excels at reducing bias, the lowest of the methods, but is less accurate again due to high variance.

Bias/variance decomposition analysis on these two data sets suggests the following trends:

1. Both bagging and option trees are successful at reducing variance, which explains their ability to reduce error beyond a single tree.

2. In terms of differences between bagging and option trees, adding trees to bagging tends to have stronger effects on bias and variance, whereas option trees are much more steady as additional options are introduced. On certain data sets, option trees can reduce the variance considerably more than bagging, but the majority of the time bagging has slightly outperformed option trees in this regard.

3. Boosting performs poorly due to high variance.

The boosting result is disappointing considering that it offers so much promise in the batch setting. Transferring the generalization power of boosting from the batch setting to the data stream setting does not appear to be as trivial as one might think. The intuitive understanding of getting models to concentrate more on examples that are harder for previous models to classify gives the impression that it should be reproducible in the stream setting. After all, Breiman [13] set out to prove with arc-x4 that the "magic" of boosting does not lie in specific details.

The study of Brain and Webb [11] suggests that management of bias may be more important than managing variance on large data sets. Reducing bias is a difficult problem, boosting is the most successful of the tested methods at reducing bias, but fails to also reduce variance, so fails at reducing error more than the other methods. Boosting trees in the batch setting has a tendency to produce large trees, but in this setting growth has been restricted. Therefore, the failure may partly be due to lack of space, and also since the hypothesis space is enlarged, there is increased risk of choosing a poorer hypothesis.

Preliminary experimentation suggesting that boosting was not competing well with other methods motivated a search for an adaptation of boosting that is more successful. Attempted implementations included several direct *parallel* boosting adaptations of AdaBoost [44], including its confidence-rated version [109]. For example, following the half correct/half incorrect weighting observation exploited by Oza and Russell [102] and inspired by Schapire's original hypothesis boosting algorithm [107], one attempt was to boost by filtering examples—subsequent models in sequence would get a correctly classified example followed by an incorrectly classified one, with any examples not conforming to the desired pattern being discarded. Other attempts included arc-x4 [13], MadaBoost [31] and stream adaptations of the *alternating decision tree algorithm* [43]. A few *block* boosting approaches were also investigated. Successful application in the data stream setting was difficult to find despite

Breiman's suggestion that the essential formula lies only in the adapting, resampling and combining process.

Oza and Russell's algorithm was chosen as the boosting representative because it performed among the best, it compliments the online bagging implementation and has literature to support it as a successful method. Unfortunately, when Oza and Russell demonstrate the ability of their method they do so without mention of fixed memory limits. Their experimental results [102, 101] using Utgoff's *ITI* decision tree algorithm [118] as the base learner exhibit similar tendencies of online boosting competing poorly with online bagging. To improve the situation they try to bolster online boosting via "priming", which helps, but is not a solution explored by this thesis because it deviates from a purely stream-based solution. In their case they combine 100 trees, but note that in using ITI they are restricted to small data sets due to its poor scalability.

Deeper analysis of where the boosting attempts failed would often point towards problems with the example weighting process. Without a normalization step, it was observed that weights on particular examples in the stream grow uncontrollably, to the point where their magnitude exceeded the representational capacity of the machine. This problem is believed to represent a fundamental shortcoming of AdaBoost in streams. Domingo and Watanabe [31] made a similar observation, which is why their MadaBoost algorithm limits the magnitude of example weights. Experimenting with simple solutions such as this removed the symptoms but also diluted the boosting procedure, and did not show signs of the spectacular generalization promised by boosting.

Why does this weighting problem occur on streamed data and not in the batch case? In the batch setting the weights are calculated over a known and fixed set of examples. In a stream, new examples are being continually introduced, potentially representing areas of the concept space that have not been encountered before. It is difficult to estimate sensible weights for these examples in relation to previous examples and previous weights.

AdaBoost was conceived as a boosting by *sampling* method, but what is really needed for successful data stream application is a boosting by *filtering* method. As reviewed in Section 6.1.2, several researchers have attempted to supply such a method [31, 20], but the proposals so far have lacked the simplicity and elegance of AdaBoost. Intuitive and successful boosting in the data stream setting, that is on par with AdaBoost in the batch setting, remains an open problem.

The question of whether several models can provide benefit over a single model of the same size has been answered affirmatively, although the evidence suggests that plenty of memory is required to make ensembles worthwhile. In the highly restrictive 100KB environment a single tree is difficult to compete with, although the memory efficient option tree shows the most promise.

7.3 Summary

This experimental study looked at limited memory induction of decision tree ensembles from data streams. Three main methods were explored—bagging and boosting, two ensemble methods that are very popular in batch learning, and the third a powerful tree representation known as option trees. In empirical comparison bagging and option trees both showed ability to outperform a single tree, by significantly reducing the variance of individual trees, with varying strengths depending on the situation. Option trees were the most efficient in memory usage, showing the best ensemble performance in 100KB of memory, and showing significant improvement over bagging in a few 32MB/400MB cases. Overall bagging was able to outperform option trees in many other cases, a more consistent performer when sufficient memory is available. The general finding is that the more trees to be combined in an ensemble, the more memory needed to gain an advantage. No successful implementation of boosting for data streams was found. Boosting, or more specifically, AdaBoost, is perhaps the most powerful ensemble method known in batch learning. It is concluded that a truly successful and straightforward translation of AdaBoost to the data stream setting has yet to be discovered.

The change from a single HTNBA tree to a Hoeffding option tree with five options per example, HOT5, caused an increase in average accuracy from 88.88% to 89.31%, which is a relative improvement of 0.48%. This improvement reduced training speeds across all environments by an average of 21.84% and reduced prediction speeds by an average of 23.50%.

Within each environment the largest relative accuracy improvement was in 100KB, with a gain of 0.98%, which reduced training speed by 22.39% and prediction speed by 20.73%. In 32MB the relative accuracy gain was 0.23%, with speed reductions of 14.29% during training and 21.54% during prediction. In 400MB accuracy was increased by an average of 0.26%, which happens to be a larger gain for this environment than the 0.10% improvement found in the numeric attribute study. Training speed reduced by a third in 400MB, and prediction speed reduced by 23.50%.

Chapter 8

Conclusions

A central argument of this thesis is that the improvement of data stream classification algorithms requires a complete evaluation framework. The framework should train and test on sufficiently large amounts of data for realistic measurement, and should account for all three dimensions that are critical aspects of behaviour—error, space and time.

Evaluation will benefit from diverse and challenging benchmark data sets. Researchers need to be aware that there is a shortage of suitable real-world data sets for evaluating classification of streamed examples. The only alternative is artificial data generation, until realistically large and public real-world data sets become freely available.

The framework developed here is used to improve Hoeffding trees, a particular class of algorithm that is known to perform well. Improvements to the basic algorithm are quantified. The benefits of an extensive evaluation process are demonstrated for example in the findings relating to tree prediction. The smaller-scale studies of Gama et al. [50, 52] concluded that permanent use of Naive Bayes prediction in the leaves of decision trees provides general improvement. The broader evaluation performed by this thesis discovered cases where it actually performs worse. This was found by testing on more diverse data sources, with differing memory limits and involving substantially more examples. The findings enabled the proposal of a new adaptive algorithm that is shown to outperform other prediction approaches.

The demands of data stream problems will differ depending on circumstance. Nevertheless, data stream problems generally share four common demands. Algorithms demonstrated to best meet these demands will most easily apply to a wide range of data stream scenarios. The demands are:

1. The algorithm must process examples in a single pass, accepting each example in the order that it arrives. This capability can be tested by a framework that supplies training examples one-by-one to the algorithm, perhaps inspecting the model in between. A desirable property of an algorithm is low sensitivity to the order of examples. The sensitivity of an algorithm to example ordering can be tested by observing the variance in behaviour between training runs as order is manipulated.

2. The algorithm must work within limited memory. Algorithms that lack guaranteed bounds on memory usage are troublesome for data stream problems because they could fail. The most simple strategy available to any algorithm is to cease learning once memory is exhausted, however empirical evidence collected for this thesis suggests that it is worthwhile to find strategies enabling the algorithm to continue working, thus learn as much as possible in a more limited capability from further examples once memory is full. An evaluation framework can enforce this by monitoring the amount of memory that the algorithm uses, possibly aborting if the requirement is violated.

3. The algorithm must work in a limited amount of time, both per training example and per test example. As with Hoeffding trees, the time per example may be directly related to the size in memory, where a maximum size directly translates into maximum time needed to update and predict. This requirement is not as well defined as the others, because the acceptability of a solution is heavily dependent on the time demands of the intended application. A testing framework can measure the speed at which examples are processed, and could be told to abort if a target speed is not attained or start to drop excess examples.

4. The algorithm should be able to provide predictions for new examples at any point between training examples. A framework can test this ability by periodically requesting predictions, and can use this procedure to monitor performance over time.

The evaluation framework developed in Chapter 2 is capable of measuring how well an algorithm can meet the four demands of data stream classification. Practitioners facing a known problem can use the framework to help decide the best algorithm for their needs. They will be able to specify the hardware speed and memory limit before testing competing solutions.

To conduct a general study of algorithm behaviour and avoid having a specific scenario in mind, the range of deployment scenarios has been divided into three general cases: sensor/100KB, handheld/32MB and server/400MB. The base algorithm described in Chapter 3, the Hoeffding tree induction algorithm, is shown via the framework to be capable of meeting the demands, thus it is generally applicable to data stream problems.

Having such an appropriate framework enables:

1. Empirical results to be produced that are of a depth and scale beyond anything previously reported.

2. Algorithm varieties to be directly compared in terms of how well they perform at classifying data streams in the three dimensions of interest—error, space and time.

3. Production of interesting and sometimes surprising results that were previously unknown. The insight gained can motivate more algorithm development.

Having the sensor/handheld/server environment breakdown creates the opportunity to observe how algorithm suitability can differ depending on application. For example, a single Hoeffding tree with majority class prediction is hard to improve when only 100KB of memory is available. Any enhancement that increases the rate of memory consumption, such as functional leaf predictions or inducing ensembles of trees, rarely proved worthwhile in this environment. In contrast, the same enhancements would demonstrate significant improvement when more memory was available.

Having a state-of-the-art base algorithm and a means to quantify potential modifications has allowed the demonstration of improvements to Hoeffding tree induction. Chapter 3 looked at several minor improvements to the general algorithm. Chapter 4 studied the important but previously unresolved issue of how to best deal with continuous numeric attributes. The study in Chapter 5 tried improving the prediction strategy of the trees. Following the background investigated in Chapter 6, Chapter 7 concluded experimentation by measuring the performance implications of combining multiple Hoeffding trees into classifier ensembles.

The studies demonstrate progressive improvement of decision tree induction. The study of numeric approximation showed that small local approximation will globally outperform methods that are locally more accurate but

Table 8.1: Average accuracy gains, relative % change from previous.

enhancement	100KB	32MB	400MB	average
numeric	12.59%	0.41%	0.10%	3.91%
prediction	-0.08%	0.08%	0.44%	0.15%
ensemble	0.98%	0.23%	0.26%	0.48%
combined	13.60%	0.72%	0.81%	4.57%

Table 8.2: Average training speed losses, relative % change from previous.

enhancement	100KB	32MB	400MB	average
numeric	-13.75%	-17.65%	50.00%	-11.88%
prediction	-2.90%	0.00%	0.00%	-2.25%
ensemble	-22.39%	-14.29%	-33.33%	-21.84%
combined	-35.00%	-29.41%	0.00%	-32.67%

Table 8.3: Average prediction speed losses, relative % change from previous.

enhancement	100KB	32MB	400MB	average
numeric	-7.95%	-4.17%	-7.79%	-6.75%
prediction	1.23%	-5.80%	-25.35%	-9.50%
ensemble	-20.73%	-21.54%	-30.19%	-23.50%
combined	-26.14%	-29.17%	-51.95%	-35.44%

consume more memory. The study of prediction methods at the tree leaves looked at the standard and robust method versus a method that can significantly improve accuracy but that is also worse in several cases—experiments showed that adapting the leaves based on previous local performance can form a hybrid method that outperforms either approach on average. The study of further improvements to classification accuracy, involving ensembles and option trees, demonstrated that combinations of trees can outperform a single tree, but with the proviso that memory restrictions must be carefully considered.

Table 8.1 lists the three relative accuracy gains from the main improvements in isolation, and also the improvement when all three enhancements are combined. Note that the average column does not form a direct average of the three other columns due to the values being a relative change—in the case of the average it is a relative change from the previous average over the environments. Improving the numeric method increased average accuracy across all three environments by 3.91%, relative to before. Improving the prediction strategy increased relative accuracy by a further 0.15%, and using an option tree added an additional relative improvement of 0.48%. The total relative improvement of all enhancements combined is 4.57%. As this is averaged across all data sets and environments, there are individual cases where the improvement was more significant, such as WAVE21, where the accuracy improved by a relative percentage of 10.03%. Tables 8.2 and 8.3 show the speed costs of the enhancements. Improving the prediction method had the least average impact on training speed, while ensemble methods caused the largest reduction of both training and prediction speeds. Combining all of the enhancements reduced training speed by 32.67% on average and prediction speed by 35.44% on average.

In 100KB of memory, the total relative accuracy improvement was 13.60%, due mostly to the 12.59% gain by improved numeric approach. In this environment, the prediction enhancement decreased accuracy by 0.08%, while the option tree introduced an accuracy gain of 0.98%. The combined enhancements reduced training speed by an average of 35% and reduced prediction speed by an average of 26.14%.

In 32MB of memory, the total relative gain was 0.72% with the largest improvement also coming from the choice of numeric strategy. Training speed reduced due to the numeric and ensemble improvements, when all improvements were combined the speed reduction was 29.41%, along with a relative prediction speed reduction of 29.17%.

In 400MB, the total relative accuracy improvement was 0.81%, with the best gain coming from the prediction strategy enhancement. The numeric improvement managed to increase training speed in large memory, which counteracted the speed reduction of ensembles on average, meaning that the training speed remained constant on average when all of the enhancements were combined. The prediction speed, however, reduced by half.

A lesson to be learned from the studies is that when memory resources are limited it is often better to take a simple and less demanding approach, one that sacrifices accuracy at the local level, because it is likely that in causing less interruption to the global induction process that a more accurate model overall will be produced.

8.1 Contributions

The contributions of this thesis to the field of data stream classification can be divided into several significant areas:

1. Evaluation Framework

   Although the basic procedure is not new, similar in process to Dominogs and Hulten [32], the survey of work in Section 2.1 suggests that such rigorous evaluation is rare in practice. In particular, the crucial element of memory management is often overlooked. The framework suggested by this thesis is motivated by ensuring practical and realistic measurement of data stream classification algorithms. The usefulness of the framework is clearly demonstrated by the outputs reported in this thesis. Besides the general procedure, the framework includes a suggested division into three representative usage scenarios, and a suite of synthetic data generators. The hope is that progress in the field can be made by adopting similar practices.

2. Algorithm Development

   The Hoeffding tree induction algorithm is enhanced in several ways not suggested before.

   - Multi-class Gaussian approach to numeric approximation (Section 4.2.4).
   - Adaptively chosen majority class/Naive Bayes prediction strategy (Section 5.3).
   - Hoeffding Option Tree induction algorithm, including a novel approach to limiting options globally in a tree (Section 6.2.3).
   - Memory management implementation details (Section 3.3), including speed optimizations (Section 3.4.1).
   - Universal skewed-split prevention (Section 3.2.6).

3. Empirical evidence

   The results reported in this thesis represent a scale—in numbers of training examples, testing examples, memory limits and data sources—that is beyond anything else previously reported. Direct comparisons are provided between many competing methods that have not been experimentally compared before.

   - Comparison of numeric methods (Chapter 4)

  - An exhaustive *binary tree* method that retains all information in memory in order to make a precise decision is often outperformed by methods that deliberately lose information to conserve space.
  - The methods using the smallest amount of space perform the best. In particular the smallest method of all, a Gaussian approximation that explores ten local split possibilities, is found to be among the most competitive while being less susceptible to data order than other approaches.

