DATA STREAM MINING
A Practical Approach

Albert Bifet, Geoff Holmes, Richard Kirkby and Bernhard Pfahringer

May 2011

COSI Centre for Open Software Innovation

Contents

Introduction

Massive **O**nline **A**nalysis (MOA) is a software environment for implementing algorithms and running experiments for online learning from evolving data streams.

MOA includes a collection of offline and online methods as well as tools for evaluation. In particular, it implements boosting, bagging, and Hoeffding Trees, all with and without Naïve Bayes classifiers at the leaves.

MOA is related to WEKA, the Waikato Environment for Knowledge Analysis, which is an award-winning 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.

This text explains the theoretical and practical foundations of the methods and streams available in MOA. 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.

Part I

Introduction and Preliminaries

1 Preliminaries

In today's information society, extraction of knowledge is becoming a very important task for many people. We live in an age of knowledge revolution. Peter Drucker [55], an influential management expert, writes "From now on, the key is knowledge. The world is not becoming labor intensive, not material intensive, not energy intensive, but knowledge intensive". This knowledge revolution is based in an economic change from adding value by producing things which is, ultimately limited, to adding value by creating and using knowledge which can grow indefinitely.

The digital universe in 2007 was estimated in [80] to be 281 exabytes or 281 billion gigabytes, and by 2011, the digital universe will be 10 times the size it was 5 years before. The amount of information created, or captured exceeded available storage for the first time in 2007.

To deal with these huge amount of data in a responsible way, green computing is becoming a necessity. *Green computing* is the study and practice of using computing resources efficiently. A main approach to green computing is based on algorithmic efficiency. The amount of computer resources required for any given computing function depends on the efficiency of the algorithms used. As the cost of hardware has declined relative to the cost of energy, the energy efficiency and environmental impact of computing systems and programs are receiving increased attention.

MOA Stream Mining

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.

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.

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. **MOA** 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 in **MOA** 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. **MOA** is developed to provide useful insight about classification performance.

Assumptions

MOA 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 or evolving. Concept drift occurs when the underlying concept defining the target being learned begins to shift over time.

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.

Requirements

The conventional machine learning setting, referred to in this text 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. 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).

Mining Strategies

The task of modifying machine learning algorithms to handle large data sets is known as *scaling up* [46]. Analogous to approaches used in data mining, there are two general strategies for taking machine learning concepts and applying

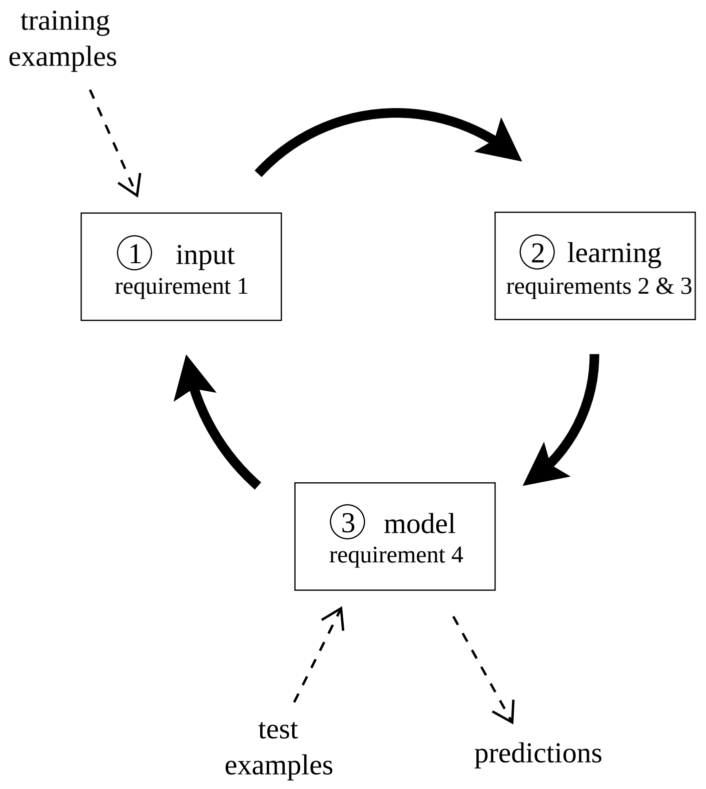


Figure 1.1: The data stream classification cycle.

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. [188], Street and Kim [180] and Chu and Zaniolo [38].

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 text . Chapter 3 studies a successful adaptation of decision trees to data streams [52] 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. [60, 61] 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 [133, 70, 121, 15].

support vector machines/neural networks Both of these methods are related and of similar power, although *support vector machines* [33] 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 [181]. Neural networks are relatively straightforward to train on a data stream. A real world application using neural networks is given by Gama and Rodrigues [78]. 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 [107] describe a method of learning Bayesian networks from data streams using Hoeffding bounds. Bouckaert [22] 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 [154, 133]. This is the other major class of algorithm studied in-depth by this text , beginning in Chapter 6.

Gaber et al. [71] 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* [52], the decision tree algorithm studied in-depth by this text ; *ensemble-based classification* by Wang et al. [188] that has been mentioned as a wrapper approach; *SCALLOP*, a rule-based learner that is the earlier work of Ferrer-Troyano et al. [60]; *ANNCAD*, which is a nearest neighbour method developed by Law and Zaniolo [133] that operates using *Haar wavelet* transformation of data and small classifier ensembles; and *LWClass* proposed by Gaber et al. [70], 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. [3] 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 [132]. 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.

Dealing with time-changing data requires strategies for detecting and quantifying change, forgetting stale examples, and for model revision. Fairly generic strategies exist for detecting change and deciding when examples are no longer relevant.

Change Detection Strategies

The following different modes of change have been identified in the literature [182, 179, 192]:

- concept change

  - concept drift
  - concept shift

- distribution or sampling change

Concept refers to the target variable, which the model is trying to predict. *Concept change* is the change of the underlying concept over time. Concept drift describes a gradual change of the concept and concept shift happens when a change between two concepts is more abrupt.

Distribution change, also known as sampling change or shift or virtual concept drift , refers to the change in the data distribution. Even if the concept remains the same, the change may often lead to revising the current model as the model's error rate may no longer be acceptable with the new data distribution.

Some authors, as Stanley [179], have suggested that from the practical point of view, it is not essential to differentiate between concept change and sampling change since the current model needs to be changed in both cases.

Change detection is not an easy task, since a fundamental limitation exists [90]: the design of a change detector is a compromise between detecting true changes and avoiding false alarms. See [90, 14] for more detailed surveys of change detection methods.

The CUSUM Test

The cumulative sum (CUSUM algorithm), first proposed in [155], is a change detection algorithm that gives an alarm when the mean of the input data is significantly different from zero. The CUSUM input ϵ_t can be any filter residual, for instance the prediction error from a Kalman filter.

The CUSUM test is as follows:

$$g_0 = 0$$

$$g_t = \max\,(0, g_{t-1} + \epsilon_t - \upsilon)$$
$$\text{if } g_t > h \text{ then alarm and } g_t = 0$$

The CUSUM test is memoryless, and its accuracy depends on the choice of parameters υ and h.

The Geometric Moving Average Test

The CUSUM test is a stopping rule. Other stopping rules exist. For example, the Geometric Moving Average (GMA) test, first proposed in [164], is the following

$$g_0 = 0$$

$$g_t = \lambda g_{t-1} + (1-\lambda)\epsilon_t$$
$$\text{if } g_t > h \text{ then alarm and } g_t = 0$$

The forgetting factor λ is used to give more or less weight to the last data arrived. The treshold h is used to tune the sensitivity and false alarm rate of the detector.

Statistical Tests

CUSUM and GMA are methods for dealing with numeric sequences. For more complex populations, we need to use other methods. There exist some statistical tests that may be used to detect change. A statistical test is a procedure for deciding whether a hypothesis about a quantitative feature of a population is true or false. We test an hypothesis of this sort by drawing a random sample from the population in question and calculating an appropriate statistic on its items. If, in doing so, we obtain a value of the statistic that would occur rarely when the hypothesis is true, we would have reason to reject the hypothesis.

To detect change, we need to compare two sources of data, and decide if the hypothesis H_0 that they come from the same distribution is true. Let's suppose we have two estimates, $\hat{\mu}_0$ and $\hat{\mu}_1$ with variances σ_0^2 and σ_1^2. If there is no change in the data, these estimates will be consistent. Otherwise, a hypothesis test will reject H_0 and a change is detected. There are several ways to construct such a hypothesis test. The simplest one is to study the difference

$$\hat{\mu}_0 - \hat{\mu}_1 \in N(0, \sigma_0^2 + \sigma_1^2), \text{ under } H_0$$

or, to make a χ^2 test

$$\frac{(\hat{\mu}_0 - \hat{\mu}_1)^2}{\sigma_0^2 + \sigma_1^2} \in \chi^2(1), \text{ under } H_0$$

from which a standard hypothesis test can be formulated.

For example, suppose we want to design a change detector using a statistical test with a probability of false alarm of 5%, that is,

$$\Pr\left(\frac{|\hat{\mu}_0 - \hat{\mu}_1|}{\sqrt{\sigma_0^2 + \sigma_1^2}} > h \right) = 0.05$$

A table of the Gaussian distribution shows that $P(X < 1.96) = 0.975$, so the test becomes

$$\frac{(\hat{\mu}_0 - \hat{\mu}_1)^2}{\sigma_0^2 + \sigma_1^2} > 1.96$$

Note that this test uses the normality hypothesis. In Chapter 8 we will propose a similar test with theoretical guarantees. However, we could have used this test on the methods of Chapter 8.

The Kolmogorov-Smirnov test [117] is another statistical test used to compare two populations. Given samples from two populations, the cumulative distribution functions can be determined and plotted. Hence the maximum value of the difference between the plots can be found and compared with a critical value. If the observed value exceeds the critical value, H_0 is rejected and a change is detected. It is not obvious how to implement the Kolmogorov-Smirnov test dealing with data streams. Kifer et al. [122] propose a KS-structure to implement Kolmogorov-Smirnov and similar tests, on the data stream setting.

Drift Detection Method

The drift detection method (DDM) proposed by Gama et al. [72] controls the number of errors produced by the learning model during prediction. It compares the statistics of two windows: the first one contains all the data, and the second one contains only the data from the beginning until the number of errors increases. This method does not store these windows in memory. It keeps only statistics and a window of recent data.

The number of errors in a sample of n examples is modelized by a binomial distribution. For each point i in the sequence that is being sampled, the error rate is the probability of misclassifying (p_i), with standard deviation given by $s_i = \sqrt{p_i(1-p_i)/i}$. It assumes (as can be argued e.g. in the PAC learning model [146]) that the error rate of the learning algorithm (p_i) will decrease while the number of examples increases if the distribution of the examples is stationary. A significant increase in the error of the algorithm, suggests that the class distribution is changing and, hence, the actual decision model is supposed to be inappropriate. Thus, it stores the values of p_i and s_i when $p_i + s_i$ reaches its minimum value during the process (obtaining p_{pmin} and s_{min}), and when the following conditions triggers:

- $p_i + s_i \geq p_{min} + 2 \cdot s_{min}$ for the warning level. Beyond this level, the examples are stored in anticipation of a possible change of context.

- $p_i + s_i \geq p_{min} + 3 \cdot s_{min}$ for the drift level. Beyond this level the concept drift is supposed to be true, the model induced by the learning method is reset and a new model is learnt using the examples stored since the warning level triggered. The values for p_{min} and s_{min} are reset too.

This approach has a good behaviour of detecting abrupt changes and gradual changes when the gradual change is not very slow, but it has difficulties when the change is slowly gradual. In that case, the examples will be stored for long time, the drift level can take too much time to trigger and the example memory can be exceeded.

Baena-García et al. proposed a new method EDDM in order to improve DDM. EDDM [12] is shown to be better than DDM for some data sets and worse for others. It is based on the estimated distribution of the distances between classification errors. The window resize procedure is governed by the same heuristics.

Exponential Weighted Moving Average

An Estimator is an algorithm that estimates the desired statistics on the input data, which may change over time. The simplest Estimator algorithm for the expected is the *linear estimator,* which simply returns the average of the data items contained in the Memory. Other examples of run-time efficient estimators are Auto-Regressive, Auto Regressive Moving Average, and Kalman filters.

An exponentially weighted moving average (EWMA) estimator is an algorithm that updates the estimation of a variable by combining the most recent measurement of the variable with the EWMA of all previous measurements:

$$X_t = \alpha z_t + (1-\alpha)X_{t-1} = X_{t-1} + \alpha(z_t - X_{t-1})$$

where X_t is the moving average, z_t is the latest measurement, and α is the weight given to the latest measurement (between 0 and 1). The idea is to produce an estimate that gives more weight to recent measurements, on the assumption that recent measurements are more likely to be relevant. Choosing an adequate α is a difficult problem, and it is not trivial.

2 MOA Experimental Setting

This chapter establishes the settings under which stream mining experiments may be conducted, presenting the framework **MOA** to place various learning algorithms under test. This experimental methodology is motivated by the requirements of the end user and their desired application.

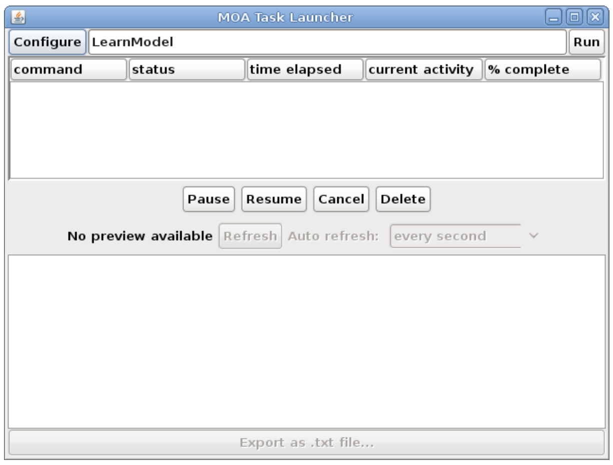


Figure 2.1: Graphical user interface of MOA

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 text. 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 text.

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.

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 [126].

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 [56]. 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 [126] 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 [126]. For increased reliability, ten repetitions of ten-fold cross-validation are commonly used. Bouckaert [21] warns that results based on this standard should still be treated with caution.

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 text. 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 [52], Gama et al. [77], Gama et al. [75], Jin and Agrawal [115], Oza and Russell [153], Street and Kim [180], Fern and Givan [59], and Chu and Zaniolo [38]. 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

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
[52]	holdout	40MB, 80MB	14 custom syn. 1 private real	100m 4m	50k 267k
[77]	holdout	none	3 public syn. (UCI)	1m	250k
[75]	holdout	none	4 public syn. (UCI)	1.5m	250k
[115]	holdout?	60MB	3 public syn. (genF1/6/7)	10m	?
[153]	5-fold cv, holdout	none	10 public real (UCI) 3 custom syn. 2 public real (UCIKDD)	54k 80k 465k	13.5k 20k 116k
[180]	5-fold cv, holdout	none	2 public real (UCI) 1 private real 1 custom syn.	45k 33k 50k	(5-fold cv) (5-fold cv) 10k
[59]	various	strict hardware	4 public real (UCI) 8 public real (spec95)	100k 2.6m	
[38]	holdout	none	1 custom syn. 1 private real	400k 100k	50k ?

Table 2.2: Paper survey part 2—Presentation styles.

paper ref.	presentation of results and comparisons
[52]	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
[77]	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
[75]	1 table of error, training time & tree size (after 100k, 500k, 750k/1m & 1m/1.5m examples)
[115]	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
[153]	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
[180]	8 plots of error vs examples
[59]	25 plots of error vs ensemble size 13 plots of error vs examples 6 plots of error vs tree nodes
[38]	3 plots of accuracy vs tree leaves 2 tables of accuracy for several parameters and methods 3 plots of accuracy vs examples

stream setting, although some papers use five-fold cross-validation, and Fern and Givan [59] 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 limits placed on algorithms. Domingos and Hulten [52] makes the most effort to explore limited memory, and the followup work by Jin and Agrawal [115] is consistent by also mentioning a fixed limit. The paper by Fern and Givan [59] 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 [9], although Jin and Agrawal [115] make use of the generator described in Section 2.5.5, and Fern and Givan [59] 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 [52] 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 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 [123] 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.

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 [52], represents a significant inspiration for this text 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.

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.

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 predefined, 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.

When 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.

Interleaved Test-Then-Train or Prequential

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

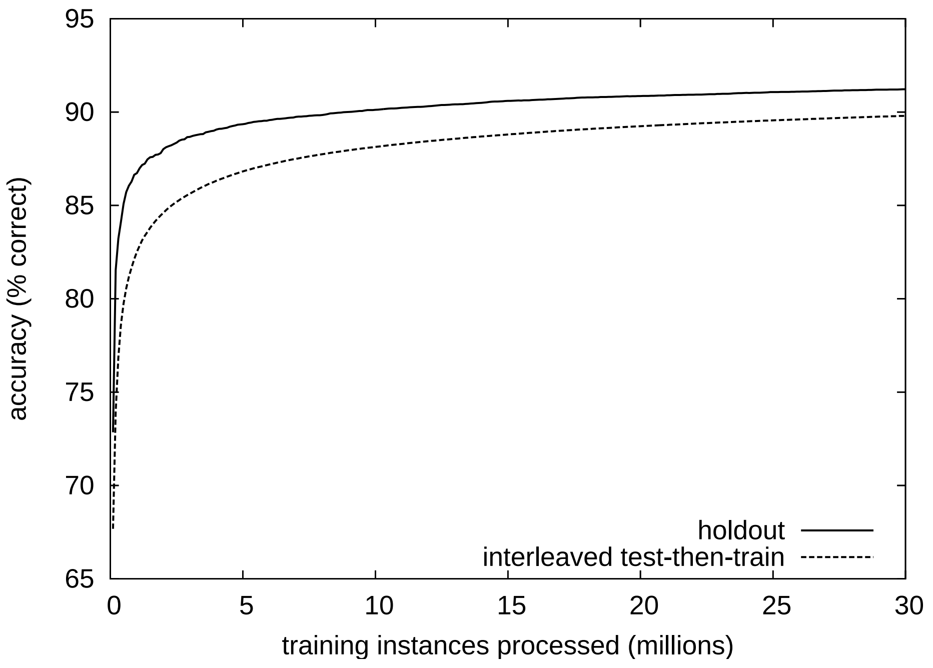


Figure 2.2: Learning curves produced for the same learning situation by two different evaluation methods, recorded every 100,000 examples.