- Comparison of prediction methods (Chapter 5)
  - *Naive Bayes* prediction used permanently, or after waiting for a fixed number of observations, sometimes performs worse than *majority class* prediction.
  - Adaptively choosing between *Naive Bayes* and *majority class* prediction per leaf, based on estimated accuracy, is shown to perform better on average than either method alone.
- Comparison of ensemble methods (Chapter 7)
  - Adaptations of *AdaBoost* to the data stream setting perform poorly.
  - An adaption of *bagging* and a novel *Hoeffding option tree* algorithm are both shown to be capable of outperforming a single Hoeffding tree. They do so by successfully reducing the *variance* of the trees.
  - Trends between memory limits and ensemble sizes suggest that as more trees are included in combination, more memory is needed to earn accuracy gains.
- Two previously suggested enhancements to Hoeffding tree induction were found to be mostly ineffective at raising accuracy—they are pre-pruning (Section 3.2.4) and poor attribute removal (Section 3.3.1).
- General differences between memory-restricted environments
  - In 100KB of memory the simplest/smallest methods fared best. Apart from simplifying the numeric approximation method it was difficult to find any enhancement that demonstrates a con-

vincing improvement over the basic algorithm of a single Hoeffding tree using majority class prediction.

- In 32MB of memory, accuracy was improved by adaptive Naive Bayes predictions, and improved further with bagging or option trees. More training examples could be processed in ten hours than in the other environments. The 100KB trees terminated early due to exhausting memory until no more growth was possible. The 400MB trees were slower at processing examples due to actively exploring many more possibilities for growth.
- In 400MB of memory the accuracy gains of more memory-intensive methods were most evident, such as ten trees/options combined via bagging or option trees. Despite being trained on less examples than the 32MB environment, the models were more complex and often the most accurate.

4. MOA (Section 3.4), an open-source Java software implementation of all algorithms and the evaluation framework—freely available at:

   http://sourceforge.net/projects/moa-datastream/

8.2 Future Work

Classification of high speed streams of examples is a discipline that is a very recent branch of machine learning, which itself is a field of research that is still in its infancy, so there is much left to explore. Avenues for future work have already been mentioned in several places in the thesis.

Chapter 1 explained that concept drift in data streams is beyond the scope of this thesis. For streams that occur over a substantial period there could be factors that cause the underlying concept to shift, rendering previous observations less relevant and causing the model to become outdated. The evaluation framework could be extended to test how well an algorithm responds to concept drift, by using synthetic generators that shift a concept and a test set that changes accordingly over time. Getting algorithms to successfully cope with concept drift is a very active area of research and typically involves revising old hypotheses formed by the model and focussing more on the most recent examples. In terms of decision trees this entails appropriate and adaptive pruning of the tree. *CVFDT* [73] is an extension of Hoeffding tree induction that is designed to manage concept drift. In terms of ensemble methods, the base members of the ensemble could be pruned to favour the models that best predict the recent trends in the stream, such as suggested by Chu and Zaniolo [25]. As soon as memory management becomes active it creates a bias towards certain areas of knowledge within a model. This bias could be tuned to treat more recent information as more valuable than older observations. In terms of option trees, each option can potentially explore concepts of varying relevance, so it may be possible to assign weights to options to adapt to shifting concepts, or prune the options that are estimated to be the least relevant.

Chapter 2 settled on an evaluation procedure that does not produce estimates of the significance between observed differences. One method of producing confidence intervals is to perform multiple runs and measure the variance. This is costly to perform when each run already requires substantial time to complete. A possible compromise may be the use of another less costly estimate of statistical significance, such as McNemar's test [26] which can be computed simply by looking at the agreement between competing algorithms on each test example. Chapter 2 also mentioned that it may be possible to use an exponentially decaying interleaved evaluation procedure, which may help to overcome the interleaved method's problem of over-penalizing early mistakes.

Chapter 3 raised issues about pre-pruning, as it did not appear to make a significant difference to experimental results. More investigation is needed to determine if and when pre-pruning is a worthwhile component of Hoeffding tree induction.

Chapter 4 revealed difficulties with the Gaussian numeric approximation method on two particular synthetic data sources, both of which are very similar in nature, GENF2 and GENF5. Other numeric approximation methods do not appear to be affected. The problem is overcome by ensemble methods, and interestingly, will also disappear if other conditions are changed, such as the splitting criterion or restricting the tree to two-way splits only. The tree does poorly because it favours splitting on an irrelevant multi-way nominal split over a relevant two-way numeric split. This result detracts from the otherwise competent performance of GAUSS10. The problem could be related to the problems seen with GAUSS100 which also makes poor split decisions on other data sets. It is speculated that these problems could be due to some unintentional bias being present in the split decisions, and perhaps a solution exists similar to the bias correction demonstrated by Quinlan [105].

Another suggestion relating to Gaussian numeric approximation is that the process could be made less susceptible to outliers by not tracking the absolute minimum and maximum values. Instead, considering points in the range that are several standard deviations away from the class approximations may be a more robust solution. It will also significantly save memory by storing only three values per attribute and class instead of five.

Chapter 5 did not explore the tailoring of memory management to Naive Bayes leaf predictions. It is possible that better decisions could be made by taking the accuracy of Naive Bayes models in account when computing the 'promise' of leaves. This information is recorded by the adaptive method but was not used when deciding which leaves to deactivate first. Interaction between poor attribute removal and Naive Bayes prediction could also be further investigated.

A significant result of Chapter 7 is that attempts at getting competitive results from *AdaBoost* in data streams were not successful. It is observed that the origins of AdaBoost lie in the boosting by *sampling* adaption of the PAC-learning framework, and as such is more suitable to the batch setting. It is believed that a boosting by *filtering* method is more likely to succeed in the data stream setting. The solution needed may be an adaptive boosting algorithm, that adapts to the errors of the base hypotheses like AdaBoost, but one that is expressed as a boosting by filtering method. The work of Domingo and Watanabe [31] and Bshouty and Gavinsky [20] claim to offer such a solution, although the former did not perform well when tested, and the latter lacks the simplicity and intuitive appeal of AdaBoost, while also lacking empirical evidence of performance.

Beyond the suggestions that have already been mentioned there are plenty of possibilities for further investigation. The evaluation framework was designed to test classification algorithms. Perhaps there are ways that it could be improved, with more efficient use of time or even more useful comparison between the behaviour of algorithms. Apart from classification, other machine learning problems face similar challenges from data streams. These include *regression*, predicting continuous numeric outputs, and *clustering*, learning from examples without guidance from class labels.

This thesis has focussed on evaluating and improving decision tree methods. There could be further enhancements that improve decision tree induction. Future research directions include the study of other classification algorithms besides decision trees. There are plenty of other machine learning methods that are successful in batch learning, some of which researchers have already tried to adapt to data streams, and others that are yet to be investigated. Use of a common evaluation framework will enable diverse methods to be compared and expose their relative strengths and weaknesses.

Appendix A

Detailed Result Tables

The figures in the following tables represent the final value that was measured at the completion of evaluation. The meaning of each table column is described below:

accuracy the percentage of examples that the tree was able to correctly predict from the one million examples reserved for testing

training examples the total number of examples that were used to train the tree before evaluation was complete

examples to full memory the number of training examples needed by the algorithm to grow the size of the tree to the memory limit (if the limit was in fact hit)

active leaves the number of active leaves in the tree (those that are capable of further splitting and Naive Bayes prediction)

inactive leaves the number of leaves that have been deactivated by the memory management scheme (these are no longer capable of splitting or Naive Bayes prediction)

total nodes the total number of nodes in the tree, including internal decision nodes

tree depth the depth of the tree—the length of the longest path from the root to a leaf

training speed the speed that the tree was able to train, expressed as a percentage of the maximum speed that examples can be generated from the data source, as measured in Section 2.6

prediction speed the speed with which the tree could make predictions on the test data, again expressed as a percentage of maximum stream speed

A.1 Numeric Methods

Table A.1: VFML10 method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	96.49	17	<1	0	8.12	11.4	8	76	84
RTSN	75.80	8	<1	0	8.47	11.5	8	83	87
RTC	61.37	3	<1	0	3.88	5.05	5	96	97
RTCN	53.63	3	<1	0	3.96	5.09	6	100	100
RRBFS	87.69	15	<1	0	4.99	9.97	18	69	84
RRBFC	87.84	6	<1	0	2.68	5.35	15	76	86
WAVE21	80.80	14	<1	0	4.00	7.99	14	86	91
WAVE40	80.28	9	<1	0	2.87	5.73	13	93	97
GENF1	95.07	15	<1	0	11.5	13.5	9	49	76
GENF2	93.94	10	<1	0	11.8	13.4	13	58	73
GENF3	97.52	55	<1	0	12.5	13.9	8	61	80
GENF4	94.46	5	<1	0	11.3	13.3	13	59	74
GENF5	92.45	5	<1	0	10.9	13.1	13	58	73
GENF6	89.70	11	<1	0	8.55	11.9	12	60	73
GENF7	96.41	10	<1	0	10.9	13.1	15	59	75
GENF8	99.40	46	<1	0	11.8	13.5	11	61	79
GENF9	95.80	13	<1	0	9.39	12.4	12	59	71
GENF10	99.89	203	<1	0	11.7	13.5	14	65	85
average	87.70	25	-	0	8.29	10.8	11	70	83

Table A.2: VFML10 method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	1030	30	28.0	51.7	150	19	10	54
RTSN	78.54	400	<10	14.9	1397	1948	17	11	76
RTC	83.58	370	<10	4.29	638	884	14	57	85
RTCN	64.95	190	<10	4.18	705	931	13	59	76
RRBFS	93.13	1000	20	24.4	570	1188	32	6	53
RRBFC	98.61	910	<10	17.3	216	467	34	21	78
WAVE21	84.20	900	<10	11.7	426	875	28	22	84
WAVE40	84.00	730	<10	6.89	324	662	27	37	88
GENF1	95.07	690	40	59.2	390	752	19	4	73
GENF2	94.10	1040	30	45.0	979	1382	21	5	68
GENF3	97.52	1230	190	61.0	314	689	17	7	74
GENF4	94.67	1070	20	44.8	936	1288	23	6	71
GENF5	92.89	980	<10	37.6	1281	1693	24	5	68
GENF6	93.35	1020	20	37.8	1151	1656	29	5	64
GENF7	96.82	1080	20	36.9	1244	1560	24	6	65
GENF8	99.42	1290	60	46.9	171	293	15	7	74
GENF9	96.81	1100	20	39.1	1298	1603	23	6	69
GENF10	99.89	1180	380	52.1	47.4	140	18	5	80
average	91.53	901	-	31.8	674	1009	22	16	72

Table A.3: VFML10 method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.98	610	?	68.6	0	128	16	6	70
RTSN	78.53	120	60	351	386	1017	18	4	81
RTC	83.87	90	<10	86.1	474	781	15	14	92
RTCN	66.06	80	20	71.7	474	738	14	24	99
RRBFS	92.43	180	180	418	16.0	867	45	1	40
RRBFC	97.41	70	50	110	38.8	297	45	2	70
WAVE21	83.50	110	100	185	19.0	409	41	3	79
WAVE40	83.31	80	60	99.5	40.0	279	35	5	87
GENF1	95.07	430	?	393	0	646	18	2	77
GENF2	94.10	300	?	553	0	732	20	2	71
GENF3	97.52	620	?	259	0	457	14	3	84
GENF4	94.66	260	?	542	0	681	20	1	72
GENF5	92.84	210	?	644	0	865	23	1	62
GENF6	93.28	230	?	570	0	849	31	1	58
GENF7	96.79	160	?	414	0	571	21	1	59
GENF8	99.42	470	?	216	0	293	19	2	74
GENF9	96.72	190	?	683	0	883	23	1	64
GENF10	99.89	1060	?	98.8	0	138	17	5	77
average	91.41	293	-	320	80.4	591	24	4	73

Table A.4: VFML100 method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	69.96	1	<1	0	0.41	0.51	3	82	89
RTSN	56.87	1	<1	0	0.05	0.06	1	94	95
RTC	54.24	1	<1	0	0.06	0.08	2	98	98
RTCN	53.32	1	<1	0	0.06	0.08	2	100	100
RRBFS	75.28	1	<1	0	0.18	0.35	6	90	98
RRBFC	54.91	1	<1	0	0.03	0.05	2	89	93
WAVE21	62.48	1	<1	0	0.03	0.05	2	97	98
WAVE40	58.97	1	<1	0	0.03	0.05	2	98	99
GENF1	95.07	26	<1	0	7.38	9.01	9	55	77
GENF2	93.92	12	<1	0	7.64	9.08	12	61	73
GENF3	97.47	29	<1	0	7.90	8.83	7	65	79
GENF4	94.40	5	<1	0	7.65	8.94	10	64	77
GENF5	82.81	7	<1	0	4.37	6.78	10	65	76
GENF6	89.98	8	<1	0	7.25	8.90	15	64	76
GENF7	96.27	9	<1	0	6.52	7.99	13	54	77
GENF8	99.38	42	<1	0	7.36	8.51	10	65	76
GENF9	95.34	16	<1	0	5.48	7.58	12	65	75
GENF10	99.88	583	<1	0	6.10	7.86	13	66	79
average	79.47	41	-	0	3.81	4.71	7	76	85

Table A.5: VFML100 method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	100.00	1170	<10	5.22	45.1	91.8	32	12	62
RTSN	78.54	660	<10	2.78	909	1366	22	19	63
RTC	78.51	380	<10	0.99	351	474	13	58	84
RTCN	61.38	240	<10	0.88	309	427	15	71	88
RRBFS	92.94	1160	<10	3.35	402	811	35	7	56
RRBFC	98.16	950	<10	0.88	130	262	45	22	76
WAVE21	83.89	940	<10	1.67	317	636	31	23	82
WAVE40	83.66	720	<10	0.84	222	446	31	36	84
GENF1	95.06	1130	<10	10.4	463	807	24	6	75
GENF2	94.11	1130	<10	9.95	754	1075	21	6	70
GENF3	97.51	1240	20	11.0	258	467	16	7	78
GENF4	94.69	1160	<10	9.16	753	994	21	6	71
GENF5	92.86	1090	<10	9.19	922	1280	24	6	65
GENF6	93.32	1070	<10	9.12	927	1272	28	6	65
GENF7	96.81	1170	<10	8.72	919	1095	24	6	69
GENF8	99.42	1300	20	7.63	177	224	16	7	82
GENF9	96.81	710	<10	8.40	748	881	22	4	65
GENF10	99.89	710	50	7.37	30.6	52.2	15	3	73
average	90.97	941	-	5.98	480	703	24	17	73

Table A.6: VFML100 method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	180	?	34.0	0	59.6	13	2	67
RTSN	78.52	100	<10	45.1	456	746	20	3	80
RTC	81.26	90	<10	13.5	390	539	12	14	92
RTCN	64.64	80	<10	11.0	373	522	14	24	100
RRBFS	92.23	110	20	51.2	106	315	40	1	49
RRBFC	97.72	100	<10	32.8	58.0	182	38	3	81
WAVE21	83.53	110	20	20.8	89.5	221	28	3	84
WAVE40	83.40	100	<10	11.8	74.8	173	26	5	92
GENF1	95.07	160	130	126	29.1	230	16	1	77
GENF2	94.10	140	70	133	108	316	25	1	68
GENF3	97.52	210	?	116	0	172	13	1	83
GENF4	94.67	130	70	119	158	345	17	1	72
GENF5	92.84	130	60	120	158	441	28	1	56
GENF6	93.21	120	40	120	212	456	30	1	60
GENF7	96.79	120	50	113	130	319	26	1	60
GENF8	99.42	160	?	93.2	0	113	16	1	79
GENF9	96.70	120	40	126	227	477	29	1	58
GENF10	99.89	400	?	43.9	0	60.6	18	2	77
average	91.19	142	-	73.9	143	316	23	4	74

Table A.7: VFML1000 method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	71.00	1	<1	0	0.42	0.53	3	77	91
RTSN	56.87	1	<1	0	0.05	0.06	1	88	93
RTC	54.65	1	<1	0	0.07	0.10	3	91	96
RTCN	53.32	1	<1	0	0.06	0.08	2	97	100
RRBFS	59.95	1	<1	0	0.04	0.07	3	82	100
RRBFC	55.23	1	<1	0	0.03	0.05	2	64	95
WAVE21	62.37	1	<1	0	0.03	0.05	2	92	98
WAVE40	64.48	1	<1	0	0.03	0.05	2	95	100
GENF1	94.81	1	<1	0	0.06	0.11	5	76	80
GENF2	93.09	1	<1	0	0.14	0.27	6	74	77
GENF3	97.35	1	<1	0	0.19	0.34	4	72	81
GENF4	84.13	1	<1	0	0.13	0.19	4	78	83
GENF5	71.27	1	<1	0	0.04	0.07	3	79	82
GENF6	76.13	1	<1	0	0.07	0.13	5	79	82
GENF7	88.63	1	<1	0	0.06	0.11	3	79	82
GENF8	98.68	1	<1	0	0.08	0.12	3	76	81
GENF9	87.35	1	<1	0	0.06	0.11	3	78	82
GENF10	99.77	1	<1	0	0.04	0.07	3	76	82
average	76.06	1	-	0	0.09	0.14	3	81	88