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 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.

Comparison

Figure 2.2 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.

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 text. The process is similar to that used by Domingos and Hulten [52], 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 is

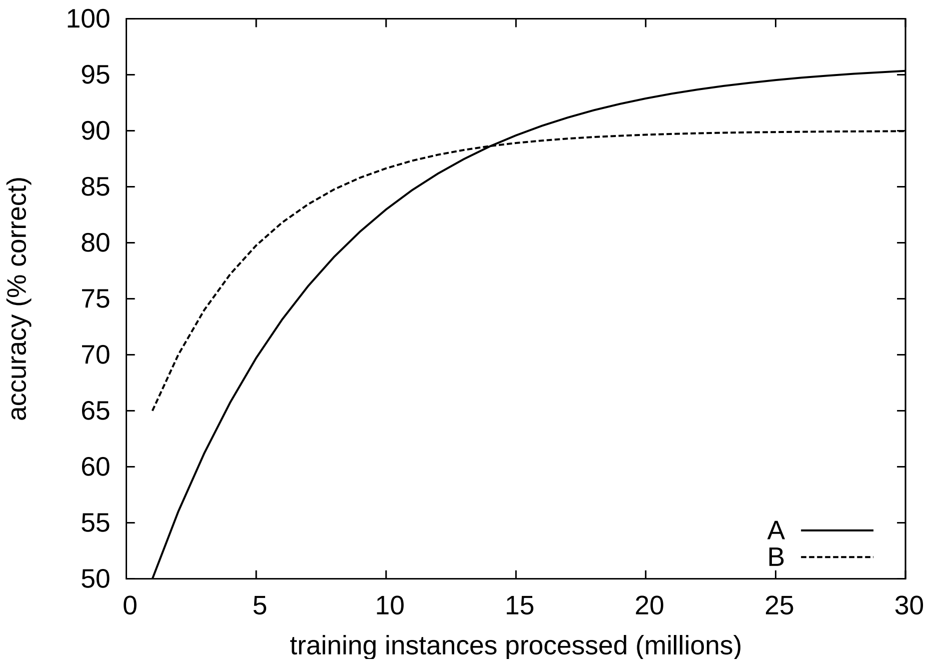


Figure 2.3: Learning curves demonstrating the problem of stopping early.

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 2.3. 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.

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 [45] 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 text, but the idea presents opportunity for future research into evaluation of data stream classification.

Environments

This section defines three environments that may be 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.

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 [6, 74] 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 [59]. Another example is a small 'packet sniffer' device designed to do real-time monitoring of network traffic [8].

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 [118] and [119] 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. [104] 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 [110] present a vision of this future 'dataspace'.

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 text 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 [92], web usage [178], site logs [177] or click streams [89]. Smaller scale computer networks also produce traffic of interest [135], as do other telecommunication activities [189], phone call logs for example. Banks may be interested in patterns of ATM transactions [93], and retail chains and online stores will want details about customer purchases [129]. Further still, there is the field of scientific observation [86], which can be astronomical [151], geophysical [144], or the massive volume of data output by particle accelerator experiments [105]. All of these activities are sources of data streams that users will conceivably want to analyze in a server environment.

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 [9] and KDD [99] 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 web trace from the University of Washington used by Domingos and Hulten to evaluate VFDT [52], and the credit card fraud data used by Wang et al. [188] and Chu and Zaniolo [38].

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 [153], Street and Kim [180] and Chu and Zaniolo [38]. 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 text, 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.

Random Tree Generator

This generator is based on that proposed by Domingos and Hulten [52], 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 as-

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

signing 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.

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.

LED Generator

This data source originates from the CART book [29]. An implementation in C was donated to the UCI [9] 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.

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.

Function Generator

This generator was introduced by Agrawal et al. in [4], and was a common source of data for early work on scaling up decision tree learners [143, 175, 81].

The generator produces a stream containing nine attributes, six numeric and three categorical, described in Table 2.4. 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 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.

```
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
       $\mathrm{group} = A$
   else
       $\mathrm{group} = B$
```

Figure 2.4: Generator functions 1-5.

```
6. if ((age < 40) ∧ (50000 ≤ (salary + commission) ≤ 100000)) ∨
       ((40 ≤ age < 60) ∧ (75000 ≤ (salary + commission) ≤ 125000)) ∨
       ((age ≥ 60) ∧ (25000 ≤ (salary + commission) ≤ 75000)) then
       group = A
   else
       group = B

7. if (0.67 × (salary + commission) − 0.2 × loan − 20000 > 0) then
       group = A
   else
       group = B

8. if (0.67 × (salary + commission) − 5000 × elevel − 20000 > 0) then
       group = A
   else
       group = B

9. if (0.67 × (salary + commission) − 5000 × elevel
       − 0.2 × loan − 10000 > 0) then
       group = A
   else
       group = B

10. if (hyears ≥ 20) then
        equity = 0.1 × hvalue × (hyears − 20)
    else
        equity = 0

    if (0.67 × (salary + commission) − 5000 × elevel
        − 0.2 × equity − 10000 > 0) then
        group = A
    else
        group = B
```

Figure 2.5: Generator functions 6-10.

Table 2.4: 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

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 MOA data generators was measured in the experimental hardware/software environment. The results are shown in Table 2.5, 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 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.

For example, considering an 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.

Table 2.5: 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

Evolving Stream Experimental Setting

This section proposes a new experimental data stream framework for studying concept drift using MOA. A majority of concept drift research in data streams mining is done using traditional data mining frameworks such as WEKA [91]. As the data stream setting has constraints that a traditional data mining environment does not, we believe that MOA will help to improve the empirical evaluation of these methods.

In data stream mining, we are interested in three main dimensions:

- accuracy
- amount of space necessary or computer memory
- the time required to learn from training examples and to predict

These properties may be interdependent: adjusting the time and space used by an algorithm can influence accuracy. 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, the more likely it is that accuracy can be increased.

In evolving data streams we are concerned about

- evolution of accuracy
- probability of false alarms
- probability of true detections
- average delay time in detection

Sometimes, learning methods do not have change detectors implemented inside, and then it may be hard to define ratios of false positives and negatives, and average delay time in detection. In these cases, learning curves may be a useful alternative for observing the evolution of accuracy in changing environments.

To summarize, the main properties of an ideal learning method for mining evolving data streams are the following: high accuracy and fast adaption to change, low computational cost in both space and time, theoretical performance guarantees, and minimal number of parameters.

Concept Drift Framework

We present the new experimental framework for concept drift in MOA. Our goal was to introduce artificial drift to data stream generators in a straightforward way.

The framework approach most similar to the one presented in this Chapter is the one proposed by Narasimhamurthy et al. [150]. They proposed a general

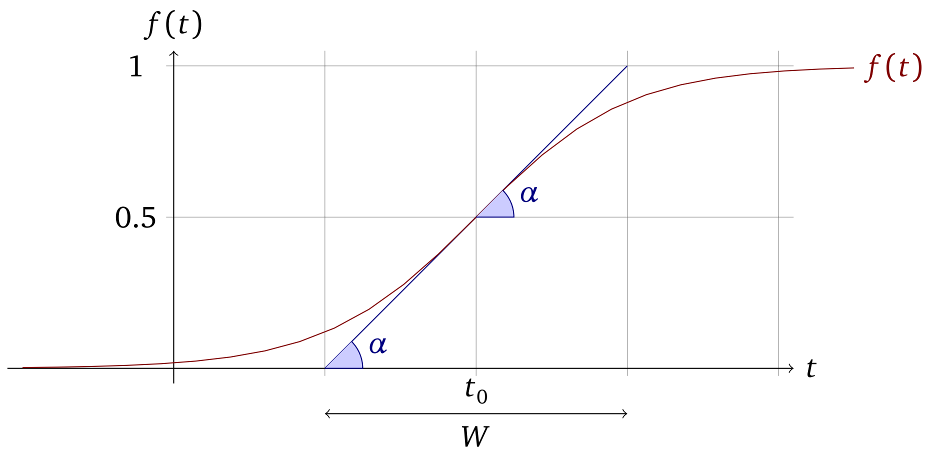


Figure 2.6: A sigmoid function $f(t) = 1/(1+e^{-s(t-t_0)})$.

framework to generate data simulating changing environments. Their framework accommodates the STAGGER and Moving Hyperplane generation strategies. They consider a set of k data sources with known distributions. As these distributions at the sources are fixed, the data distribution at time t, $D^{(t)}$ is specified through $v_i(t)$, where $v_i(t) \in [0,1]$ specify the extent of the influence of data source i at time t:

$$D^{(t)} = \{v_1(t), v_2(t), \ldots, v_k(t)\}, \sum_i v_i(t) = 1$$

Their framework covers gradual and abrupt changes. Our approach is more concrete, we begin by dealing with a simple scenario: a data stream and two different concepts. Later, we will consider the general case with more than one concept drift events.