Table A.8: VFML1000 method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	100.00	1080	<10	3.40	53.9	104	44	11	62
RTSN	78.52	690	<10	2.25	799	1226	23	20	61
RTC	79.99	380	<10	0.50	270	369	15	58	85
RTCN	60.49	230	<10	0.98	259	365	17	70	95
RRBFS	92.78	1150	<10	3.22	359	724	54	7	53
RRBFC	98.01	960	<10	0.60	118	237	46	22	78
WAVE21	83.73	950	<10	1.30	287	577	36	23	81
WAVE40	83.51	730	<10	0.84	200	402	34	37	87
GENF1	95.06	730	<10	6.47	302	508	23	4	74
GENF2	94.10	1110	<10	7.38	662	953	22	6	66
GENF3	97.51	1210	<10	6.81	240	421	17	6	78
GENF4	94.67	1190	<10	7.88	694	934	20	6	69
GENF5	92.87	710	<10	6.77	674	866	23	4	67
GENF6	93.30	1110	<10	8.51	845	1176	29	6	66
GENF7	96.81	1180	<10	5.38	704	854	23	6	68
GENF8	99.42	1270	<10	3.71	140	172	15	7	77
GENF9	96.77	1210	<10	7.04	777	923	22	6	70
GENF10	99.89	1250	20	3.97	40.4	53.6	15	6	73
average	90.97	952	-	4.28	412	604	27	17	73

Table A.9: VFML1000 method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.98	100	<10	12.8	18.2	53.1	20	1	67
RTSN	78.50	90	<10	10.8	329	523	24	3	81
RTC	82.07	80	<10	34.3	195	309	13	13	93
RTCN	63.36	70	<10	7.29	240	344	15	22	99
RRBFS	92.02	100	<10	9.07	83.0	184	38	1	64
RRBFC	97.60	100	<10	12.2	57.3	139	38	3	82
WAVE21	83.29	110	<10	4.01	60.7	129	24	3	90
WAVE40	83.06	90	<10	1.69	48.2	99.8	23	5	95
GENF1	95.06	110	20	21.8	85.6	157	18	1	78
GENF2	94.10	110	20	25.4	143	213	18	1	72
GENF3	97.52	120	40	15.4	49.3	78.2	13	1	83
GENF4	94.66	110	20	27.6	170	244	16	1	74
GENF5	92.84	110	<10	23.8	218	309	22	1	71
GENF6	93.23	110	<10	23.8	219	319	28	1	65
GENF7	96.79	110	20	23.0	161	233	27	1	64
GENF8	99.41	130	70	41.3	35.0	93.0	17	1	79
GENF9	96.70	100	<10	22.3	168	258	22	1	71
GENF10	99.89	200	?	25.6	0	31.0	16	1	76
average	91.12	108	-	19.0	127	206	22	3	78

Table A.10: BINTREE method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	66.81	1	<1	0	0.33	0.41	3	70	94
RTSN	56.87	1	<1	0	0.05	0.06	1	82	94
RTC	54.23	1	<1	0	0.06	0.08	2	85	97
RTCN	53.31	1	<1	0	0.06	0.08	2	94	100
RRBFS	59.72	1	<1	0	0.04	0.07	3	68	100
RRBFC	55.27	1	<1	0	0.03	0.05	2	43	95
WAVE21	62.84	1	<1	0	0.03	0.05	2	82	99
WAVE40	57.60	1	<1	0	0.02	0.03	1	89	100
GENF1	94.79	1	<1	0	0.06	0.11	4	65	81
GENF2	77.23	1	<1	0	0.08	0.15	5	75	81
GENF3	97.08	1	<1	0	0.10	0.16	2	76	85
GENF4	82.24	1	<1	0	0.15	0.20	3	77	84
GENF5	71.23	1	<1	0	0.04	0.07	3	75	83
GENF6	77.06	1	<1	0	0.04	0.07	3	79	85
GENF7	88.63	1	<1	0	0.06	0.11	3	73	81
GENF8	98.00	1	<1	0	0.04	0.07	2	71	80
GENF9	87.40	1	<1	0	0.06	0.11	3	76	82
GENF10	99.77	1	<1	0	0.04	0.07	3	80	86
average	74.45	1	-	0	0.07	0.11	3	76	89

Table A.11: BINTREE method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	850	<10	2.44	52.2	97.2	21	9	52
RTSN	78.49	610	<10	2.26	709	1065	20	18	66
RTC	73.08	350	<10	0.69	257	347	14	53	83
RTCN	59.14	220	<10	0.35	238	328	15	67	87
RRBFS	92.63	960	<10	2.24	301	607	33	6	59
RRBFC	97.88	790	<10	0.49	98.5	198	33	18	81
WAVE21	83.70	820	<10	1.26	245	492	31	20	83
WAVE40	83.41	650	<10	0.51	172	344	33	32	89
GENF1	95.06	960	<10	8.21	371	628	23	5	74
GENF2	94.09	610	<10	6.69	407	576	20	3	68
GENF3	97.51	1070	<10	4.89	212	373	17	6	78
GENF4	94.66	1010	<10	6.60	620	817	19	5	68
GENF5	92.86	960	<10	6.40	846	1093	23	5	68
GENF6	93.27	950	<10	7.38	742	1014	25	5	65
GENF7	96.80	1030	<10	5.03	590	721	22	5	67
GENF8	99.43	1040	<10	3.89	122	149	15	6	76
GENF9	96.74	1050	<10	5.69	685	825	21	6	70
GENF10	99.89	1100	<10	0.98	47.9	59.2	17	5	79
average	90.48	835	-	3.67	373	541	22	15	73

Table A.12: BINTREE method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.97	70	<10	11.7	18.2	50.9	15	1	69
RTSN	78.49	70	<10	9.36	263	411	19	2	98
RTC	76.15	60	<10	3.30	172	234	11	10	93
RTCN	61.84	50	<10	2.55	199	279	15	17	99
RRBFS	91.29	40	<10	7.88	36.9	89.5	31	0	68
RRBFC	96.83	40	<10	7.32	31.0	76.6	26	1	86
WAVE21	82.28	20	<10	1.41	14.0	30.8	19	1	94
WAVE40	82.18	20	<10	1.02	14.0	30.1	20	2	95
GENF1	95.05	80	20	61.6	23.8	121	16	0	80
GENF2	94.06	80	<10	14.2	129	188	19	0	71
GENF3	97.52	80	20	11.3	44.3	68.4	12	0	82
GENF4	94.66	80	<10	20.1	125	177	18	0	74
GENF5	92.81	80	<10	37.2	177	262	20	0	70
GENF6	93.23	80	<10	12.5	218	286	23	0	67
GENF7	96.75	50	<10	13.8	89.3	134	20	0	65
GENF8	99.40	10	<10	4.31	2.89	11.3	14	0	78
GENF9	96.65	70	<10	15.4	110	176	19	0	71
GENF10	99.89	100	30	11.3	4.99	21.6	16	0	80
average	90.50	60	-	13.7	92.9	147	19	2	80

Table A.13: GK100 method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	90.34	3	<1	0	3.09	4.25	6	74	88
RTSN	72.81	2	<1	0	2.66	3.63	7	83	90
RTC	53.78	1	<1	0	0.08	0.12	4	94	97
RTCN	52.96	1	<1	0	0.08	0.12	4	100	100
RRBFS	85.74	4	<1	0	2.01	4.01	14	59	88
RRBFC	53.11	1	<1	0	0.05	0.09	4	72	90
WAVE21	63.72	1	<1	0	0.04	0.07	2	90	98
WAVE40	72.21	1	<1	0	0.09	0.17	4	93	99
GENF1	95.07	19	<1	0	7.17	8.26	9	56	78
GENF2	85.94	10	<1	0	6.05	7.72	12	62	75
GENF3	97.47	43	<1	0	7.94	9.07	7	64	79
GENF4	94.29	5	<1	0	7.15	8.41	9	63	78
GENF5	92.37	4	<1	0	7.93	9.12	11	59	76
GENF6	92.05	5	<1	0	7.15	8.71	15	60	76
GENF7	96.23	10	<1	0	6.62	8.19	13	54	77
GENF8	99.35	39	<1	0	7.02	8.24	10	65	78
GENF9	95.21	16	<1	0	5.48	7.83	12	63	74
GENF10	99.88	361	<1	0	6.95	8.24	17	68	79
average	82.92	29	-	0	4.31	5.35	9	71	84

Table A.14: GK100 method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	100.00	1170	<10	9.91	42.6	92.4	44	12	64
RTSN	78.54	580	<10	5.35	1035	1613	39	17	62
RTC	67.14	390	<10	1.18	500	695	39	59	85
RTCN	53.96	230	<10	1.52	394	572	46	70	92
RRBFS	93.13	1160	<10	5.96	476	965	40	7	56
RRBFC	98.23	970	<10	1.94	168	340	45	23	75
WAVE21	83.98	920	<10	2.98	346	698	41	22	81
WAVE40	83.75	690	<10	1.66	253	508	38	34	87
GENF1	95.06	1190	<10	9.80	537	920	48	6	71
GENF2	94.09	1150	<10	9.81	777	1126	32	6	68
GENF3	97.51	720	30	11.6	203	348	34	4	79
GENF4	94.66	1160	<10	10.2	771	1049	24	6	68
GENF5	92.86	1140	<10	9.37	1016	1367	31	6	66
GENF6	93.36	700	<10	8.94	775	983	26	4	66
GENF7	96.82	1200	<10	8.38	962	1154	23	6	68
GENF8	99.42	1370	30	8.22	182	233	19	7	79
GENF9	96.80	1210	<10	8.63	1063	1252	24	7	68
GENF10	99.89	1360	110	8.34	49.2	71.3	18	6	80
average	89.96	962	-	6.88	531	777	34	17	73

Table A.15: GK100 method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	280	?	36.8	0	62.7	13	3	70
RTSN	78.50	90	20	63.1	365	738	50	3	74
RTC	68.51	90	<10	14.6	415	602	49	14	91
RTCN	53.93	70	<10	14.6	281	486	47	22	99
RRBFS	92.31	120	40	72.9	125	397	55	1	52
RRBFC	97.59	100	<10	47.9	63.4	223	43	2	79
WAVE21	83.41	100	20	31.3	90.7	244	43	3	87
WAVE40	83.36	90	20	17.6	78.8	193	34	5	94
GENF1	95.05	180	160	149	44.0	312	33	1	72
GENF2	94.08	130	80	133	101	328	24	1	68
GENF3	97.50	230	?	132	0	204	22	1	82
GENF4	94.64	130	80	123	138	348	23	1	68
GENF5	92.83	120	50	118	198	440	26	1	60
GENF6	93.31	130	50	120	290	517	27	1	64
GENF7	96.79	140	60	121	170	394	27	1	61
GENF8	99.42	240	?	126	0	152	22	1	82
GENF9	96.68	130	60	140	242	530	23	1	59
GENF10	99.89	480	?	51.4	0	64.4	20	2	80
average	89.88	158	-	84.0	145	346	32	4	75

Table A.16: GK1000 method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	69.96	1	<1	0	0.41	0.51	3	58	88
RTSN	56.87	1	<1	0	0.05	0.06	1	76	94
RTC	54.64	1	<1	0	0.07	0.10	3	75	98
RTCN	53.32	1	<1	0	0.06	0.08	2	89	100
RRBFS	59.91	1	<1	0	0.04	0.07	3	40	100
RRBFC	55.21	1	<1	0	0.03	0.05	2	26	95
WAVE21	61.83	1	<1	0	0.04	0.07	2	58	97
WAVE40	57.60	1	<1	0	0.02	0.03	1	66	100
GENF1	94.81	1	<1	0	0.06	0.11	5	59	80
GENF2	77.78	1	<1	0	0.07	0.13	5	55	81
GENF3	97.18	1	<1	0	0.11	0.18	3	69	84
GENF4	82.45	1	<1	0	0.12	0.17	3	64	83
GENF5	70.95	1	<1	0	0.04	0.07	3	55	82
GENF6	76.40	1	<1	0	0.07	0.13	5	48	82
GENF7	88.98	1	<1	0	0.06	0.11	3	67	82
GENF8	98.72	1	<1	0	0.08	0.12	3	55	81
GENF9	87.30	1	<1	0	0.06	0.11	3	64	82
GENF10	99.85	1	<1	0	0.10	0.16	4	45	83
average	74.65	1	-	0	0.08	0.13	3	59	88

Table A.17: GK1000 method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	100.00	1120	<10	3.05	44.6	83.7	49	11	70
RTSN	78.53	680	<10	2.17	758	1165	24	20	69
RTC	80.44	360	<10	0.58	267	363	15	55	92
RTCN	59.41	240	<10	0.55	261	370	18	71	96
RRBFS	92.80	1120	<10	2.41	342	688	49	6	56
RRBFC	97.94	930	<10	0.55	113	227	50	22	77
WAVE21	83.78	840	<10	1.22	263	528	34	20	84
WAVE40	83.48	630	<10	0.64	180	360	35	32	86
GENF1	95.07	1190	<10	4.04	407	696	25	6	76
GENF2	94.11	1120	<10	3.98	581	833	25	6	71
GENF3	97.51	1190	<10	4.38	239	411	17	6	77
GENF4	94.68	1190	<10	3.98	664	867	21	6	68
GENF5	92.88	830	<10	4.06	691	877	23	4	71
GENF6	93.36	1100	<10	3.77	780	1017	29	6	65
GENF7	96.81	1170	<10	3.82	698	833	22	6	69
GENF8	99.42	1240	<10	3.11	140	170	15	7	78
GENF9	96.77	1200	<10	3.88	786	928	23	6	72
GENF10	99.89	1140	20	2.32	38.4	48.7	15	5	79
average	90.94	961	-	2.70	403	581	27	16	75

Table A.18: GK1000 method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.97	60	<10	17.6	8.00	42.2	10	1	69
RTSN	78.50	90	<10	18.7	351	574	20	3	81
RTC	81.85	70	<10	7.10	192	267	12	11	94
RTCN	62.60	60	<10	5.17	223	316	20	19	100
RRBFS	92.08	100	<10	17.5	89.1	213	42	1	64
RRBFC	97.57	90	<10	16.5	55.1	143	46	2	82
WAVE21	82.94	50	<10	4.73	35.3	80.1	22	1	95
WAVE40	82.78	40	<10	3.16	27.0	60.3	20	2	95
GENF1	95.06	100	20	25.0	79.4	152	17	1	77
GENF2	94.10	100	20	18.3	138	204	19	1	71
GENF3	97.52	110	50	17.6	45.8	76.6	13	1	85
GENF4	94.67	100	<10	23.6	133	193	16	1	76
GENF5	92.86	100	<10	22.9	209	290	19	1	72
GENF6	93.34	100	<10	18.8	249	326	27	1	69
GENF7	96.80	110	20	18.4	156	222	28	1	67
GENF8	99.41	110	70	37.9	28.0	80.0	16	1	80
GENF9	96.67	100	<10	23.0	178	275	23	1	65
GENF10	99.89	150	?	19.8	0	24.3	16	1	80
average	91.03	91	-	17.6	122	197	21	3	79

Table A.19: GAUSS10 method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	96.95	11	<1	0	9.05	12.6	8	74	82
RTSN	75.20	6	<1	0	9.35	12.7	11	81	85
RTC	62.49	10	<1	0	8.32	10.6	6	95	96
RTCN	53.63	10	<1	0	8.57	10.7	5	100	100
RRBFS	88.56	8	<1	0	5.44	10.9	19	64	79
RRBFC	91.36	12	<1	0	4.92	9.83	29	74	83
WAVE21	81.21	12	<1	0	4.92	9.83	16	81	87
WAVE40	81.20	13	<1	0	4.62	9.23	16	91	95
GENF1	95.07	11	<1	0	11.4	13.8	11	47	76
GENF2	78.46	4	<1	0	9.90	13.0	10	56	72
GENF3	97.50	35	<1	0	12.2	14.1	7	59	79
GENF4	93.68	6	<1	0	11.3	13.7	12	57	74
GENF5	71.73	4	<1	0	8.75	12.5	11	56	72
GENF6	91.89	5	<1	0	11.0	13.6	11	57	75
GENF7	96.51	9	<1	0	10.5	13.2	13	59	75
GENF8	99.41	36	<1	0	11.6	13.9	10	61	76
GENF9	96.07	12	<1	0	8.69	12.4	12	57	70
GENF10	99.88	281	<1	0	10.6	13.4	13	64	82
average	86.16	27	-	0	8.96	12.2	12	69	81

Table A.20: GAUSS10 method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	1110	1000	89.0	3.78	176	16	11	62
RTSN	78.48	350	20	34.6	1890	2509	23	10	67
RTC	83.00	300	<10	11.6	729	1001	13	45	67
RTCN	62.45	220	<10	12.5	849	1103	11	67	92
RRBFS	93.27	730	60	97.5	576	1346	53	4	36
RRBFC	98.72	770	<10	67.0	227	588	35	18	76
WAVE21	84.37	730	40	44.4	505	1098	37	18	78
WAVE40	84.21	620	20	26.9	391	835	32	31	89
GENF1	95.07	1000	190	144	504	1175	20	5	73
GENF2	94.03	900	60	120	988	1643	23	5	60
GENF3	97.52	1170	470	168	253	787	17	6	79
GENF4	94.67	920	80	139	824	1372	25	5	65
GENF5	92.36	720	50	99.6	966	1804	36	4	49
GENF6	93.31	840	40	108	1152	1727	21	4	63
GENF7	96.81	1010	60	109	1151	1593	21	5	62
GENF8	99.42	1350	690	161	169	467	17	7	79
GENF9	96.78	990	50	120	1120	1560	19	5	70
GENF10	99.89	2320	?	156	0	229	20	11	83
average	91.35	892	-	94.8	683	1167	24	14	69

Table A.21: GAUSS10 method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	1170	?	94.2	0	178	16	12	71
RTSN	78.45	170	?	1114	0	1450	23	5	82
RTC	83.02	80	20	218	442	928	14	13	92
RTCN	61.87	50	40	190	40.6	309	12	17	99
RRBFS	92.93	350	?	752	0	1503	57	2	37
RRBFC	98.21	200	?	300	0	601	50	5	67
WAVE21	84.01	250	?	485	0	969	57	6	76
WAVE40	83.80	180	?	346	0	691	59	9	88
GENF1	95.07	480	?	405	0	696	19	3	77
GENF2	94.00	430	?	817	0	1269	23	2	61
GENF3	97.52	1020	?	388	0	722	17	5	83
GENF4	94.65	460	?	743	0	1028	27	3	70
GENF5	92.15	350	?	984	0	1767	39	2	48
GENF6	93.28	370	?	932	0	1309	24	2	61
GENF7	96.79	290	?	654	0	868	19	2	65
GENF8	99.42	810	?	199	0	280	17	4	82
GENF9	96.74	360	?	952	0	1250	19	2	66
GENF10	99.89	2310	?	155	0	228	20	11	82
average	91.21	518	-	540	26.8	891	28	6	73

Table A.22: GAUSS100 method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	98.00	15	<1	0	8.97	12.5	7	74	79
RTSN	70.84	4	<1	0	8.01	12.0	31	73	83
RTC	63.38	10	<1	0	8.04	10.5	6	95	96
RTCN	52.99	5	<1	0	7.40	10.1	15	100	100
RRBFS	87.86	8	<1	0	5.44	10.9	20	54	75
RRBFC	84.95	8	<1	0	4.92	9.83	118	57	74
WAVE21	81.30	12	<1	0	4.91	9.81	16	75	92
WAVE40	81.05	12	<1	0	4.67	9.33	22	86	95
GENF1	95.00	11	<1	0	10.1	13.1	13	45	74
GENF2	71.53	4	<1	0	7.30	11.8	14	47	68
GENF3	97.52	29	<1	0	13.1	14.5	8	59	79
GENF4	88.75	4	<1	0	7.84	12.1	11	50	72
GENF5	82.80	5	<1	0	9.44	12.8	14	51	72
GENF6	88.15	7	<1	0	8.11	12.2	15	53	72
GENF7	96.53	10	<1	0	10.5	13.3	14	56	74
GENF8	99.41	36	<1	0	11.3	13.7	10	58	75
GENF9	96.00	11	<1	0	8.83	12.4	12	55	70
GENF10	99.88	311	<1	0	10.9	13.5	14	61	79
average	85.33	28	-	0	8.33	11.9	20	64	79

Table A.23: GAUSS100 method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	1310	?	73.8	0	136	15	13	61
RTSN	78.06	260	20	43.6	1150	2029	109	8	52
RTC	83.27	340	<10	10.8	805	1115	16	52	83
RTCN	54.68	190	<10	9.92	840	1329	94	57	89
RRBFS	93.32	780	60	95.2	620	1431	62	4	48
RRBFC	98.49	700	<10	64.2	268	664	197	16	66
WAVE21	84.34	670	40	45.9	491	1074	47	16	79
WAVE40	84.18	560	20	28.2	379	814	37	28	86
GENF1	95.06	930	190	142	516	1200	28	5	69
GENF2	94.04	810	60	119	867	1577	47	4	49
GENF3	97.52	1170	430	165	285	846	19	6	79
GENF4	94.65	840	80	129	769	1408	51	5	50
GENF5	92.64	770	40	98.4	1126	1843	51	4	54
GENF6	93.17	790	50	104	1005	1716	42	4	52
GENF7	96.82	940	60	107	1131	1595	22	5	58
GENF8	99.42	1300	610	161	168	476	21	7	72
GENF9	96.78	990	50	119	1089	1533	21	5	62
GENF10	99.89	2000	?	152	0	226	24	9	68
average	90.91	853	-	92.6	639	1167	50	14	65

Table A.24: GAUSS100 method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	1350	?	74.4	0	137	15	14	70
RTSN	78.02	170	?	1018	0	1670	107	5	58
RTC	82.94	80	20	217	433	916	20	12	93
RTCN	54.30	70	30	185	263	693	193	22	96
RRBFS	92.95	350	?	753	0	1506	70	2	35
RRBFC	97.75	180	?	328	0	656	205	4	59
WAVE21	83.99	240	?	469	0	939	80	6	79
WAVE40	83.79	170	?	333	0	666	66	9	88
GENF1	95.07	640	?	571	0	1031	29	3	71
GENF2	93.99	410	?	829	0	1418	56	2	49
GENF3	97.52	960	?	392	0	731	19	5	83
GENF4	94.62	430	?	810	0	1317	61	2	48
GENF5	92.61	350	?	987	0	1576	57	2	51
GENF6	93.10	370	?	910	0	1515	47	2	48
GENF7	96.79	400	?	892	0	1235	28	2	54
GENF8	99.42	1110	?	302	0	441	22	6	74
GENF9	96.75	350	?	953	0	1290	31	2	55
GENF10	99.89	2050	?	156	0	231	24	9	70
average	90.75	538	-	566	38.7	998	63	6	66

A.2 Prediction Methods

Table A.25: HTMC method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	96.95	11	<1	0	9.05	12.6	8	74	82
RTSN	75.20	6	<1	0	9.35	12.7	11	81	85
RTC	62.49	10	<1	0	8.32	10.6	6	95	96
RTCN	53.63	10	<1	0	8.57	10.7	5	100	100
RRBFS	88.56	8	<1	0	5.44	10.9	19	64	79
RRBFC	91.36	12	<1	0	4.92	9.83	29	74	83
LED	73.94	28	<1	0	2.98	5.95	11	76	81
WAVE21	81.21	12	<1	0	4.92	9.83	16	81	87
WAVE40	81.20	13	<1	0	4.62	9.23	16	91	95
GENF1	95.07	11	<1	0	11.4	13.8	11	47	76
GENF2	78.46	4	<1	0	9.90	13.0	10	56	72
GENF3	97.50	35	<1	0	12.2	14.1	7	59	79
GENF4	93.68	6	<1	0	11.3	13.7	12	57	74
GENF5	71.73	4	<1	0	8.75	12.5	11	56	72
GENF6	91.89	5	<1	0	11.0	13.6	11	57	75
GENF7	96.51	9	<1	0	10.5	13.2	13	59	75
GENF8	99.41	36	<1	0	11.6	13.9	10	61	76
GENF9	96.07	12	<1	0	8.69	12.4	12	57	70
GENF10	99.88	281	<1	0	10.6	13.4	13	64	82
average	85.51	27	-	0	8.64	11.9	12	69	81

Table A.26: HTMC method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	1110	1000	89.0	3.78	176	16	11	62
RTSN	78.48	350	20	34.6	1890	2509	23	10	67
RTC	83.00	300	<10	11.6	729	1001	13	45	67
RTCN	62.45	220	<10	12.5	849	1103	11	67	92
RRBFS	93.27	730	60	97.5	576	1346	53	4	36
RRBFC	98.72	770	<10	67.0	227	588	35	18	76
LED	73.99	1080	200	45.8	219	529	18	12	52
WAVE21	84.37	730	40	44.4	505	1098	37	18	78
WAVE40	84.21	620	20	26.9	391	835	32	31	89
GENF1	95.07	1000	190	144	504	1175	20	5	73
GENF2	94.03	900	60	120	988	1643	23	5	60
GENF3	97.52	1170	470	168	253	787	17	6	79
GENF4	94.67	920	80	139	824	1372	25	5	65
GENF5	92.36	720	50	99.6	966	1804	36	4	49
GENF6	93.31	840	40	108	1152	1727	21	4	63
GENF7	96.81	1010	60	109	1151	1593	21	5	62
GENF8	99.42	1350	690	161	169	467	17	7	79
GENF9	96.78	990	50	120	1120	1560	19	5	70
GENF10	99.89	2320	?	156	0	229	20	11	83
average	90.44	902	-	92.3	659	1134	24	14	69

Table A.27: HTMC method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	1170	?	94.2	0	178	16	12	71
RTSN	78.45	170	?	1114	0	1450	23	5	82
RTC	83.02	80	20	218	442	928	14	13	92
RTCN	61.87	50	40	190	40.6	309	12	17	99
RRBFS	92.93	350	?	752	0	1503	57	2	37
RRBFC	98.21	200	?	300	0	601	50	5	67
LED	73.96	650	?	188	0	375	17	7	56
WAVE21	84.01	250	?	485	0	969	57	6	76
WAVE40	83.80	180	?	346	0	691	59	9	88
GENF1	95.07	480	?	405	0	696	19	3	77
GENF2	94.00	430	?	817	0	1269	23	2	61
GENF3	97.52	1020	?	388	0	722	17	5	83
GENF4	94.65	460	?	743	0	1028	27	3	70
GENF5	92.15	350	?	984	0	1767	39	2	48
GENF6	93.28	370	?	932	0	1309	24	2	61
GENF7	96.79	290	?	654	0	868	19	2	65
GENF8	99.42	810	?	199	0	280	17	4	82
GENF9	96.74	360	?	952	0	1250	19	2	61
GENF10	99.89	2310	?	155	0	228	20	11	82
average	90.30	525	-	522	25.4	864	28	6	71

Table A.28: HTNB method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	96.87	11	<1	0	9.15	12.6	8	73	80
RTSN	75.21	6	<1	0	9.34	12.6	12	81	85
RTC	61.22	10	<1	0	8.41	10.7	6	97	98
RTCN	53.63	10	<1	0	8.57	10.7	5	100	100
RRBFS	88.51	8	<1	0	5.44	10.9	18	63	78
RRBFC	91.24	14	<1	0	4.92	9.83	30	80	84
LED	73.94	30	<1	0	2.96	5.91	11	76	81
WAVE21	81.28	13	<1	0	4.93	9.85	16	83	87
WAVE40	81.20	14	<1	0	4.67	9.33	18	91	96
GENF1	95.07	11	<1	0	11.3	13.7	11	47	75
GENF2	78.84	4	<1	0	9.77	12.9	11	56	72
GENF3	97.49	33	<1	0	12.3	14.2	7	61	79
GENF4	93.83	6	<1	0	11.6	13.8	12	47	74
GENF5	71.84	4	<1	0	8.63	12.4	10	57	72
GENF6	92.08	5	<1	0	10.8	13.4	12	58	76
GENF7	96.52	9	<1	0	10.7	13.4	13	47	75
GENF8	99.41	40	<1	0	11.6	13.8	11	60	78
GENF9	95.97	12	<1	0	9.16	12.6	12	58	73
GENF10	99.89	266	<1	0	10.9	13.5	14	65	82
average	85.48	27	-	0	8.69	11.9	12	68	81

Table A.29: HTNB method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	670	?	81.0	0	152	16	7	30
RTSN	78.41	370	20	31.6	1928	2557	23	11	68
RTC	83.16	350	<10	10.8	822	1119	14	53	83
RTCN	62.32	220	<10	11.0	847	1096	11	68	92
RRBFS	93.60	780	60	87.0	600	1374	53	5	43
RRBFC	98.85	790	<10	24.8	202	454	33	19	60
LED	74.02	1090	100	21.0	195	431	18	12	44
WAVE21	84.82	760	40	41.0	512	1106	37	18	76
WAVE40	84.55	410	20	26.2	281	614	32	21	78
GENF1	94.99	630	190	121	323	775	19	3	65
GENF2	94.01	910	50	92.7	989	1597	23	5	57
GENF3	97.48	1240	280	121	286	760	17	7	72
GENF4	94.65	1010	60	99.6	935	1450	25	5	65
GENF5	92.27	520	40	94.1	733	1425	33	3	48
GENF6	93.26	870	40	85.4	1351	1868	20	5	58
GENF7	96.77	1020	60	89.1	1333	1758	21	5	65
GENF8	99.36	1320	510	133	179	431	18	7	74
GENF9	96.77	620	50	112	696	1025	20	3	63
GENF10	99.84	2100	?	145	0	210	20	10	47
average	90.48	825	-	75.1	643	1063	24	14	63

Table A.30: HTNB method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	1120	?	92.9	0	176	16	11	41
RTSN	78.07	150	140	912	66.8	1274	23	5	50
RTC	83.78	80	20	178	462	897	14	13	78
RTCN	62.50	80	40	180	404	778	13	25	88
RRBFS	93.52	250	?	552	0	1104	56	1	21
RRBFC	98.44	180	?	279	0	558	48	4	27
LED	73.99	330	?	93.8	0	188	16	4	15
WAVE21	85.21	250	?	484	0	968	57	6	38
WAVE40	84.89	180	?	345	0	691	59	10	44
GENF1	94.80	430	?	365	0	618	19	2	42
GENF2	93.72	400	?	768	0	1196	22	2	33
GENF3	97.36	910	?	350	0	648	16	5	43
GENF4	94.27	430	?	720	0	975	27	2	36
GENF5	91.67	230	?	703	0	1272	39	1	27
GENF6	92.18	220	?	710	0	937	23	1	31
GENF7	95.49	390	?	889	0	1173	19	2	33
GENF8	99.26	1080	?	280	0	388	18	6	45
GENF9	95.64	350	?	933	0	1225	19	2	31
GENF10	99.84	1750	?	123	0	177	19	11	42
average	90.24	464	-	471	49.1	802	28	6	40

Table A.31: HTNB1K method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	96.87	11	<1	0	9.15	12.6	8	74	80
RTSN	75.21	6	<1	0	9.34	12.6	12	81	86
RTC	61.22	10	<1	0	8.41	10.7	6	96	96
RTCN	53.63	10	<1	0	8.57	10.7	5	100	100
RRBFS	88.51	8	<1	0	5.44	10.9	18	52	80
RRBFC	91.24	14	<1	0	4.92	9.83	30	74	83
LED	73.94	30	<1	0	2.96	5.91	11	76	82
WAVE21	81.28	13	<1	0	4.93	9.85	16	81	87
WAVE40	81.20	14	<1	0	4.67	9.33	18	91	95
GENF1	95.07	11	<1	0	11.3	13.7	11	47	76
GENF2	78.84	4	<1	0	9.77	12.9	11	57	72
GENF3	97.49	33	<1	0	12.3	14.2	7	62	81
GENF4	93.83	6	<1	0	11.6	13.8	12	56	72
GENF5	71.84	4	<1	0	8.63	12.4	10	57	72
GENF6	92.08	5	<1	0	10.8	13.4	12	58	75
GENF7	96.52	9	<1	0	10.7	13.4	13	59	75
GENF8	99.41	40	<1	0	11.6	13.8	11	61	77
GENF9	95.97	12	<1	0	9.16	12.6	12	58	72
GENF10	99.89	266	<1	0	10.9	13.5	14	45	77
average	85.48	27	-	0	8.69	11.9	12	68	81