Considering data streams as data generated from pure distributions, we can model a concept drift event as a weighted combination of two pure distributions that characterizes the target concepts before and after the drift. In our framework, we need to define the probability that every new instance of the stream belongs to the new concept after the drift. We will use the sigmoid function, as an elegant and practical solution.

We see from Figure 2.6 that the sigmoid function

$$f(t) = 1/(1+e^{-s(t-t_0)})$$

has a derivative at the point t_0 equal to $f'(t_0) = s/4$. The tangent of angle α is equal to this derivative, $\tan\alpha = s/4$. We observe that $\tan\alpha = 1/W$, and as $s = 4\tan\alpha$ then $s = 4/W$. So the parameter s in the sigmoid gives the length of W and the angle α. In this sigmoid model we only need to specify two parameters : t_0 the point of change, and W the length of change. Note that

$$f(t_0 + \beta \cdot W) = 1 - f(t_0 - \beta \cdot W),$$

and that $f(t_0 + \beta \cdot W)$ and $f(t_0 - \beta \cdot W)$ are constant values that don't depend on t_0 and W:

$$f(t_0 + W/2) = 1 - f(t_0 - W/2) = 1/(1+e^{-2}) \approx 88.08\%$$

$$f(t_0 + W) = 1 - f(t_0 - W) = 1/(1 + e^{-4}) \approx 98.20\%$$
$$f(t_0 + 2W) = 1 - f(t_0 - 2W) = 1/(1 + e^{-8}) \approx 99.97\%$$

Definition 1. *Given two data streams a, b, we define $c = a \oplus^W_{t_0} b$ as the data stream built joining the two data streams a and b, where t_0 is the point of change, W is the length of change and*

- $\Pr[c(t) = a(t)] = \mathrm{e}^{-4(t-t_0)/W}/(1 + \mathrm{e}^{-4(t-t_0)/W})$
- $\Pr[c(t) = b(t)] = 1/(1 + \mathrm{e}^{-4(t-t_0)/W})$.

We observe the following properties, if $a \neq b$:

- $a \oplus^W_{t_0} b \neq b \oplus^W_{t_0} a$
- $a \oplus^W_{t_0} a = a$
- $a \oplus^0_0 b = b$
- $a \oplus^W_{t_0} (b \oplus^W_{t_0} c) \neq (a \oplus^W_{t_0} b) \oplus^W_{t_0} c$
- $a \oplus^W_{t_0} (b \oplus^W_{t_1} c) \approx (a \oplus^W_{t_0} b) \oplus^W_{t_1} c$ if $t_0 < t_1$ and $W \ll |t_1 - t_0|$

In order to create a data stream with multiple concept changes, we can build new data streams joining different concept drifts:

$$(((a \oplus^{W_0}_{t_0} b) \oplus^{W_1}_{t_1} c) \oplus^{W_2}_{t_2} d) \ldots$$

Datasets for concept drift

Synthetic data has several advantages – it is easier to reproduce and there is little cost in terms of storage and transmission. For this framework, the data generators most commonly found in the literature have been collected.

SEA Concepts Generator This dataset contains abrupt concept drift, first introduced in [180]. It is generated using three attributes, where only the two first attributes are relevant. All three attributes have values between 0 and 10. The points of the dataset are divided into 4 blocks with different concepts. In each block, the classification is done using $f_1 + f_2 \leq \theta$, where f_1 and f_2 represent the first two attributes and θ is a threshold value. The most frequent values are 9, 8, 7 and 9.5 for the data blocks. In our framework, SEA concepts are defined as follows:

$$(((SEA_9 \oplus^W_{t_0} SEA_8) \oplus^W_{2t_0} SEA_7) \oplus^W_{3t_0} SEA_{9.5})$$

STAGGER Concepts Generator They were introduced by Schlimmer and Granger in [171]. The concept description in STAGGER is a collection of elements, where each individual element is a Boolean function of attribute-valued pairs that is represented by a disjunct of conjuncts. A typical example of a concept description covering either green rectangles or red triangles can be represented by (shape rectangle and colour green) or (shape triangles and colour red).

Rotating Hyperplane This dataset was used as testbed for CVFDT versus VFDT in [106]. A hyperplane in d-dimensional space is the set of points x that satisfy

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

where x_i, is the ith coordinate of x. Examples for which $\sum_{i=1}^{d} w_i x_i \geq w_0$ are labeled positive, and examples for which $\sum_{i=1}^{d} w_i x_i < w_0$ are labeled negative. Hyperplanes are useful for simulating time-changing concepts, because we can change the orientation and position of the hyperplane in a smooth manner by changing the relative size of the weights. We introduce change to this dataset adding drift to each weight attribute $w_i = w_i + d\sigma$, where σ is the probability that the direction of change is reversed and d is the change applied to every example.

Random RBF Generator This generator was devised to offer an alternate complex concept type that is not straightforward to approximate 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. Drift is introduced by moving the centroids with constant speed. This speed is initialized by a drift parameter.

Data streams may be considered infinite sequences of (x, y) where x is the feature vector and y the class label. Zhang et al. [195] observe that $p(x, y) = p(x|t) \cdot p(y|x)$ and categorize concept drift in two types:

- *Loose Concept Drifting (LCD)* when concept drift is caused only by the change of the class prior probability $p(y|x)$,

- *Rigorous Concept Drifting (RCD)* when concept drift is caused by the change of the class prior probability $p(y|x)$ and the conditional probability $p(x|t)$

Note that the Random RBF Generator has RCD drift, and the rest of the dataset generators have LCD drift.

Real-World Data

It is not easy to find large real-world datasets for public benchmarking, especially with substantial concept change. The UCI machine learning repository [9] contains some real-world benchmark data for evaluating machine learning techniques. We will consider three : Forest Covertype, Poker-Hand, and Electricity.

Forest Covertype dataset It contains the forest cover type for 30 x 30 meter cells obtained from US Forest Service (USFS) Region 2 Resource Information System (RIS) data. It contains 581, 012 instances and 54 attributes, and it has been used in several papers on data stream classification [77, 153].

Poker-Hand dataset It consists of 1, 000, 000 instances and 11 attributes. Each record of the Poker-Hand dataset is an example of a hand consisting of five playing cards drawn from a standard deck of 52. Each card is described using two attributes (suit and rank), for a total of 10 predictive attributes. There is one Class attribute that describes the “Poker Hand”. The order of cards is important, which is why there are 480 possible Royal Flush hands instead of 4.

Electricity dataset Another widely used dataset is the Electricity Market Dataset described by M. Harries [95] and used by Gama [72]. This data was collected from the Australian New South Wales Electricity Market. In this market, the prices are not fixed and are affected by demand and supply of the market. The prices in this market are set every five minutes. The ELEC2 dataset contains 45, 312 instances. Each example of the dataset refers to a period of 30 minutes, i.e. there are 48 instances for each time period of one day. The class label identifies the change of the price related to a moving average of the last 24 hours. The class level only reflect deviations of the price on a one day average and removes the impact of longer term price trends.

The size of these datasets is small, compared to tens of millions of training examples of synthetic datasets: 45, 312 for ELEC2 dataset, 581, 012 for CoverType, and 1, 000, 000 for Poker-Hand. Another important fact is that we do not know when drift occurs or if there is any drift. We may simulate RCD concept drift, joining the three datasets, merging attributes, and supposing that each dataset corresponds to a different concept.

$$\text{CovPokElec} = (\text{CoverType} \oplus^{5{,}000}_{581{,}012} \text{Poker}) \oplus^{5{,}000}_{1{,}000{,}000} \text{ELEC2}$$

Part II

Stationary Data Stream Learning

3 Hoeffding Trees

Hoeffding trees were introduced by Domingos and Hulten in the paper "Mining High-Speed Data Streams" [52]. 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 text 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 [107], 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* [143], *SPRINT* [175] and *RAINFOREST* [81]. 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* [184] and its successor *ITI* [185], 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* [160] and *CART* [29], 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 [91] 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 [141]. 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 [34] 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.

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 text 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 [102], 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 [115]. The Hoeffding bound formulation is well founded and works well empirically, so tighter bounds are not explored in this text.

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.

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.

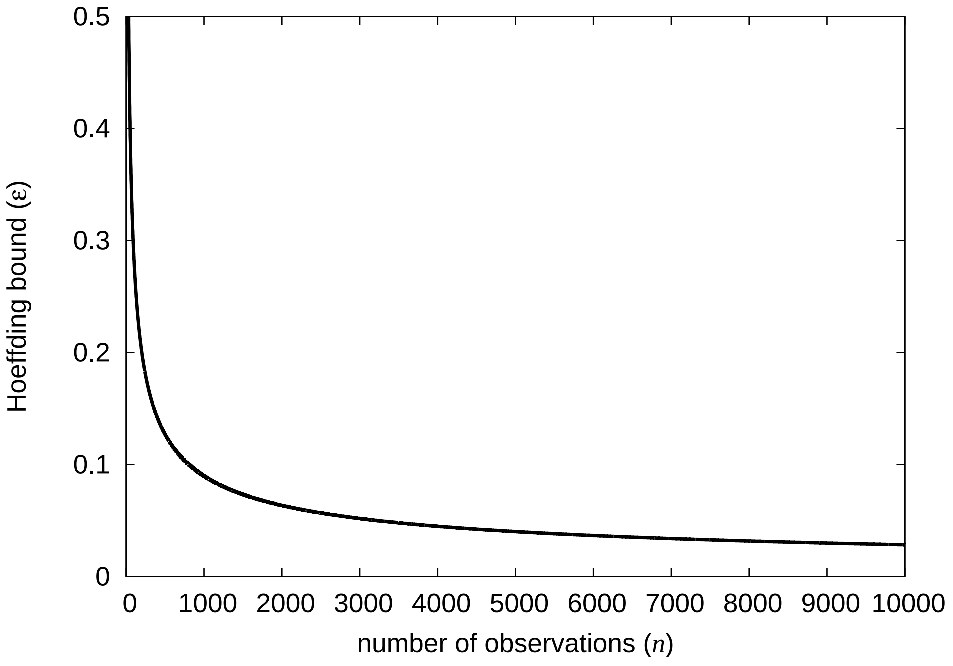


Figure 3.1: Hoeffding bound on a two-class problem with default parameters.