Table A.32: HTNB1K method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	1010	830	84.6	5.42	170	16	10	32
RTSN	78.39	360	20	32.6	1884	2500	23	11	65
RTC	83.16	350	<10	10.8	822	1119	14	53	83
RTCN	62.24	190	<10	11.1	817	1058	11	57	75
RRBFS	93.61	810	60	85.9	616	1404	53	5	43
RRBFC	98.84	810	<10	24.7	205	460	33	19	75
LED	74.01	1100	100	20.5	197	436	18	12	47
WAVE21	84.80	770	40	40.9	517	1115	37	19	77
WAVE40	84.49	630	20	24.4	387	822	32	31	85
GENF1	95.02	1030	190	106	532	1156	20	5	68
GENF2	94.01	910	50	92.7	989	1597	23	5	55
GENF3	97.47	1220	280	122	281	751	17	6	72
GENF4	94.65	980	60	101	913	1416	25	5	63
GENF5	92.37	770	40	79.6	1007	1819	34	4	49
GENF6	93.26	880	40	85.0	1362	1882	20	5	62
GENF7	96.77	1000	60	89.7	1316	1738	21	5	61
GENF8	99.37	1330	510	133	180	433	18	7	74
GENF9	96.78	1030	50	96.6	1104	1506	20	6	61
GENF10	99.85	2010	?	139	0	200	20	9	43
average	90.48	905	-	72.5	691	1136	24	14	63

Table A.33: HTNB1K method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	1120	?	92.9	0	176	16	11	41
RTSN	78.36	150	140	912	66.8	1274	23	5	56
RTC	83.53	80	20	178	462	897	14	13	82
RTCN	62.49	80	40	180	404	778	13	25	89
RRBFS	93.53	350	?	750	0	1501	57	2	21
RRBFC	98.15	180	?	279	0	558	48	4	31
LED	73.99	330	?	93.8	0	188	16	4	15
WAVE21	85.20	250	?	484	0	968	57	6	39
WAVE40	84.92	180	?	345	0	691	59	9	45
GENF1	94.80	640	?	520	0	923	20	3	42
GENF2	93.81	400	?	768	0	1196	22	2	35
GENF3	97.37	630	?	251	0	451	16	3	44
GENF4	94.38	430	?	720	0	975	27	2	38
GENF5	92.00	340	?	960	0	1728	39	2	29
GENF6	92.74	330	?	981	0	1309	24	2	34
GENF7	95.97	280	?	631	0	839	19	2	34
GENF8	99.30	740	?	196	0	265	18	4	46
GENF9	96.10	340	?	916	0	1202	19	2	33
GENF10	99.86	1700	?	118	0	171	19	12	42
average	90.34	450	-	494	49.1	847	28	6	42

Table A.34: HTNBA method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	96.92	10	<1	0	9.15	12.6	8	73	84
RTSN	74.91	6	<1	0	9.45	12.7	11	80	86
RTC	61.22	10	<1	0	8.41	10.7	6	95	95
RTCN	53.60	10	<1	0	8.57	10.7	5	100	100
RRBFS	88.43	8	<1	0	5.44	10.9	17	58	79
RRBFC	91.19	13	<1	0	4.92	9.83	31	77	84
LED	73.96	30	<1	0	2.96	5.91	11	71	81
WAVE21	81.23	13	<1	0	4.93	9.85	16	77	88
WAVE40	81.20	14	<1	0	4.67	9.33	18	87	96
GENF1	95.07	11	<1	0	11.2	13.7	11	46	75
GENF2	78.30	4	<1	0	9.76	13.0	12	51	72
GENF3	97.49	37	<1	0	12.2	14.1	7	61	79
GENF4	93.86	6	<1	0	11.7	13.8	12	54	74
GENF5	72.10	4	<1	0	8.41	12.3	10	52	73
GENF6	92.09	5	<1	0	10.7	13.3	11	55	77
GENF7	96.53	10	<1	0	10.7	13.3	13	46	74
GENF8	99.41	40	<1	0	11.6	13.8	11	59	76
GENF9	95.98	13	<1	0	8.97	12.5	12	57	73
GENF10	99.89	303	<1	0	10.7	13.4	14	65	84
average	85.44	29	-	0	8.65	11.9	12	67	82

Table A.35: HTNBA method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	940	820	84.2	3.84	166	16	10	37
RTSN	78.49	400	20	29.0	2043	2704	23	12	78
RTC	83.10	340	<10	11.0	798	1087	14	52	83
RTCN	62.26	180	<10	11.2	805	1042	11	56	74
RRBFS	93.60	770	60	87.0	594	1361	53	4	43
RRBFC	98.85	830	<10	24.6	208	466	33	19	78
LED	74.02	980	100	24.0	142	333	17	11	49
WAVE21	84.80	720	40	41.5	494	1071	37	17	77
WAVE40	84.52	600	20	24.6	373	794	32	30	85
GENF1	95.06	1030	190	105	532	1154	20	5	72
GENF2	94.05	920	50	91.9	994	1603	23	5	58
GENF3	97.52	1210	270	121	279	746	17	6	74
GENF4	94.68	970	60	101	905	1404	25	5	65
GENF5	92.41	770	40	79.1	1009	1820	34	4	49
GENF6	93.31	870	40	84.8	1352	1867	20	5	64
GENF7	96.82	1010	60	88.8	1325	1747	21	5	63
GENF8	99.42	1300	500	132	175	424	18	7	78
GENF9	96.81	620	50	111	697	1025	20	3	66
GENF10	99.89	2080	?	144	0	208	20	10	44
average	90.51	871	-	73.4	670	1106	24	14	65

Table A.36: HTNBA method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	1040	?	90.9	0	172	16	11	51
RTSN	78.44	150	140	908	70.2	1274	23	5	81
RTC	83.84	80	20	178	463	897	14	13	85
RTCN	63.19	70	40	180	355	712	13	24	96
RRBFS	93.84	340	?	731	0	1462	57	2	27
RRBFC	98.95	170	?	268	0	536	48	4	41
LED	73.98	400	?	111	0	221	17	4	22
WAVE21	85.66	180	?	347	0	694	54	4	58
WAVE40	85.52	170	?	326	0	652	58	9	66
GENF1	95.05	620	?	508	0	899	20	3	54
GENF2	94.05	400	?	768	0	1196	22	2	48
GENF3	97.51	890	?	343	0	634	16	5	59
GENF4	94.65	420	?	708	0	958	27	2	54
GENF5	92.40	230	?	703	0	1272	39	1	38
GENF6	93.29	320	?	959	0	1279	24	2	47
GENF7	96.80	390	?	889	0	1173	19	2	49
GENF8	99.42	1050	?	271	0	376	18	6	48
GENF9	96.78	330	?	899	0	1179	19	2	46
GENF10	99.89	1540	?	108	0	157	19	11	42
average	90.70	463	-	489	46.7	828	28	6	53

A.3 Ensemble Methods

Table A.37: BAG3 method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	average tree depth	training speed (%)	prediction speed (%)
RTS	89.01	2	<1	0	8.53	11.5	5.3	55	66
RTSN	71.45	2	<1	0	8.99	11.7	6.0	70	77
RTC	60.33	1	<1	0	3.45	4.38	4.0	93	99
RTCN	53.22	1	<1	0	3.56	4.48	3.0	99	100
RRBFS	88.16	2	<1	0	5.19	10.4	10.7	38	58
RRBFC	85.51	2	<1	0	3.63	7.23	9.0	61	73
LED	73.94	6	<1	0	2.04	4.05	8.0	56	64
WAVE21	81.42	4	<1	0	4.32	8.61	10.0	64	75
WAVE40	81.17	3	<1	0	3.47	6.91	9.7	78	90
GENF1	95.07	8	<1	0	11.7	13.5	7.0	44	57
GENF2	92.61	2	<1	0	10.0	12.7	8.3	38	55
GENF3	97.51	2	<1	0	11.9	13.6	7.0	40	59
GENF4	93.44	2	<1	0	10.3	12.8	7.7	38	54
GENF5	86.83	2	<1	0	9.05	12.2	8.3	39	54
GENF6	89.90	2	<1	0	8.85	12.1	8.3	38	52
GENF7	96.35	3	<1	0	9.54	12.5	9.0	40	51
GENF8	99.39	12	<1	0	10.9	13.2	8.7	45	60
GENF9	95.45	3	<1	0	8.78	12.1	8.3	41	53
GENF10	99.88	49	<1	0	11.2	13.2	9.0	46	69
average	85.82	6	-	0	7.66	10.4	7.8	54	67

Table A.38: BAG3 method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	average tree depth	training speed (%)	prediction speed (%)
RTS	100.00	780	<10	65.5	214	514	18.0	8	42
RTSN	78.49	630	<10	1.39	3294	4332	15.7	18	64
RTC	83.17	370	<10	6.11	1931	2574	12.0	56	82
RTCN	59.75	190	<10	6.70	1926	2453	9.7	57	75
RRBFS	93.86	930	20	23.1	1501	3048	34.7	5	21
RRBFC	99.32	1010	<10	19.1	579	1197	28.3	24	52
LED	73.94	1230	30	16.1	418	868	17.0	13	37
WAVE21	85.07	850	20	18.1	1197	2430	33.0	21	57
WAVE40	85.02	700	<10	14.5	942	1913	27.7	35	74
GENF1	95.06	1200	40	42.1	1562	2778	16.7	6	49
GENF2	94.11	1130	20	23.5	2734	3676	17.7	6	40
GENF3	97.51	1470	30	74.7	968	1907	16.3	8	55
GENF4	94.68	1110	20	30.7	2402	3380	19.7	6	41
GENF5	92.83	950	<10	6.44	2987	4095	22.0	5	29
GENF6	93.33	1000	<10	9.48	3001	4050	18.3	5	34
GENF7	96.83	1180	20	33.4	2706	3452	19.3	6	41
GENF8	99.42	1650	90	105	770	1200	17.0	9	60
GENF9	96.79	1190	20	21.0	3236	3933	18.0	6	38
GENF10	99.89	1910	590	131	191	445	18.3	9	64
average	90.48	1025	-	34.1	1714	2539	20.0	16	50

Table A.39: BAG3 method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	average tree depth	training speed (%)	prediction speed (%)
RTS	99.99	560	?	259	0	474	14.3	6	29
RTSN	78.44	80	50	891	620	2021	14.0	3	61
RTC	84.54	100	<10	175	1364	2149	12.3	15	79
RTCN	61.63	80	20	173	1681	2389	11.0	26	92
RRBFS	94.17	230	?	1621	0	3242	46.3	1	10
RRBFC	99.39	130	60	346	249	1190	35.3	3	29
LED	73.98	270	?	276	0	551	17.0	3	10
WAVE21	85.92	170	130	743	226	1938	53.3	4	32
WAVE40	85.78	130	70	398	282	1359	52.0	7	45
GENF1	95.04	420	?	1108	0	1815	15.3	2	29
GENF2	94.04	300	?	1735	0	2340	20.3	2	27
GENF3	97.50	610	?	722	0	1273	14.0	3	32
GENF4	94.61	200	?	1281	0	1735	20.0	1	26
GENF5	92.77	170	?	1471	0	2315	22.0	1	19
GENF6	93.26	250	240	1786	82.2	2745	20.3	1	24
GENF7	96.80	270	?	1750	0	2275	21.7	2	23
GENF8	99.42	740	?	602	0	846	19.7	4	22
GENF9	96.79	220	190	1805	289	2616	21.3	1	30
GENF10	99.89	1370	?	283	0	392	19.7	6	26
average	90.73	332	-	917	252	1772	23.7	5	34

Table A.40: BAG5 method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	average tree depth	training speed (%)	prediction speed (%)
RTS	84.84	1	<1	0	7.99	10.5	4.4	45	62
RTSN	69.82	1	<1	0	8.05	10.5	5.0	62	75
RTC	54.13	1	<1	0	0.35	0.44	1.4	94	99
RTCN	52.96	1	<1	0	0.39	0.49	1.6	99	100
RRBFS	87.68	1	<1	0	4.95	9.85	9.6	28	45
RRBFC	76.66	1	<1	0	2.39	4.73	7.0	50	72
LED	73.35	2	<1	0	1.27	2.49	5.6	42	59
WAVE21	81.01	2	<1	0	3.67	7.29	8.2	53	69
WAVE40	80.30	1	<1	0	2.26	4.47	7.0	64	85
GENF1	95.07	3	<1	0	11.8	13.2	6.4	35	47
GENF2	92.18	1	<1	0	8.84	11.7	6.4	29	43
GENF3	97.51	2	<1	0	11.2	13.0	6.0	34	50
GENF4	91.61	1	<1	0	10.2	12.4	7.0	30	44
GENF5	78.36	1	<1	0	9.64	12.2	6.0	30	47
GENF6	90.64	1	<1	0	8.81	11.7	7.4	31	46
GENF7	96.18	2	<1	0	7.97	11.2	8.2	31	41
GENF8	99.40	9	<1	0	9.88	12.2	7.6	36	49
GENF9	94.86	2	<1	0	8.72	11.8	7.4	32	43
GENF10	99.88	43	<1	0	10.5	12.6	8.2	38	56
average	84.02	4	-	0	6.78	9.10	6.3	45	60

Table A.41: BAG5 method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	average tree depth	training speed (%)	prediction speed (%)
RTS	100.00	1240	<10	57.3	436	905	20.4	13	36
RTSN	78.48	740	<10	0.05	3340	4402	14.6	21	51
RTC	79.66	300	<10	4.56	2329	3055	11.2	46	77
RTCN	60.06	190	<10	5.98	2119	2694	9.8	58	74
RRBFS	93.93	830	<10	5.73	1744	3499	31.2	5	12
RRBFC	99.47	640	<10	17.6	673	1382	26.6	15	41
LED	73.98	760	20	16.2	447	927	16.0	8	26
WAVE21	85.19	820	<10	4.95	1580	3170	28.2	20	43
WAVE40	85.06	650	<10	8.20	1287	2591	25.0	33	62
GENF1	95.07	1160	30	12.1	2115	3582	16.4	6	39
GENF2	94.11	1170	<10	1.17	3299	4346	17.8	6	27
GENF3	97.51	1370	30	48.1	1384	2583	15.2	7	45
GENF4	94.68	1160	<10	3.03	3161	4236	18.6	6	27
GENF5	92.83	1130	<10	0.01	3041	4234	20.6	6	21
GENF6	93.34	1180	<10	0.06	3298	4358	18.0	6	28
GENF7	96.84	1220	<10	3.29	3555	4415	18.2	6	30
GENF8	99.43	1590	40	86.3	1255	1780	16.4	8	50
GENF9	96.82	1290	<10	0.44	3889	4618	17.6	7	29
GENF10	99.89	1530	220	127	266	540	17.4	7	55
average	90.33	998	-	21.2	2064	3017	18.9	15	41

Table A.42: BAG5 method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	average tree depth	training speed (%)	prediction speed (%)
RTS	99.99	380	?	394	0	711	14.2	4	20
RTSN	78.49	100	30	850	2479	4445	14.2	3	47
RTC	81.90	90	<10	171	2113	3107	11.6	15	76
RTCN	62.29	70	<10	170	2635	3607	10.8	23	88
RRBFS	94.29	160	140	1603	314	3833	41.4	1	7
RRBFC	99.56	120	30	342	423	1532	31.4	3	24
LED	73.97	200	?	278	0	555	17.0	2	7
WAVE21	86.14	130	80	734	425	2319	45.6	3	22
WAVE40	85.98	110	50	392	466	1716	45.8	6	34
GENF1	95.03	320	?	1529	0	2437	15.2	2	19
GENF2	94.09	230	190	1771	362	2902	20.2	1	22
GENF3	97.50	450	?	939	0	1604	13.4	2	23
GENF4	94.66	220	170	1782	499	3046	19.4	1	22
GENF5	92.83	170	120	1707	720	3878	22.2	1	14
GENF6	93.32	190	120	1726	689	3451	19.4	1	17
GENF7	96.83	210	160	1743	554	2993	21.4	1	20
GENF8	99.42	560	?	744	0	1009	19.6	3	15
GENF9	96.82	170	100	1732	1096	3523	20.0	1	19
GENF10	99.89	1010	?	360	0	493	18.8	5	16
average	90.68	257	-	998	672	2482	22.2	4	27

Table A.43: BAG10 method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	average tree depth	training speed (%)	prediction speed (%)
RTS	77.75	1	<1	0	6.99	9.14	3.2	36	57
RTSN	65.29	1	<1	0	6.17	7.80	3.0	54	69
RTC	56.31	1	<1	0	1.82	2.28	2.1	84	96
RTCN	52.96	1	<1	0	1.51	1.90	1.9	93	100
RRBFS	85.87	1	<1	0	4.33	8.56	7.7	20	31
RRBFC	64.87	1	<1	0	0.70	1.30	3.3	44	70
LED	10.00	1	<1	0	0.10	0.10	0.0	43	63
WAVE21	79.45	1	<1	0	2.29	4.48	5.6	40	64
WAVE40	69.71	1	<1	0	0.38	0.66	1.9	66	88
GENF1	95.03	1	<1	0	10.4	11.8	5.9	22	35
GENF2	93.22	1	<1	0	7.17	9.96	5.6	21	33
GENF3	97.51	1	<1	0	8.35	10.5	5.1	23	38
GENF4	86.02	1	<1	0	9.02	11.1	4.9	21	33
GENF5	78.47	1	<1	0	8.87	11.0	5.4	22	34
GENF6	87.55	1	<1	0	7.90	10.4	5.7	22	33
GENF7	95.54	1	<1	0	5.53	8.98	7.0	21	31
GENF8	99.37	4	<1	0	7.20	9.88	6.7	24	38
GENF9	93.73	1	<1	0	7.22	10.1	6.0	21	31
GENF10	99.88	19	<1	0	8.35	10.6	7.2	27	43
average	78.34	2	-	0	5.49	7.40	4.6	37	52

Table A.44: BAG10 method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	average tree depth	training speed (%)	prediction speed (%)
RTS	99.99	1070	<10	44.5	815	1551	19.3	11	26
RTSN	78.43	410	<10	0	3328	4396	13.6	20	38
RTC	79.82	360	<10	2.35	2833	3704	10.6	54	71
RTCN	60.07	230	<10	3.00	2823	3582	9.2	68	87
RRBFS	93.92	790	<10	0.15	1821	3642	29.0	5	7
RRBFC	99.64	740	<10	10.1	1170	2359	28.6	17	23
LED	73.96	990	<10	3.96	970	1948	16.0	11	15
WAVE21	85.14	760	<10	0.20	1711	3423	24.1	18	29
WAVE40	85.09	610	<10	1.64	1627	3257	22.3	30	48
GENF1	95.07	1260	<10	0.73	2585	4009	15.3	7	26
GENF2	94.11	770	<10	0	3444	4431	16.4	6	16
GENF3	97.52	1360	20	7.67	2084	3637	14.3	7	32
GENF4	94.67	690	<10	0	3367	4387	17.0	6	15
GENF5	92.79	510	<10	0	3025	4223	18.8	5	11
GENF6	93.33	630	<10	0	3246	4329	16.9	4	14
GENF7	96.84	960	<10	0	3773	4574	17.3	6	18
GENF8	99.43	1380	20	55.5	2097	2753	14.9	7	34
GENF9	96.82	630	<10	0	4000	4675	16.2	6	17
GENF10	99.89	1520	80	117	497	822	15.3	7	39
average	90.34	825	-	13.0	2380	3458	17.6	16	30

Table A.45: BAG10 method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	average tree depth	training speed (%)	prediction speed (%)
RTS	99.99	130	?	613	0	1073	13.5	1	10
RTSN	78.48	80	20	810	4262	6772	13.8	3	32
RTC	82.83	80	<10	165	3558	5019	11.0	13	68
RTCN	62.77	70	<10	162	4701	6229	10.3	21	82
RRBFS	94.43	110	70	1546	960	5012	39.3	1	3
RRBFC	99.71	100	<10	338	702	2079	29.6	3	15
LED	73.99	130	120	321	104	849	16.0	1	5
WAVE21	86.22	100	40	714	852	3131	41.1	3	13
WAVE40	86.03	90	30	380	821	2402	40.0	5	21
GENF1	95.05	210	190	1763	471	3363	14.6	1	22
GENF2	94.10	170	80	1720	1343	4076	18.3	1	12
GENF3	97.51	290	?	1433	0	2329	12.7	2	13
GENF4	94.66	160	80	1723	1615	4589	18.1	1	11
GENF5	92.86	130	50	1632	1954	5655	20.0	1	7
GENF6	93.36	140	40	1658	1879	5137	18.2	1	8
GENF7	96.85	160	70	1677	1788	4401	19.5	1	11
GENF8	99.42	340	?	1047	0	1358	19.0	2	8
GENF9	96.84	150	40	1646	3033	5703	18.8	1	11
GENF10	99.89	650	?	526	0	699	16.8	3	9
average	90.79	173	-	1046	1476	3678	20.6	3	19

Table A.46: BOOST3 method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	average tree depth	training speed (%)	prediction speed (%)
RTS	90.74	2	<1	0	8.55	11.5	5.3	49	63
RTSN	69.30	2	<1	0	9.22	11.8	5.7	65	76
RTC	56.23	1	<1	0	3.47	4.39	3.7	87	93
RTCN	53.55	1	<1	0	3.55	4.46	3.0	95	100
RRBFS	87.49	2	<1	0	5.19	10.4	11.3	28	48
RRBFC	86.85	2	<1	0	3.63	7.23	8.7	50	72
LED	73.88	6	<1	0	2.13	4.23	8.7	42	58
WAVE21	81.06	3	<1	0	4.28	8.53	9.7	51	73
WAVE40	81.08	3	<1	0	3.52	7.01	9.3	68	87
GENF1	93.61	8	<1	0	11.8	13.6	9.0	35	50
GENF2	88.27	2	<1	0	9.21	12.3	7.3	33	48
GENF3	96.70	2	<1	0	10.7	12.9	8.3	37	51
GENF4	87.85	2	<1	0	10.6	12.9	8.0	31	48
GENF5	79.86	2	<1	0	9.81	12.6	7.3	32	48
GENF6	91.03	2	<1	0	10.2	12.7	9.0	32	48
GENF7	96.14	3	<1	0	9.27	12.4	9.3	35	47
GENF8	99.38	11	<1	0	10.5	13.2	9.3	37	49
GENF9	95.12	4	<1	0	9.11	12.3	8.3	33	45
GENF10	99.87	77	<1	0	9.89	12.9	11.3	43	53
average	84.63	7	-	0	7.61	10.4	8.0	46	61

Table A.47: BOOST3 method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	average tree depth	training speed (%)	prediction speed (%)
RTS	100.00	1410	<10	50.8	857	1496	25.0	14	41
RTSN	78.43	550	<10	1.54	2992	4176	17.0	16	56
RTC	78.37	250	<10	6.48	1880	2475	11.7	39	64
RTCN	58.34	200	<10	6.73	1931	2459	10.0	62	83
RRBFS	93.24	540	<10	20.2	1540	3120	31.7	3	18
RRBFC	98.98	840	<10	12.1	1070	2165	30.7	20	48
LED	73.95	1140	30	11.9	593	1210	17.7	12	33
WAVE21	84.46	690	<10	12.0	1380	2783	28.3	17	52
WAVE40	84.28	590	<10	10.7	1151	2323	26.7	30	70
GENF1	91.98	1190	<10	14.7	2247	3593	19.0	6	43
GENF2	92.13	1080	<10	8.57	2485	3815	19.3	6	29
GENF3	95.44	1410	<10	26.2	2161	3341	20.0	7	48
GENF4	93.00	1020	<10	12.5	2434	3718	20.0	5	31
GENF5	91.62	950	<10	2.28	2534	3950	21.0	5	27
GENF6	92.11	970	<10	3.91	2720	4008	19.0	5	31
GENF7	96.13	1120	<10	13.1	2769	3854	20.7	6	31
GENF8	99.35	1350	<10	37.0	2143	3119	22.3	7	41
GENF9	96.49	1150	<10	7.54	3045	4090	19.7	6	30
GENF10	99.88	1570	20	44.5	1956	2911	25.7	7	50
average	89.38	948	-	15.9	1994	3085	21.3	14	43

Table A.48: BOOST3 method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	average tree depth	training speed (%)	prediction speed (%)
RTS	99.99	370	160	668	254	1486	20.3	4	31
RTSN	78.39	130	50	864	1247	3176	17.3	4	44
RTC	80.82	100	<10	174	1545	2370	12.3	15	79
RTCN	59.55	80	20	174	1595	2279	10.7	25	88
RRBFS	93.43	130	70	1354	676	4059	42.0	1	10
RRBFC	99.08	150	40	346	505	1701	36.3	4	34
LED	73.96	210	?	200	0	399	17.0	2	14
WAVE21	85.39	160	110	741	297	2076	52.0	4	30
WAVE40	85.16	130	60	395	390	1571	47.3	7	45
GENF1	92.97	150	60	1258	900	3507	19.3	1	33
GENF2	92.24	150	80	1402	842	3791	20.7	1	25
GENF3	96.18	140	60	1207	1050	3513	19.0	1	39
GENF4	93.13	130	70	1393	886	3768	20.0	1	23
GENF5	91.65	130	70	1529	748	3944	21.7	1	18
GENF6	91.97	120	60	1517	841	3828	20.7	1	21
GENF7	96.11	100	50	1384	1227	3948	22.3	1	22
GENF8	99.32	180	110	1262	801	3122	24.7	1	30
GENF9	96.34	90	40	1488	1487	4380	21.0	1	22
GENF10	99.87	310	200	1369	1222	3830	26.0	1	35
average	89.77	156	-	985	869	2987	24.8	4	34

Table A.49: BOOST5 method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	average tree depth	training speed (%)	prediction speed (%)
RTS	88.38	1	<1	0	7.89	10.4	4.8	38	56
RTSN	68.59	1	<1	0	8.24	10.6	4.4	55	70
RTC	59.23	1	<1	0	0.76	0.96	2.0	87	94
RTCN	53.55	1	<1	0	1.01	1.28	2.0	93	100
RRBFS	87.13	1	<1	0	5.07	10.1	9.2	19	37
RRBFC	79.87	1	<1	0	2.39	4.73	6.8	37	65
LED	73.87	1	<1	0	1.17	2.29	5.8	20	54
WAVE21	80.93	2	<1	0	3.72	7.39	8.4	40	64
WAVE40	80.48	1	<1	0	2.35	4.65	7.4	51	82
GENF1	93.72	1	<1	0	11.1	13.2	8.4	24	39
GENF2	86.18	1	<1	0	9.00	11.8	6.2	22	38
GENF3	96.40	5	<1	0	9.66	12.3	6.0	29	40
GENF4	93.26	1	<1	0	9.95	12.4	5.6	24	38
GENF5	68.72	1	<1	0	10.4	12.5	5.6	23	40
GENF6	89.24	1	<1	0	8.91	11.8	7.0	24	40
GENF7	95.93	2	<1	0	9.57	13.3	8.6	25	35
GENF8	99.36	7	<1	0	9.73	12.4	8.2	30	39
GENF9	94.93	2	<1	0	9.08	12.0	7.6	25	35
GENF10	99.88	42	<1	0	9.58	12.8	9.6	34	45
average	83.67	4	-	0	6.82	9.30	6.5	37	53

Table A.50: BOOST5 method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	average tree depth	training speed (%)	prediction speed (%)
RTS	99.99	1270	<10	27.1	1673	2730	23.4	13	31
RTSN	78.39	680	<10	0.06	2996	4231	17.0	19	47
RTC	82.39	350	<10	4.14	2354	3144	12.4	53	76
RTCN	59.27	190	<10	5.88	2135	2718	9.8	57	76
RRBFS	93.01	600	<10	3.37	1776	3559	29.6	3	9
RRBFC	99.19	800	<10	6.64	1420	2854	30.0	19	31
LED	73.97	930	20	7.94	785	1586	17.6	10	24
WAVE21	84.49	670	<10	2.69	1644	3294	27.6	16	38
WAVE40	84.32	570	<10	4.27	1496	3000	27.6	29	61
GENF1	90.93	1460	<10	1.11	2490	3948	19.4	8	31
GENF2	92.01	1190	<10	0.19	2485	3965	18.4	6	18
GENF3	95.47	1520	<10	8.12	2270	3718	19.2	8	30
GENF4	93.17	1240	<10	0.45	2606	4015	18.4	7	21
GENF5	91.80	1100	<10	0	2408	3929	19.8	6	17
GENF6	92.19	1180	<10	0.02	2594	4017	18.4	6	18
GENF7	96.14	1330	<10	0.35	3137	4264	20.4	7	23
GENF8	99.34	1710	<10	16.8	2679	3746	22.8	9	29
GENF9	96.43	1280	<10	0.09	3256	4326	20.2	7	21
GENF10	99.87	1220	<10	26.0	2362	3449	25.4	6	36
average	89.60	1015	-	6.07	2240	3500	20.9	15	34

Table A.51: BOOST5 method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	average tree depth	training speed (%)	prediction speed (%)
RTS	99.98	260	30	816	1231	3172	20.8	3	22
RTSN	78.34	100	30	846	1760	4029	17.8	3	31
RTC	84.24	90	<10	170	2177	3295	12.8	14	74
RTCN	60.12	50	<10	176	1229	1823	10.0	16	81
RRBFS	93.30	80	40	1432	1152	5168	37.4	1	6
RRBFC	99.30	100	20	345	775	2238	30.6	2	25
LED	73.92	190	?	306	0	611	17.0	2	10
WAVE21	85.37	130	60	723	721	2888	41.8	3	21
WAVE40	85.06	110	40	384	741	2250	43.4	6	35
GENF1	93.23	80	30	1374	1762	5184	19.6	0	20
GENF2	92.48	80	40	1487	1393	5083	19.2	0	14
GENF3	96.41	80	30	1381	1719	5170	19.2	0	22
GENF4	93.31	80	30	1487	1672	5362	19.2	0	15
GENF5	91.66	80	40	1562	1456	5448	21.0	0	11
GENF6	92.07	80	40	1574	1618	5510	19.6	0	12
GENF7	96.02	60	20	1499	2110	5364	21.0	0	14
GENF8	99.28	140	50	1438	2123	5348	24.0	1	22
GENF9	96.19	60	20	1573	2547	6021	20.4	0	15
GENF10	99.86	230	110	1600	2168	5602	26.0	1	27
average	90.01	109	-	1062	1492	4188	23.2	3	25

Table A.52: BOOST10 method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	average tree depth	training speed (%)	prediction speed (%)
RTS	78.61	1	<1	0	6.25	8.08	3.6	28	44
RTSN	67.94	1	<1	0	6.31	8.05	3.4	47	62
RTC	57.63	1	<1	0	1.15	1.45	1.6	81	92
RTCN	55.70	1	<1	0	0.75	0.95	1.5	93	100
RRBFS	86.36	1	<1	0	4.43	8.76	7.7	13	25
RRBFC	62.93	1	<1	0	0.56	1.02	2.6	36	61
LED	9.99	1	<1	0	0.10	0.10	0.0	32	61
WAVE21	80.19	1	<1	0	2.21	4.32	5.7	31	57
WAVE40	75.39	1	<1	0	0.42	0.74	2.2	55	85
GENF1	93.96	1	<1	0	16.8	20.7	7.1	14	23
GENF2	71.77	1	<1	0	9.45	11.6	3.9	17	29
GENF3	95.91	1	<1	0	10.5	13.8	5.6	17	26
GENF4	85.44	1	<1	0	10.5	12.6	4.4	16	26
GENF5	65.92	1	<1	0	9.22	11.1	4.4	16	29
GENF6	88.64	1	<1	0	8.84	11.1	5.2	16	26
GENF7	95.57	1	<1	0	12.9	17.8	7.6	14	21
GENF8	99.36	5	<1	0	12.8	16.2	6.9	18	23
GENF9	94.46	1	<1	0	9.19	12.6	6.6	15	23
GENF10	99.87	15	<1	0	10.0	13.8	7.8	22	31
average	77.14	2	-	0	6.97	9.20	4.6	31	44

Table A.53: BOOST10 method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	average tree depth	training speed (%)	prediction speed (%)
RTS	99.99	860	<10	7.96	2315	3640	23.0	9	15
RTSN	78.34	390	<10	0	2975	4219	15.9	17	33
RTC	76.14	330	<10	2.00	2990	3845	10.4	51	71
RTCN	61.09	210	<10	3.12	2792	3544	9.5	65	80
RRBFS	92.75	630	<10	0.06	1821	3643	26.4	4	6
RRBFC	99.32	610	<10	2.51	1664	3334	26.4	14	19
LED	73.92	710	<10	0.98	1065	2133	16.9	8	21
WAVE21	84.44	610	<10	0.07	1713	3426	25.4	15	27
WAVE40	84.32	490	<10	0.74	1674	3350	27.3	25	41
GENF1	90.83	1300	<10	0.06	2593	4014	18.6	7	15
GENF2	92.07	780	<10	0	2439	3944	17.6	5	10
GENF3	95.49	1760	<10	0.29	2780	4095	18.9	9	21
GENF4	92.88	610	<10	0	2594	4017	16.3	5	11
GENF5	91.75	470	<10	0	2375	3912	17.6	4	8
GENF6	92.26	570	<10	0	2601	4019	16.8	5	9
GENF7	96.15	870	<10	0	3118	4260	19.3	6	11
GENF8	99.31	1310	<10	5.98	2983	4092	21.5	7	12
GENF9	96.40	600	<10	0	3213	4306	18.0	5	11
GENF10	99.87	2360	<10	10.6	2753	3909	23.4	11	24
average	89.33	814	-	1.81	2445	3774	19.4	14	23