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.

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 text, 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.

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 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 78, 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.

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.

For experimentation n_{min} is fixed at 200, the default setting found in the original paper [52]. 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

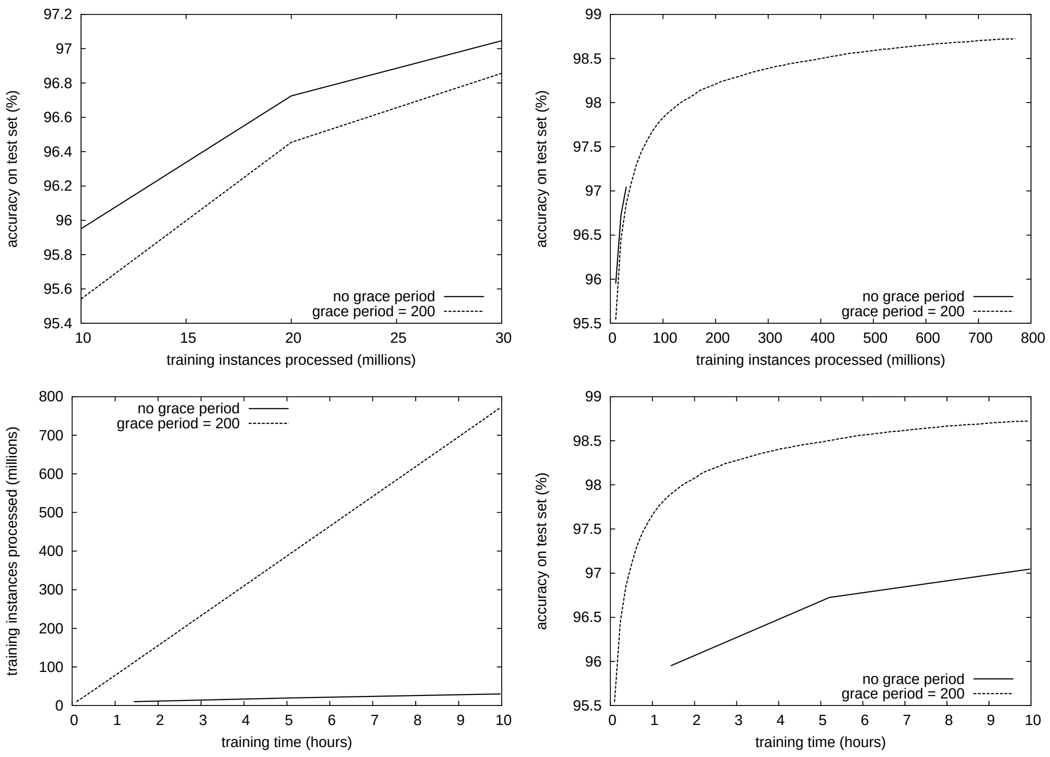


Figure 3.2: Effect of grace period on the RRBFC data with a 32MB memory limit.

the same time period. Viewing accuracy per time spent in the bottom-right plot makes the advantage of using a grace period clear.

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.

All of the Hoeffding tree implementations used in experiments for this text had pre-pruning enabled, but revisiting some of the results without pre-pruning yielded 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 have an effect. This question remains open and is not further explored by this text.

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. One reason for premature pre-pruning could be lack of sufficient data, which is not a problem in abundant data streams.

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 text.

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 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.

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. [77] and originally formulated for two-way numeric splits. In this text 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

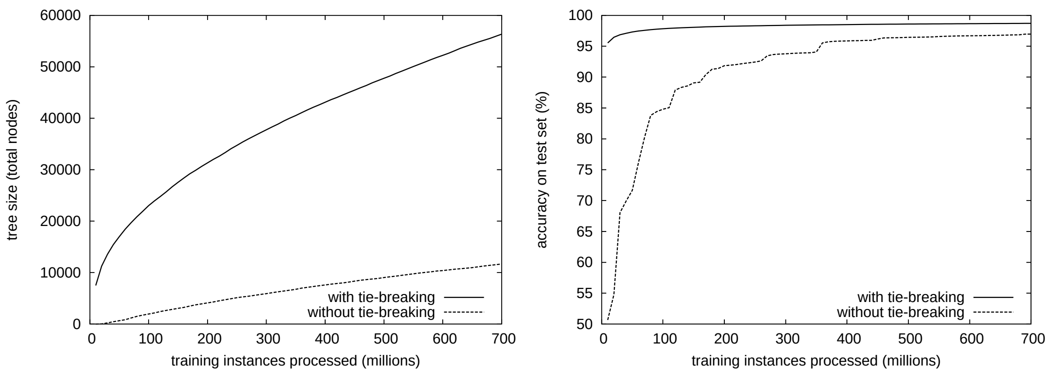


Figure 3.3: Effect of tie-breaking on the RRBFC data with a 32MB memory limit.

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.

Memory Management

The basic algorithm as described will continue to split the tree as needed, without regard for the ever-increasing memory requirements. This text 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

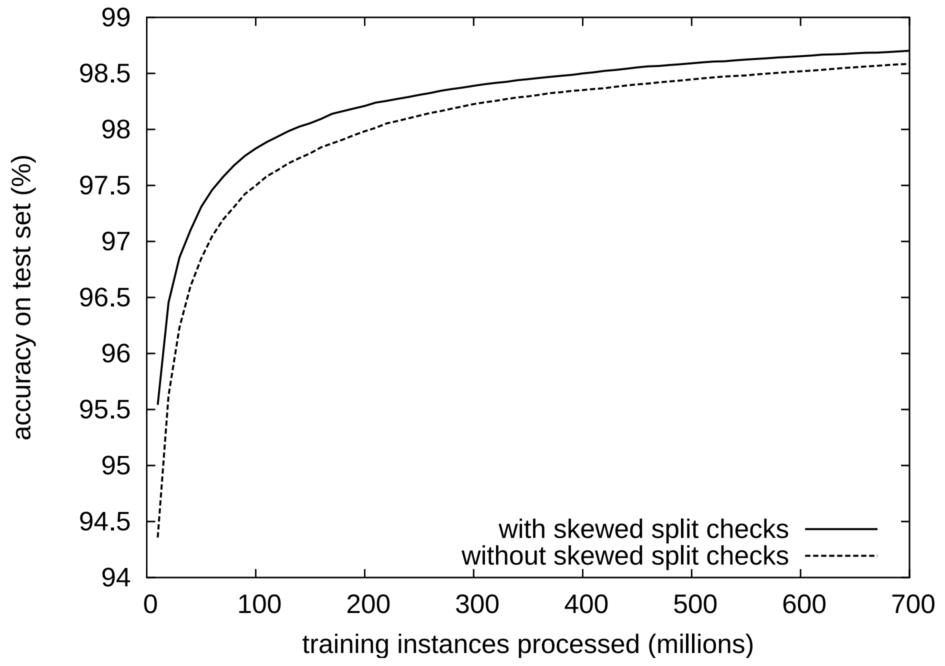


Figure 3.4: Effect of preventing skewed splits on the RRBFC data with a 32MB memory limit.

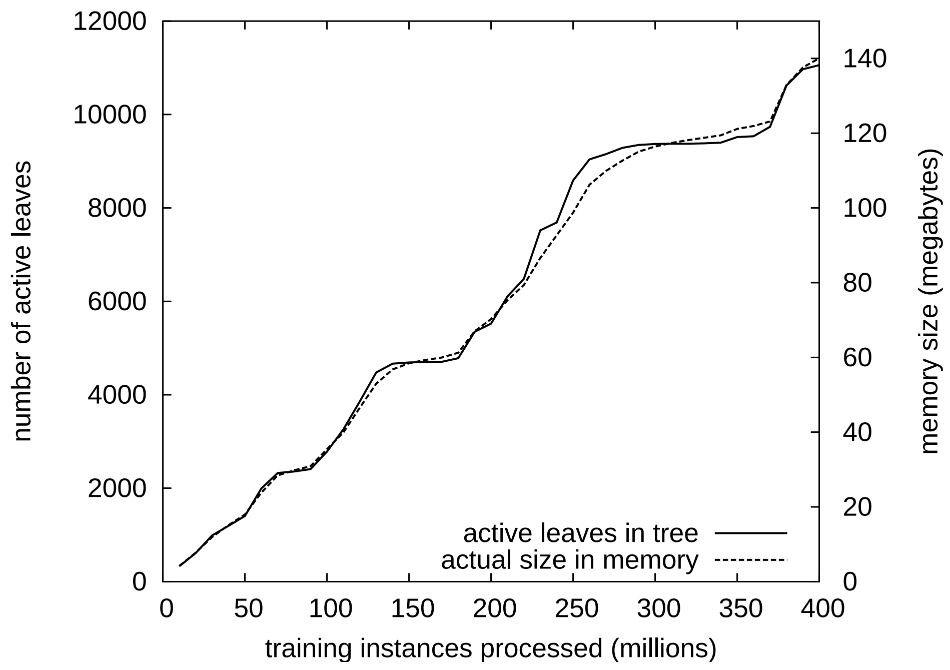


Figure 3.5: The size of an unbounded tree in memory is closely related to how many active leaves it has. This growth pattern occurs when learning a Hoeffding tree on the LED data.