Table A.54: BOOST10 method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	average tree depth	training speed (%)	prediction speed (%)
RTS	99.99	200	<10	827	3304	6317	20.9	2	15
RTSN	78.36	80	20	803	3152	6240	17.7	2	20
RTC	80.11	70	<10	163	4111	5671	11.3	12	64
RTCN	61.73	60	<10	166	3766	5055	10.7	20	79
RRBFS	93.30	50	20	1454	2163	7234	34.4	0	3
RRBFC	99.52	140	<10	330	2304	5267	26.5	3	15
LED	73.90	120	100	318	71.0	778	16.0	1	6
WAVE21	85.42	70	30	710	931	3281	32.6	2	12
WAVE40	85.03	80	20	369	1281	3300	28.9	4	24
GENF1	93.51	40	<10	1468	2910	7247	19.9	0	11
GENF2	92.65	50	20	1515	2961	7882	19.5	0	7
GENF3	96.60	40	<10	1505	3494	7624	19.8	0	16
GENF4	93.40	50	20	1508	3274	8098	17.9	0	7
GENF5	91.68	50	20	1563	2860	7872	19.2	0	6
GENF6	92.15	50	20	1546	3316	8238	18.8	0	6
GENF7	96.08	40	<10	1523	3986	8262	20.5	0	7
GENF8	99.23	90	30	1510	3905	8060	25.0	1	12
GENF9	96.14	40	<10	1573	4455	8883	19.8	0	7
GENF10	99.85	180	50	1613	4643	9139	25.2	1	17
average	89.93	79	-	1077	2994	6550	21.3	3	18

Table A.55: HOT3 method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	95.85	7	<1	0	10.1	12.8	8	67	77
RTSN	73.71	4	<1	0	10.2	12.8	10	77	86
RTC	64.79	3	<1	0	8.90	10.8	5	89	78
RTCN	53.82	1	<1	0	4.80	6.07	5	64	82
RRBFS	87.77	4	<1	0	5.06	10.1	18	41	63
RRBFC	79.84	1	<1	0	3.49	6.96	42	35	76
LED	73.91	21	<1	0	4.11	6.16	11	83	89
WAVE21	81.23	4	<1	0	6.95	10.4	11	100	100
WAVE40	81.14	4	<1	0	6.63	9.94	12	21	25
GENF1	95.06	8	<1	0	11.7	13.7	10	45	61
GENF2	93.47	3	<1	0	11.2	13.5	12	37	52
GENF3	97.48	27	<1	0	12.5	14.1	7	49	67
GENF4	93.80	4	<1	0	11.9	13.8	12	45	63
GENF5	85.09	2	<1	0	9.34	12.4	8	37	58
GENF6	91.95	5	<1	0	11.3	13.5	11	43	59
GENF7	96.40	5	<1	0	8.86	11.5	12	46	60
GENF8	99.40	20	<1	0	11.0	13.7	9	50	63
GENF9	95.85	9	<1	0	8.66	12.0	11	54	66
GENF10	99.88	146	<1	0	11.2	13.5	14	50	63
average	86.34	15	-	0	8.83	11.5	12	54	68

Table A.56: HOT3 method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	920	200	80.8	44.4	230	16	10	55
RTSN	78.48	380	20	26.1	2127	2813	23	9	71
RTC	84.09	290	<10	9.15	1162	1582	12	50	88
RTCN	63.06	160	<10	7.27	1741	2218	10	51	92
RRBFS	93.61	680	40	73.4	749	1645	51	3	21
RRBFC	99.13	730	<10	21.6	392	826	37	17	56
LED	73.95	810	50	19.2	202	442	17	16	62
WAVE21	85.07	550	20	32.5	728	1521	34	29	100
WAVE40	84.91	470	<10	20.8	555	1152	30	7	20
GENF1	95.06	900	40	86.7	800	1564	19	4	53
GENF2	94.09	820	30	70.6	1489	2180	22	4	45
GENF3	97.52	1210	250	116	332	835	17	6	68
GENF4	94.67	960	50	93.7	1031	1557	26	5	55
GENF5	92.62	470	20	46.2	1626	2647	32	2	26
GENF6	93.31	860	30	73.0	1597	2169	20	4	45
GENF7	96.83	930	30	67.8	1870	2365	21	4	42
GENF8	99.42	1240	410	131	186	436	18	6	65
GENF9	96.82	1070	40	88.3	1292	1720	20	5	49
GENF10	99.89	1590	1500	152	13.9	243	19	7	28
average	90.66	792	-	64.0	944	1481	23	13	55

Table A.57: HOT3 method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	770	?	120	0	219	16	9	46
RTSN	78.45	140	110	904	239	1489	23	4	76
RTC	84.53	60	<10	177	649	1167	13	11	77
RTCN	64.23	50	20	178	454	846	11	19	80
RRBFS	93.93	280	?	932	0	1864	56	1	14
RRBFC	99.21	90	80	354	50.2	809	43	2	30
LED	73.97	260	?	170	0	340	16	5	19
WAVE21	85.96	160	?	668	0	1337	53	9	75
WAVE40	85.80	100	100	402	12.4	830	50	2	13
GENF1	95.04	280	?	514	0	840	18	1	43
GENF2	94.07	300	?	980	0	1449	22	1	30
GENF3	97.51	840	?	368	0	676	16	4	52
GENF4	94.65	400	?	759	0	1024	27	2	40
GENF5	92.71	190	?	1519	0	2650	38	1	15
GENF6	93.29	290	?	1074	0	1452	24	1	32
GENF7	96.79	200	?	726	0	968	18	1	27
GENF8	99.42	1000	?	286	0	394	18	5	36
GENF9	96.77	190	?	724	0	936	19	1	33
GENF10	99.89	1570	?	164	0	240	19	7	28
average	90.85	377	-	580	73.9	1028	26	5	40

Table A.58: HOT5 method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	95.85	7	<1	0	10.1	12.8	8	68	78
RTSN	73.71	4	<1	0	10.2	12.8	10	77	86
RTC	64.79	3	<1	0	8.90	10.8	5	74	75
RTCN	54.64	1	<1	0	6.09	7.65	3	79	84
RRBFS	87.93	2	<1	0	4.97	9.90	15	30	51
RRBFC	78.63	1	<1	0	4.16	8.26	26	33	73
LED	73.91	21	<1	0	4.11	6.16	11	87	97
WAVE21	81.23	4	<1	0	6.95	10.4	11	100	100
WAVE40	81.14	4	<1	0	6.63	9.94	12	21	26
GENF1	95.06	8	<1	0	11.7	13.7	10	45	59
GENF2	93.47	3	<1	0	11.2	13.5	12	37	54
GENF3	97.48	27	<1	0	12.5	14.1	7	49	64
GENF4	93.80	4	<1	0	11.9	13.8	12	44	59
GENF5	84.25	1	<1	0	10.4	13.1	10	29	46
GENF6	91.95	5	<1	0	11.3	13.5	11	42	56
GENF7	96.38	6	<1	0	10.5	13.1	11	42	55
GENF8	99.40	16	<1	0	11.3	13.5	9	41	50
GENF9	95.77	8	<1	0	9.81	12.7	12	43	54
GENF10	99.88	146	<1	0	11.2	13.5	14	50	64
average	86.28	14	-	0	9.15	11.8	11	52	65

Table A.59: HOT5 method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	910	190	80.2	46.7	233	16	10	49
RTSN	78.47	380	<10	25.1	2170	2867	23	10	74
RTC	84.10	280	<10	9.12	1160	1580	12	47	82
RTCN	63.83	140	<10	7.43	1723	2196	10	44	83
RRBFS	93.83	530	20	63.6	883	1893	46	2	15
RRBFC	99.19	640	<10	19.6	518	1076	33	15	41
LED	73.96	730	40	18.3	226	489	17	15	59
WAVE21	85.13	500	20	28.6	854	1765	31	26	100
WAVE40	84.95	450	<10	19.4	632	1302	29	7	20
GENF1	95.07	860	40	83.4	858	1649	19	4	51
GENF2	94.08	810	30	68.9	1530	2228	22	4	42
GENF3	97.51	1220	230	112	389	934	17	6	66
GENF4	94.67	980	50	90.0	1103	1659	25	5	55
GENF5	92.70	350	<10	35.6	1983	3009	31	2	22
GENF6	93.33	810	30	72.3	1591	2177	20	4	45
GENF7	96.83	850	30	67.1	1883	2384	20	4	41
GENF8	99.42	1290	320	129	223	484	18	6	62
GENF9	96.82	1040	40	87.1	1315	1752	20	5	43
GENF10	99.89	1480	1390	152	14.4	244	19	7	27
average	90.72	750	-	61.5	1005	1575	23	12	51

Table A.60: HOT5 method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	780	?	122	0	224	16	9	46
RTSN	78.38	90	?	856	0	1114	21	2	72
RTC	84.58	60	<10	177	657	1180	13	11	77
RTCN	65.61	50	<10	176	878	1418	11	16	73
RRBFS	94.16	210	?	1123	0	2247	56	1	9
RRBFC	99.15	40	40	361	6.60	734	40	1	15
LED	73.91	170	?	106	0	213	15	4	19
WAVE21	86.03	130	?	739	0	1478	49	7	65
WAVE40	85.86	100	90	399	95.1	989	46	2	12
GENF1	95.06	420	?	719	0	1211	19	2	42
GENF2	94.07	180	?	704	0	1044	21	1	28
GENF3	97.51	780	?	386	0	706	16	4	51
GENF4	94.64	230	?	567	0	743	25	1	39
GENF5	92.78	140	?	1749	0	2922	36	1	12
GENF6	93.28	180	?	779	0	1054	21	1	29
GENF7	96.79	290	?	1129	0	1508	19	1	22
GENF8	99.42	910	?	288	0	396	18	4	33
GENF9	96.78	290	?	1004	0	1327	19	1	28
GENF10	99.89	1490	?	168	0	245	19	7	27
average	90.94	344	-	608	86.1	1092	25	4	37

Table A.61: HOT10 method with 100KB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	95.85	7	<1	0	10.1	12.8	8	68	78
RTSN	73.71	4	<1	0	10.2	12.8	10	77	86
RTC	64.79	3	<1	0	8.90	10.8	5	74	80
RTCN	54.64	2	<1	0	8.92	10.8	3	90	88
RRBFS	87.33	2	<1	0	7.81	11.7	13	26	42
RRBFC	71.98	1	<1	0	7.07	10.6	24	34	49
LED	73.91	21	<1	0	4.11	6.16	11	87	93
WAVE21	81.23	4	<1	0	6.95	10.4	11	100	100
WAVE40	81.14	4	<1	0	6.63	9.94	12	21	26
GENF1	95.06	8	<1	0	11.7	13.7	10	45	61
GENF2	93.47	3	<1	0	11.2	13.5	12	37	51
GENF3	97.48	27	<1	0	12.5	14.1	7	49	65
GENF4	93.80	4	<1	0	11.9	13.8	12	43	63
GENF5	84.25	1	<1	0	10.4	13.1	10	30	46
GENF6	91.95	5	<1	0	11.3	13.5	11	41	57
GENF7	96.38	6	<1	0	10.5	13.1	11	39	50
GENF8	99.40	16	<1	0	11.3	13.5	9	41	49
GENF9	95.77	8	<1	0	9.81	12.7	12	45	57
GENF10	99.88	146	<1	0	11.2	13.5	14	39	63
average	85.90	14	-	0	9.60	12.1	11	52	63

Table A.62: HOT10 method with 32MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	880	190	80.5	45.9	232	16	10	57
RTSN	78.48	370	<10	24.5	2194	2901	23	9	75
RTC	83.99	250	<10	8.34	1495	1894	12	43	81
RTCN	63.54	150	<10	6.75	1872	2381	9	47	89
RRBFS	93.80	420	20	59.2	943	2005	44	2	11
RRBFC	99.20	470	<10	16.3	723	1479	32	11	27
LED	73.97	620	20	19.9	333	706	16	13	43
WAVE21	85.13	380	<10	26.2	922	1896	30	20	97
WAVE40	84.93	400	<10	17.3	752	1538	28	6	19
GENF1	95.07	840	40	82.8	872	1669	19	4	53
GENF2	94.08	750	30	69.2	1523	2216	22	4	41
GENF3	97.51	1180	200	106	473	1074	17	5	66
GENF4	94.67	930	40	86.1	1177	1760	25	4	48
GENF5	92.72	270	<10	34.6	2003	3032	31	1	16
GENF6	93.34	780	30	65.6	1756	2379	20	4	41
GENF7	96.83	840	30	65.1	1933	2444	20	4	34
GENF8	99.42	1210	260	129	225	488	18	6	61
GENF9	96.82	970	30	89.3	1246	1677	20	5	40
GENF10	99.89	1420	1180	145	44.0	276	19	6	38
average	90.70	691	-	59.6	1081	1687	22	11	49

Table A.63: HOT10 method with 400MB memory limit.

dataset	accuracy (%)	training examples (millions)	examples to full memory (millions)	active leaves (hundreds)	inactive leaves (hundreds)	total nodes (hundreds)	tree depth	training speed (%)	prediction speed (%)
RTS	99.99	750	?	122	0	223	16	8	46
RTSN	78.45	120	90	903	289	1553	22	3	74
RTC	84.61	60	<10	177	668	1194	13	11	77
RTCN	65.31	40	<10	176	957	1523	11	14	71
RRBFS	94.20	180	?	1326	0	2653	55	1	6
RRBFC	99.36	60	20	342	348	1381	38	1	16
LED	73.95	150	?	217	0	434	15	3	10
WAVE21	86.12	110	100	751	134	1770	46	6	52
WAVE40	85.95	80	60	402	138	1080	43	1	10
GENF1	95.06	400	?	734	0	1234	19	2	40
GENF2	94.07	180	?	757	0	1125	21	1	28
GENF3	97.51	470	?	287	0	495	14	2	51
GENF4	94.65	350	?	862	0	1152	27	2	30
GENF5	92.76	80	?	1396	0	2326	32	0	10
GENF6	93.31	250	?	1193	0	1639	23	1	24
GENF7	96.80	180	?	758	0	1022	18	1	19
GENF8	99.42	550	?	215	0	287	18	3	31
GENF9	96.76	170	?	769	0	1007	19	1	25
GENF10	99.89	1360	?	182	0	265	19	6	24
average	90.96	292	-	609	133	1177	25	4	34

References

[1] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. On demand classification of data streams. In *Knowledge Discovery and Data Mining*, pages 503–508, 2004.

[2] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, 1993.

[3] Rakesh Agrawal and Arun Swami. A one-pass space-efficient algorithm for finding quantiles. In *International Conference on Management of Data*, 1995.

[4] Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102–116, 2002.

[5] Khaled Alsabti, Sanjay Ranka, and Vineet Singh. A one-pass algorithm for accurately estimating quantiles for disk-resident data. In *International Conference on Very Large Databases*, pages 346–355, 1997.

[6] S. Ansari, S.G. Rajeev, and H.S. Chandrashekar. Packet sniffing: A brief introduction. *IEEE Potentials*, 21(5):17–19, 2002.

[7] A. Asuncion and D. J. Newman. UCI Machine Learning Repository [http://www.ics.uci.edu/∼mlearn/mlrepository.html]. University of California, Irvine, School of Information and Computer Sciences, 2007.

[8] Jürgen Beringer and Eyke Hüllermeier. An efficient algorithm for instance-based learning on data streams. In *Industrial Conference on Data Mining*, pages 34–48, 2007.

[9] Remco R. Bouckaert. Choosing between two learning algorithms based on calibrated tests. In *International Conference on Machine Learning*, pages 51–58, 2003.

[10] Remco R. Bouckaert. Voting massive collections of bayesian network classifiers for data streams. In *Australian Joint Conference on Artificial Intelligence*, pages 243–252, 2006.

[11] Damien Brain and Geoffrey I. Webb. The need for low bias algorithms in classification learning from large data sets. In *European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 62–73, 2002.

[12] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[13] Leo Breiman. Arcing classifiers. *The Annals of Statistics*, 26(3):801–824, 1998.

[14] Leo Breiman. Rejoinder to discussion of the paper "arcing classifiers". *The Annals of Statistics*, 26(3):841–849, 1998.

[15] Leo Breiman. Pasting bites together for prediction in large data sets and on-line. *Machine Learning*, 36(1/2):85–103, 1999.

[16] Leo Breiman. Prediction games and arcing algorithms. *Neural Computation*, 11(7):1493–1517, 1999.

[17] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[18] Leo Breiman, Jerome Friedman, R. Olshen, and Charles J. Stone. *Classification and Regression Trees.* Wadsworth and Brooks, Monterey, CA, 1984.

[19] S Terry Brugger. KDD Cup '99 dataset (Network Intrusion) considered harmful, KDnuggets newsletter, 07(18), 15 September 2007 [http://www.kdnuggets.com/news/2007/n18/4i.html].

[20] Nader H. Bshouty and Dmitry Gavinsky. On boosting with polynomially bounded distributions. *Journal of Machine Learning Research*, 3:483–506, 2002.

[21] Wray Buntine. Learning classification trees. *Statistics and Computing*, 2(2):63–73, 1992.

[22] Christopher J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

[23] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *International Conference on Machine Learning*, pages 161–168, 2006.

[24] Tony F. Chan and John Gregg Lewis. Computing standard deviations: Accuracy. *Communications of the ACM*, 22(9):526–531, 1979.

[25] Fang Chu and Carlo Zaniolo. Fast and light boosting for adaptive mining of data streams. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 282–292, 2004.

[26] Thomas G. Dietterich. Approximate statistical test for comparing supervised classification learning algorithms. *Neural Computation*, 10(7):1895–1923, 1998.

[27] Thomas G. Dietterich. Machine learning research: Four current directions. *The AI Magazine*, 18(4):97–136, 1998.

[28] Thomas G. Dietterich. Ensemble methods in machine learning. *Lecture Notes in Computer Science*, 1857:1–15, 2000.

[29] Thomas G. Dietterich. An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine Learning*, 40(2):139–157, 2000.

[30] Thomas G. Dietterich and Ghulum Bakiri. Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research*, 2:263–286, 1995.

[31] Carlos Domingo and Osamu Watanabe. MadaBoost: A modification of AdaBoost. In *ACM Annual Workshop on Computational Learning Theory*, pages 180–189, 2000.

[32] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *International Conference on Knowledge Discovery and Data Mining*, pages 71–80, 2000.

[33] Pedro Domingos and Michael J. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29(2/3):103–130, 1997.

[34] James Dougherty, Ron Kohavi, and Mehran Sahami. Supervised and unsupervised discretization of continuous features. In *International Conference on Machine Learning*, pages 194–202, 1995.

[35] Bradley Efron. Estimating the error rate of a prediction rule: Improvement on cross-validation. *Journal of the American Statistical Association*, 78(382):316–330, 1983.

[36] Wei Fan, Salvatore J. Stolfo, and Junxin Zhang. The application of adaboost for distributed, scalable and on-line learning. In *International Conference on Knowledge Discovery and Data Mining*, pages 362–366, 1999.

[37] Usama M. Fayyad and Keki B. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *International Joint Conference on Artificial Intelligence*, pages 1022–1027, 1993.

[38] Alan Fern and Robert Givan. Online ensemble learning: An empirical study. *Machine Learning*, 53(1/2):71–109, 2003.

[39] Francisco Ferrer-Troyano, Jesús S. Aguilar-Ruiz, and José C. Riquelme. Discovering decision rules from numerical data streams. In *ACM Symposium on Applied computing*, pages 649–653, 2004.

[40] Francisco Ferrer-Troyano, Jesús S. Aguilar-Ruiz, and José C. Riquelme. Data streams classification by incremental rule learning with parameterized generalization. In *ACM Symposium on Applied Computing*, pages 657–661, 2006.

[41] Yoav Freund. Boosting a weak learning algorithm by majority. *Information and Computation*, 121(2):256–285, 1995.

[42] Yoav Freund. An adaptive version of the boost by majority algorithm. *Machine Learning*, 43(3):293–318, 2001.

[43] Yoav Freund and Llew Mason. The alternating decision tree learning algorithm. In *International Conference on Machine Learning*, pages 124–133, 1999.

[44] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[45] Yoav Freund and Robert E. Schapire. Discussion of the paper "arcing classifiers" by Leo Breiman. *The Annals of Statistics*, 26(3):824–832, 1998.

[46] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting. *The Annals of Statistics*, 28:337–374, 2000.

[47] Mohamed Medhat Gaber, Shonali Krishnaswamy, and Arkady Zaslavsky. On-board mining of data streams in sensor networks. In Sanghamitra Bandyopadhyay, Ujjwal Maulik, Lawrence B. Holder, and Diane J. Cook, editors, *Advanced Methods for Knowledge Discovery from Complex Data*, pages 307–335. Springer, Berlin Heidelberg, 2005.

[48] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. A survey of classification methods in data streams. In Charu C. Aggarwal, editor, *Data Streams: Models and Algorithms*, chapter 3. Springer, New York, 2007.

[49] João Gama and Mohamed Medhat Gaber, editors. *Learning from Data Streams: Processing Techniques in Sensor Networks*. Springer, 2007.

[50] João Gama, Pedro Medas, and Ricardo Rocha. Forest trees for on-line data. In *ACM Symposium on Applied Computing*, pages 632–636, 2004.

[51] João Gama and Carlos Pinto. Discretization from data streams: Applications to histograms and data mining. In *ACM Symposium on Applied Computing*, pages 662–667, 2006.

[52] João Gama, Ricardo Rocha, and Pedro Medas. Accurate decision trees for mining high-speed data streams. In *International Conference on Knowledge Discovery and Data Mining*, pages 523–528, 2003.

[53] João Gama and Pedro Pereira Rodrigues. Stream-based electricity load forecast. In *European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 446–453, 2007.

[54] Johannes Gehrke, Raghu Ramakrishnan, and Venkatesh Ganti. RainForest - a framework for fast decision tree construction of large datasets. *Data Mining and Knowledge Discovery*, 4(2/3):127–162, 2000.

[55] Stuart Geman, Elie Bienenstock, and René Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4(1):1–58, 1992.

[56] Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. In *ACM Special Interest Group on Management Of Data Conference*, pages 58–66, 2001.

[57] Robert L. Grossman, Chandrika Kamath, Philip Kegelmeyer, Vipin Kumar, and Raju R. Namburu, editors. *Data Mining for Scientific and Engineering Applications*. Kluwer Academic Publishers, 2001.

[58] Sudipto Guha, Adam Meyerson, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams: Theory and practice. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):515–528, 2003.

[59] Sudipto Guha, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams. In *IEEE Symposium on Foundations of Computer Science*, pages 359–366, 2000.

[60] Sule Gündüz and M. Tamer Özsu. A web page prediction model based on click-stream tree representation of user behavior. In *International Conference on Knowledge Discovery and Data Mining*, pages 535–540, 2003.

[61] Jiawei Han and Kevin Chang. Data mining for web intelligence. *IEEE Computer*, 35(11):64–70, 2002.

[62] David J. Hand. Mining personal banking data to detect fraud. In Paula Brito, Guy Cucumel, Patrice Bertrand, and Francisco de Carvalho, editors, *Selected Contributions in Data Analysis and Classification*, pages 377–386. Springer, 2007.

[63] Lars Kai Hansen and Peter Salamon. Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(10):993–1001, 1990.

[64] J. A. Hartigan and M. A. Wong. A k-means clustering algorithm. *Applied Statistics*, 28(1):100–108, 1979.

[65] S. Hettich and S. D. Bay. The UCI KDD archive [http://kdd.ics.uci.edu/]. University of California, Irvine, School of Information and Computer Sciences, 1999.

[66] J. Hilden. Statistical diagnosis based on conditional independence does not require it. *Computers in Biology and Medicine*, 14(4):429–435, 1984.

[67] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[68] Geoffrey Holmes, Richard Kirkby, and Bernhard Pfahringer. Stress-testing hoeffding trees. In *European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 495–502, 2005.

[69] Osnat Horovitz, Shonali Krishnaswamy, and Mohamed Medhat Gaber. A fuzzy approach for interpretation of ubiquitous data stream clustering and its application in road safety. *Intelligent Data Analysis*, 11(1):89–108, 2007.

[70] Wolfgang Hoschek, Francisco Javier Janez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data management in an international data grid project. In *International Workshop on Grid Computing*, pages 77–90, 2000.

[71] Geoff Hulten and Pedro Domingos. Mining complex models from arbitrarily large databases in constant time. In *International Conference on Knowledge Discovery and Data Mining*, pages 525–531, 2002.

[72] Geoff Hulten and Pedro Domingos. VFML – a toolkit for mining high-speed time-changing data streams [http://www.cs.washington.edu/dm/vfml/]. 2003.

[73] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *International Conference on Knowledge Discovery and Data Mining*, pages 97–106, 2001.

[74] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is np-complete. *Information Processing Letters*, 5(1):15–17, 1976.

[75] Tomasz Imielinski and Badri Nath. Wireless graffiti – data, data everywhere. In *International Conference on Very Large Databases*, pages 9–19, 2002.

[76] Raj Jain and Imrich Chlamtac. The P^2 algorithm for dynamic calculation of quantiles and histograms without storing observations. *Communications of the ACM*, 28(10):1076–1085, 1985.

[77] Ruoming Jin and Gagan Agrawal. Efficient decision tree construction on streaming data. In *Knowledge Discovery and Data Mining*, pages 571–576, 2003.

[78] Hillol Kargupta, Ruchita Bhargava, Kun Liu, Michael Powers, Patrick Blair, Samuel Bushra, James Dull, Kakali Sarkar, Martin Klein, Mitesh Vasa, and David Handy. VEDAS: A mobile and distributed data stream mining system for real-time vehicle monitoring. In *SIAM International Conference on Data Mining*, 2004.

[79] Hillol Kargupta, Byung-Hoon Park, Sweta Pittie, Lei Liu, Deepali Kushraj, and Kakali Sarkar. MobiMine: Monitoring the stock market from a PDA. *SIGKDD Explorations*, 3(2):37–46, 2002.

[80] Michael Kearns and Leslie Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *Journal of the ACM*, 41(1):67–95, 1994.

[81] Maleq Khan, Qin Ding, and William Perrizo. k-nearest neighbor classification on spatial data streams using p-trees. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 517–518, 2002.

[82] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International Joint Conference on Artificial Intelligence*, pages 1137–1145, 1995.

[83] Ron Kohavi. Scaling up the accuracy of Naive-Bayes classifiers: a decision-tree hybrid. In *International Conference on Knowledge Discovery and Data Mining*, pages 202–207, 1996.

[84] Ron Kohavi and Clayton Kunz. Option decision trees with majority votes. In *International Conference on Machine Learning*, pages 161–169, 1997.

[85] Ron Kohavi and Foster J. Provost. Applications of data mining to electronic commerce. *Data Mining and Knowledge Discovery*, 5(1/2):5–10, 2001.

[86] Ron Kohavi and David H. Wolpert. Bias plus variance decomposition for zero-one loss functions. In *International Conference on Machine Learning*, pages 275–283, 1996.

[87] Pat Langley, Wayne Iba, and Kevin Thompson. An analysis of bayesian classifiers. In *National Conference on Artificial Intelligence*, pages 223–228, 1992.

[88] Mark Last. Online classification of nonstationary data streams. *Intelligent Data Analysis*, 6(2):129–147, 2002.

[89] Yan-Nei Law and Carlo Zaniolo. An adaptive nearest neighbor classification algorithm for data streams. In *European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 108–120, 2005.

[90] Wenke Lee, Salvatore J. Stolfo, and Kui W. Mok. Mining in a data-flow environment: experience in network intrusion detection. In *International Conference on Knowledge Discovery and Data Mining*, pages 114–124, 1999.

[91] Nick Littlestone and Manfred K. Warmuth. The weighted majority algorithm. *Information and Computation*, 108(2):212–261, 1994.

[92] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *ACM Special Interest Group on Management Of Data Conference*, pages 426–435, 1998.

[93] Dragos D. Margineantu and Thomas G. Dietterich. Pruning adaptive boosting. In *International Conference on Machine Learning*, pages 211–218, 1997.

[94] J. Kent Martin and D. S. Hirschberg. On the complexity of learning decision trees. In *International Symposium on Artificial Intelligence and Mathematics*, pages 112–115, 1996.

[95] Ross A. McDonald, David J. Hand, and Idris A. Eckley. An empirical comparison of three boosting algorithms on real data sets with artificial class noise. In *International Workshop on Multiple Classifier Systems*, pages 35–44, 2003.

[96] Manish Mehta, Rakesh Agrawal, and Jorma Rissanen. SLIQ: A fast scalable classifier for data mining. In *Extending Database Technology*, pages 18–32, 1996.

[97] Edmond Mesrobian, Richard Muntz, Eddie Shek, Siliva Nittel, Mark La Rouche, Marc Kriguer, Carlos Mechoso, John Farrara, Paul Stolorz, and Hisashi Nakamura. Mining geophysical data for knowledge. *IEEE Expert: Intelligent Systems and Their Applications*, 11(5):34–44, 1996.

[98] J. Ian Munro and Mike Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12:315–323, 1980.

[99] Sreerama K. Murthy and Steven Salzberg. Lookahead and pathology in decision tree induction. In *International Joint Conference on Artificial Intelligence*, pages 1025–1033, 1995.

[100] Michael K. Ng, Zhexue Huang, and Markus Hegland. Data-mining massive time series astronomical data sets - a case study. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 401–402, 1998.

[101] Nikunj C. Oza and Stuart Russell. Experimental comparisons of online and batch versions of bagging and boosting. In *International Conference on Knowledge Discovery and Data Mining*, pages 359–364, 2001.

[102] Nikunj C. Oza and Stuart Russell. Online bagging and boosting. In *Artificial Intelligence and Statistics*, pages 105–112, 2001.

[103] J. Ross Quinlan. Miniboosting decision trees, submitted for publication 1998, available at http://citeseer.ist.psu.edu/quinlan99miniboosting.html.

[104] J. Ross Quinlan. *C4.5: programs for machine learning.* Morgan Kaufmann, San Francisco, 1993.

[105] J. Ross Quinlan. Improved use of continuous attributes in C4.5. *Journal of Artificial Intelligence Research*, 4:77–90, 1996.

[106] Jorma Rissanen. A universal prior for integers and estimation by minimum description length. *The Annals of Statistics*, 11(2):416–431, 1983.

[107] Robert E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2):197–227, 1990.

[108] Robert E. Schapire, Yoav Freund, Peter Bartlett, and Wee Sun Lee. Boosting the margin: A new explanation for the effectiveness of voting methods. In *International Conference on Machine Learning*, pages 322–330, 1997.

[109] Robert E. Schapire and Yoram Singer. Improved boosting using confidence-rated predictions. *Machine Learning*, 37(3):297–336, 1999.

[110] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *ACM Special Interest Group on Management Of Data Conference*, pages 23–34, 1979.

[111] John Shafer, Rakesh Agrawal, and Manish Mehta. SPRINT: A scalable parallel classifier for data mining. In *International Conference on Very Large Databases*, pages 544–555, 1996.

[112] Myra Spiliopoulou. The laborious way from data mining to web log mining. *International Journal of Computer Systems Science and Engineering*, 14(2):113–125, 1999.

[113] Jaideep Srivastava, Robert Cooley, Mukund Deshpande, and Pang-Ning Tan. Web usage mining: Discovery and applications of usage patterns from web data. *SIGKDD Explorations*, 1(2):12–23, 2000.

[114] W. Nick Street and YongSeog Kim. A streaming ensemble algorithm (SEA) for large-scale classification. In *International Conference on Knowledge Discovery and Data Mining*, pages 377–382, 2001.

[115] Nadeem Ahmed Syed, Huan Liu, and Kah Kay Sung. Handling concept drifts in incremental learning with support vector machines. In *International Conference on Knowledge Discovery and Data Mining*, pages 317–321, 1999.

[116] Kagan Tumer and Joydeep Ghosh. Error correlation and error reduction in ensemble classifiers. *Connection Science*, 8(3/4):385–403, 1996.

[117] Paul E. Utgoff. Incremental induction of decision trees. *Machine Learning*, 4(2):161–186, 1989.

[118] Paul E. Utgoff, Neil C. Berkman, and Jeffery A. Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29(1):5–44, 1997.

[119] Leslie G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[120] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.

[121] Haixun Wang, Wei Fan, Philip S. Yu, and Jiawei Han. Mining concept-drifting data streams using ensemble classifiers. In *International Conference on Knowledge Discovery and Data Mining*, pages 226–235, 2003.

[122] Gary M. Weiss. Data mining in telecommunications. In Oded Maimon and Lior Rokach, editors, *The Data Mining and Knowledge Discovery Handbook*, pages 1189–1201. Springer, 2005.

[123] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.

[124] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, 2nd edition, 2005.