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 text 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 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

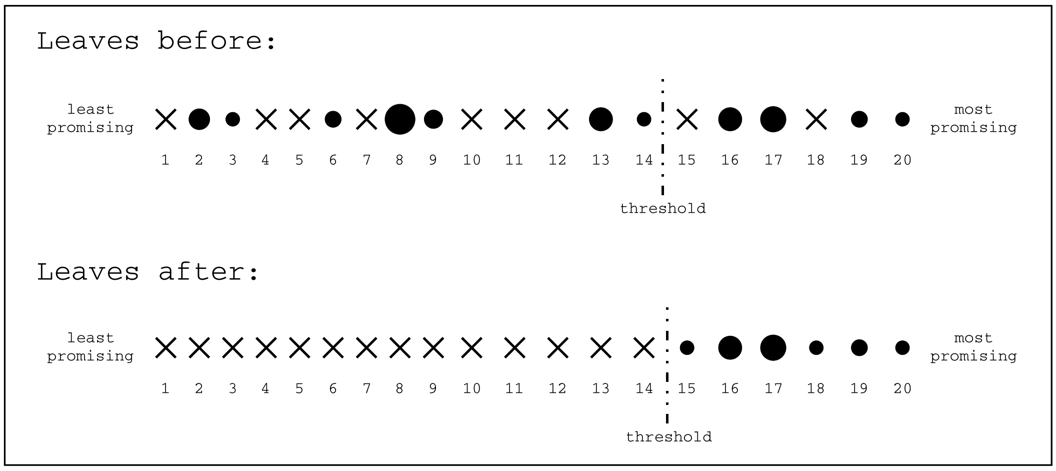


Figure 3.6: The memory management strategy employed after leaves of a tree have been sorted in order of *promise*. Dots represent *active* leaves which store sufficient statistics of various size, crosses represent *inactive* leaves which do not store sufficient statistics.

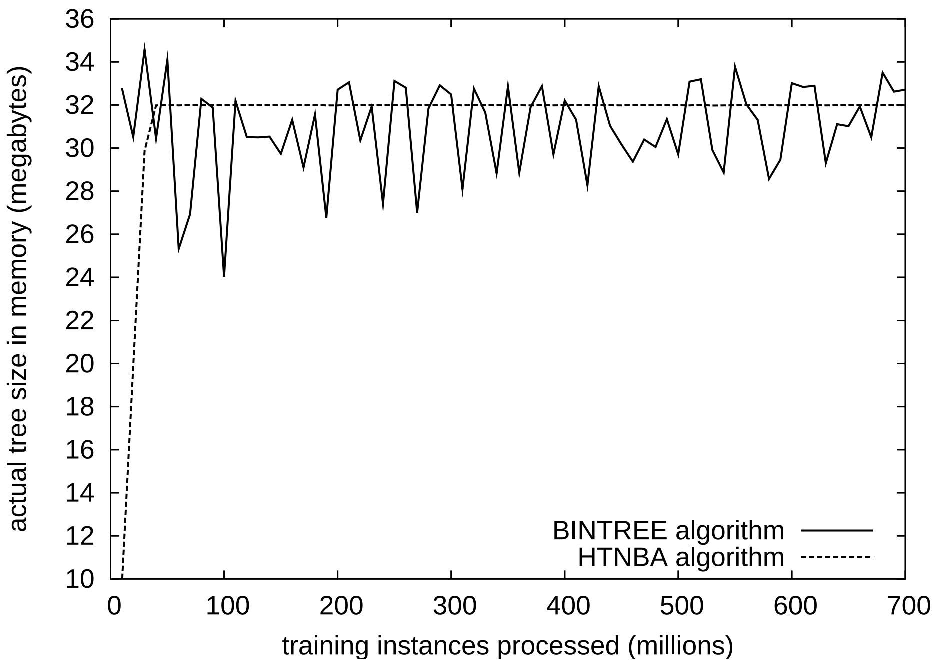


Figure 3.7: How closely two algorithms manage to obey a memory limit of 32 megabytes on the LED data.

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

Poor Attribute Removal

An additional strategy for saving space was also suggested by Domingos and Hulten [52]. 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

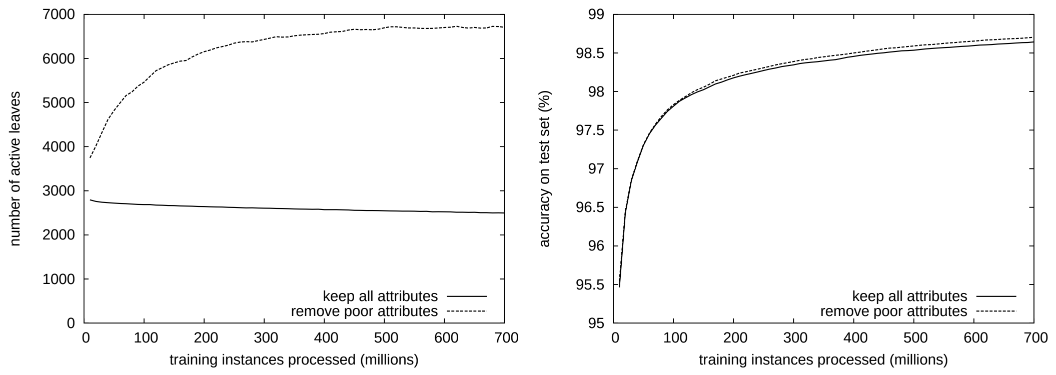


Figure 3.8: Effect of poor attribute removal on RRBFC data with a 32MB limit.

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.

MOA Java Implementation Details

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 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 text. 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 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.

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

```
...
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.

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:

$$overhead = \frac{trueSize}{active \times activeSize + inactive \times inactiveSize} \tag{3.3}$$

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.

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 [52] 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.

Batch Setting Approaches

Strategies for handling continuous attribute values have been extensively studied in the batch setting. Some algorithms, for example *support vector machines* [33], 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 [54].

Methods for discretizing data in the batch setting are surveyed by Dougherty et al. [54]. 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. [54] categorize the *k-means clustering* method as *local*, while Gama and Pinto [76] 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 [76] 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.

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*

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.

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.

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 [96]. 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.

Fayyad and Irani

This method [58] is quite different from those described above as it is *supervised* and *non-parametric*. The algorithm is categorized [54] 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 [58] 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) [163] to estimate when dividing the numeric range ceases to provide any benefit.

C4.5

The procedure for discretizing numeric values in Quinlan's C4.5 [160] 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 [54] 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 [161] by removing some bias in the way that numeric cut-points were chosen.

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 text 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 [76], 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 text, but the problem has been worked on by several researchers such as Guha et al. [88, 87].

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 [76]. 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

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

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.

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

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 [108]. 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.

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 [77]. 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.

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.

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 [174].

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 [187] 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 [147] 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 [112], 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 [5] 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. [7] provides guaranteed error bounds, continued by Manku et al. [138] who demonstrate an improved method with tighter bounds.

The quantile estimation algorithm of Manku et al. [138] was the best known method until Greenwald and Khanna [85] 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 text, 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.

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.

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** $variance = \frac{valueSqSum - mean \times valueSum}{weightSum - 1}$
```

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 [191], and its advantages are studied in [37]. This is the method used in the experimental implementation.

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** $variance = \frac{varianceSum}{weightSum - 1}$
```

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 [75]. 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 example 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

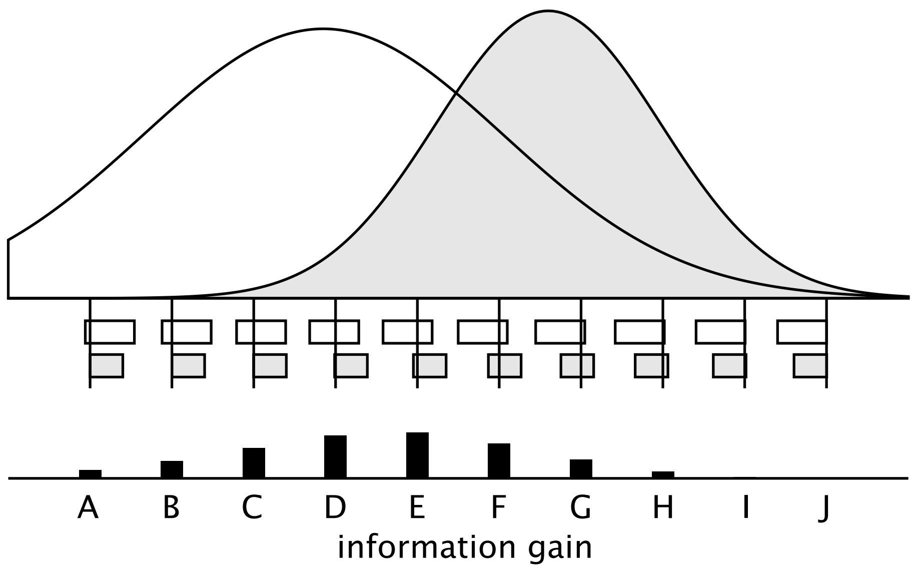


Figure 4.1: Gaussian approximation of 2 classes.

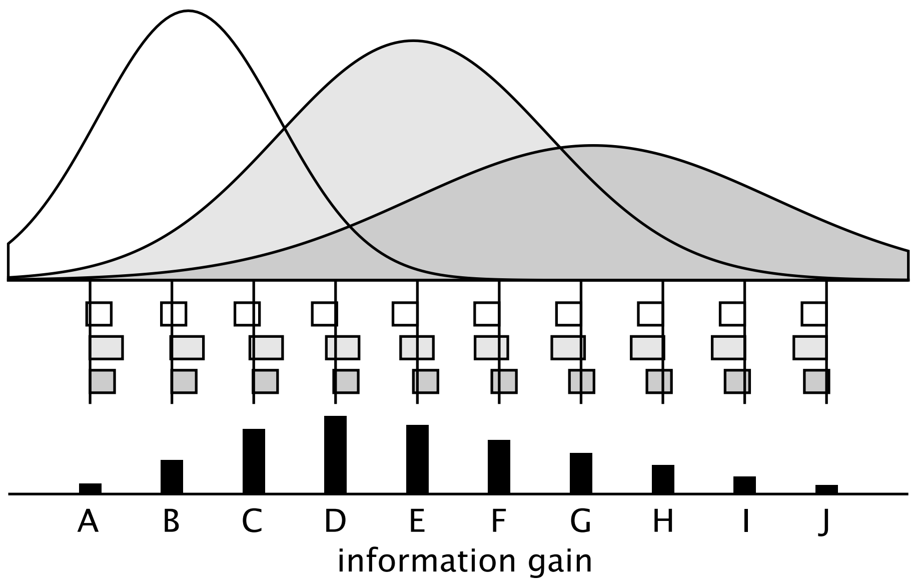


Figure 4.2: Gaussian approximation of 3 classes.

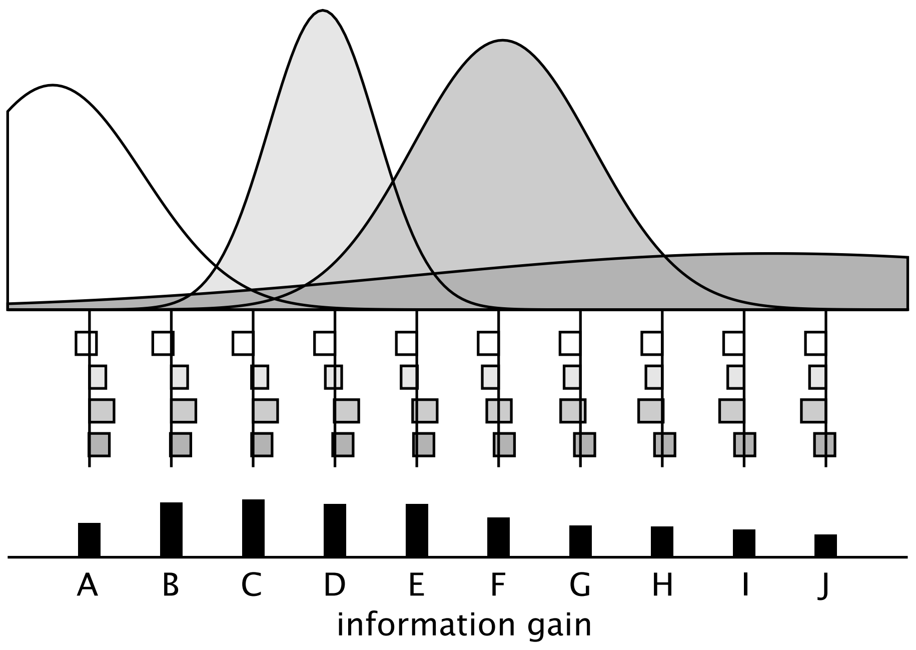


Figure 4.3: Gaussian approximation of 4 classes.

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 34, 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.

Numerical Interval Pruning

Another contribution is the work of Jin and Agrawal [115] 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 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.

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.

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

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* [75, 77].

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

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 [131]. 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 [53, 100].

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* [127]. 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 [75, 77], 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 [103] has covered several attempts at option (1) of 'priming' new leaves with better information. One such attempt, suggested by Gama et al. [75] 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.

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.

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 [94, 183]. 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 [94] 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 [47]. 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 text, 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 [34].

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 [23] 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 dif-

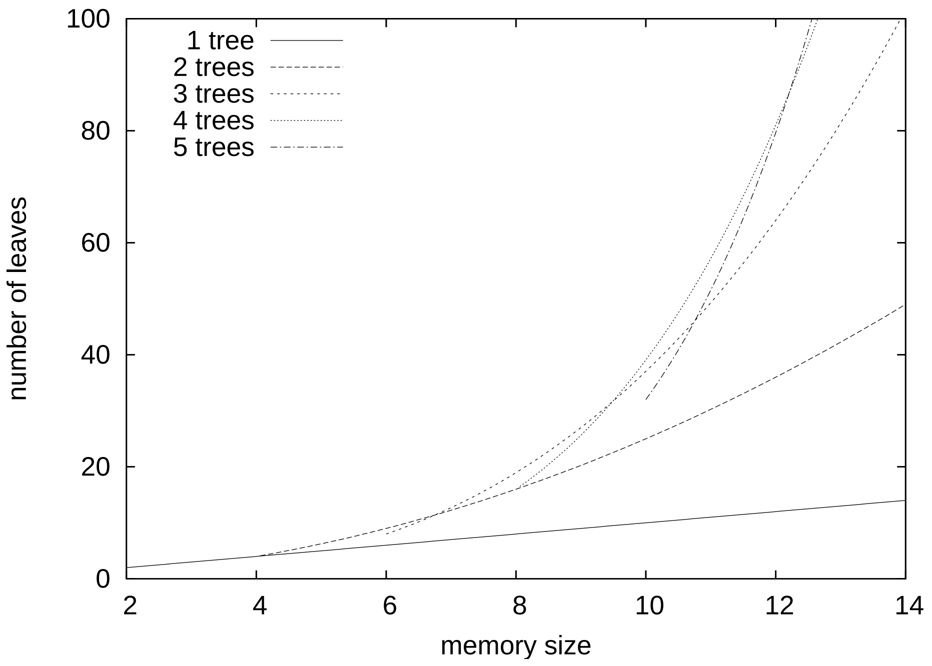


Figure 6.1: A simple model of the leaf count of combinations of decision trees as a function of total memory size.

ferences 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 [52] prove that overall the algorithm approximates the batch induction of decision trees. The empirical results of the tree ensembles 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 [109], 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 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 [159] 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.

A useful way to analyze ensemble behaviour is to consider the implications of *bias* and *variance* [82, 130]. 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 [130] is a tool that can help with understanding the behaviour of machine learning methods, and is discussed where appropriate throughout the chapter.

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 text. They include removing attributes, such as attempted by Tumer and Ghosh [183]; manipulating the class labels, as used by Dietterich and Bakiri [49] in their *error-correcting output codes* technique; and introducing randomness to the base model inducer, such as Breiman's popular *random forest* approach [28].

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.

Batch Setting

Bagging

Bagging (**b**ootstrap **agg**regat**ing**) was introduced by Breiman [23]. 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.

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 [186]. Two concepts were defined by Kearns and Valiant [120], 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" [167] 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
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 hypo text.

Schapire's work was improved by Freund [62], 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

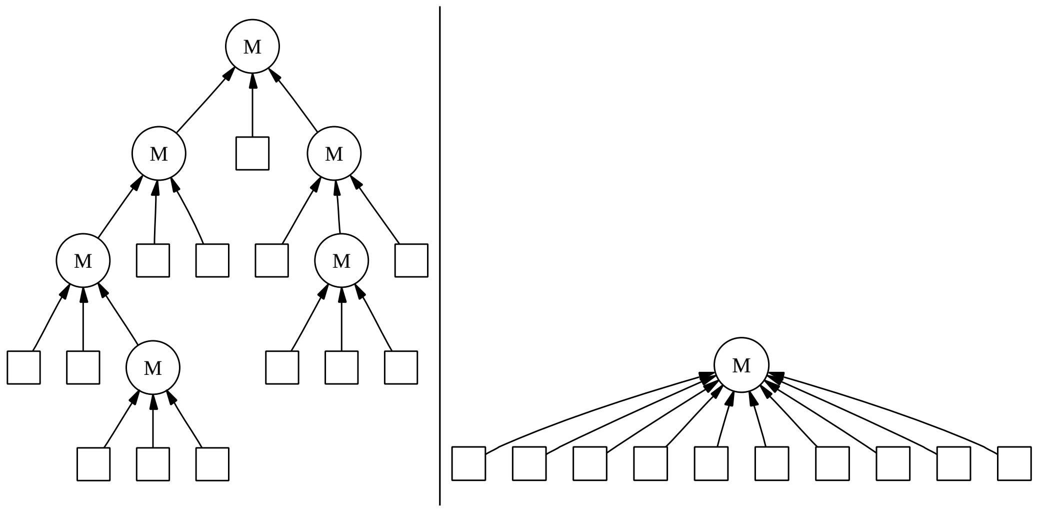


Figure 6.2: Left hand side: recursive tree structure used by original hypothesis boosting. Right hand side: flat structure of *boost-by-majority* and AdaBoost.

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 [65]. 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* [136], 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 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 $h_t : X \to Y$
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: $h_{fin}(x) = \arg\max_{y \in Y} \sum_{t:h_t(x)=y} \log 1/\beta_t$
```

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 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 [24] compared the approaches he found that accuracy did not significantly differ between the two.

Breiman looked at AdaBoost from a different perspective [24]. He introduced his own terminology, describing the procedure as *arcing* (**a**daptively **r**esample 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 [24], 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 $h_t : X \rightarrow Y$
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 [24] 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 [66].

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. [169] 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 [25, 27] 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 [68] 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 [170] 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 [48] 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 [63], 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 [142] found that BrownBoost is more robust than AdaBoost in class noise, but that LogitBoost performs at a comparable level.

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 [32] and further explored by Kohavi and Kunz [128], 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 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.

[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.

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 [128, 64].

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 [64].

The option tree induction approach by Kohavi and Kunz [128] 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 [148], 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 [128] 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 [32], who argued that options are more effective nearer the leaves.

Data Stream Setting

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 [154] 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 [154], and have collected experimental evidence to show this [153]. One issue they have not addressed is memory management, a consideration that is paramount in this text. 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.

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 [154, 57]. 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 text 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 [139] 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' [26], and the study by Fan et al. [57].

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 [59] online adaptation of arc-x4, and Domingo and Watanabe's [50] *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 [30] 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.

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: $h_{fin}(x) = \arg\max_{y \in Y} \sum_{m: h_m(x) = y} \log 1/\beta_m$
```

The data stream boosting approach adopted by this text is a parallel boosting algorithm that was developed by Oza and Russell [154], 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 clear. Preliminary experimental work for this text 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.

Option Trees

A new algorithm for inducing option trees from data streams was devised for this text. 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 [128]. 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 48. 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

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
```

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 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 text 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 text. 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 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

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
```

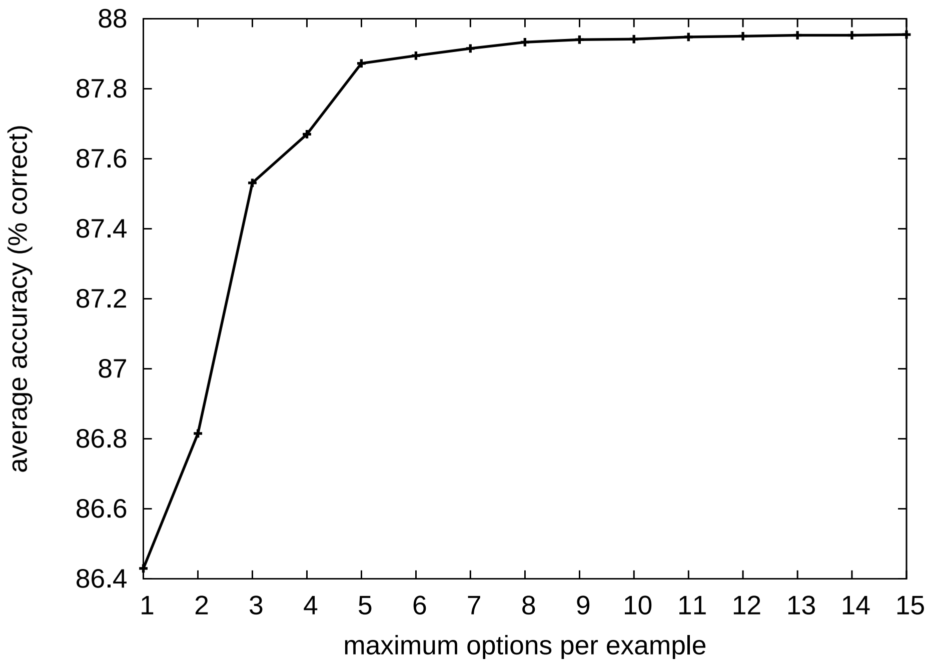


Figure 6.3: Average accuracy of Hoeffding option tree over many data sets versus the maximum number of options per example. Accuracies were estimated in unbounded memory.

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.

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.

Part III

Evolving Data Stream Learning

7 Evolving data streams

Data mining has traditionally been performed over static datasets, where data mining algorithms can afford to read the input data several times. When the source of data items is an open-ended data stream, not all data can be loaded into the memory and off-line mining with a fixed size dataset is no longer technically feasible due to the unique features of streaming data.

The following constraints apply in the Data Stream model:

1. The amount of data that has arrived and will arrive in the future is extremely large; in fact, the sequence is potentially infinite. Thus, it is impossible to store it all. Only a small summary can be computed and stored, and the rest of the information is thrown away. Even if the information could be all stored, it would be unfeasible to go over it for further processing.

2. The speed of arrival is large, so that each particular element has to be processed essentially in real time, and then discarded.

3. The distribution generating the items can change over time. Thus, data from the past may become irrelevant (or even harmful) for the current summary.

Constraints 1 and 2 limit the amount of memory and time-per-item that the streaming algorithm can use. Intense research on the algorithmics of Data Streams has produced a large body of techniques for computing fundamental functions with low memory and time-per-item, usually in combination with the sliding-window technique discussed next.

Constraint 3, the need to adapt to time changes, has been also intensely studied. A typical approach for dealing is based on the use of so-called sliding windows: The algorithm keeps a window of size W containing the last W data items that have arrived (say, in the last W time steps). When a new item arrives, the oldest element in the window is deleted to make place for it. The summary of the Data Stream is at every moment computed or rebuilt from the data in the window only. If W is of moderate size, this essentially takes care of the requirement to use low memory.

In most cases, the quantity W is assumed to be externally defined, and fixed through the execution of the algorithm. The underlying hypothesis is that the user can guess W so that the distribution of the data can be thought to be essentially constant in most intervals of size W; that is, the distribution changes smoothly at a rate that is small w.r.t. W, or it can change drastically from time to time, but the time between two drastic changes is often much greater than W.

Unfortunately, in most of the cases the user does not know in advance what the rate of change is going to be, so its choice of W is unlikely to be optimal. Not only that, the rate of change can itself vary over time, so the optimal W may itself vary over time.

Algorithms for mining with change

In this section we review some of the data mining methods that deal with data streams and concept drift. There are many algorithms in the literature that address this problem. We focus on the ones that they are more referred to in other works.

OLIN: Last

Last in [132] describes an online classification system that uses the info-fuzzy network (IFN). The system called OLIN (On Line Information Network) gets a continuous stream of non-stationary data and builds a network based on a sliding window of the latest examples. The system dynamically adapts the size of the training window and the frequency of model re-construction to the current rate of concept drift

OLIN uses the statistical significance of the difference between the training and the validation accuracy of the current model as an indicator of concept stability.

OLIN adjusts dynamically the number of examples between model reconstructions by using the following heuristic: keep the current model for more examples if the concept appears to be stable and reduce drastically the size of the validation window, if a concept drift is detected.

OLIN generates a new model for every new sliding window. This approach ensures accurate and relevant models over time and therefore an increase in the classification accuracy. However, the OLIN algorithm has a major drawback, which is the high cost of generating new models. OLIN does not take into account the costs involved in replacing the existing model with a new one.

CVFDT: Domingos

Hulten, Spencer and Domingos presented Concept-adapting Very Fast Decision Trees CVFDT [106] algorithm as an extension of VFDT to deal with concept change.

Figure 7.1 shows CVFDT algorithm. CVFDT keeps the model it is learning in sync with such changing concepts by continuously monitoring the quality of old search decisions with respect to a sliding window of data from the data stream, and updating them in a fine-grained way when it detects that the distribution of data is changing. In particular, it maintains sufficient statistics throughout time

```
CVFDT($Stream, \delta$)
1  Let HT be a tree with a single leaf(root)
2  Init counts $n_{ijk}$ at root
3  for each example $(x, y)$ in Stream
4        do Add, Remove and Forget Examples
5           CVFDTGrow($(x, y), HT, \delta$)
6           CheckSplitValidity($HT, n, \delta$)

CVFDTGrow($(x, y), HT, \delta$)
1  Sort $(x, y)$ to leaf $l$ using $HT$
2  Update counts $n_{ijk}$ at leaf $l$ and nodes traversed in the sort
3  if examples seen so far at $l$ are not all of the same class
4     then Compute $G$ for each attribute
5          if $G$(Best Attr.)−$G$(2nd best) $> \sqrt{\frac{R^2 \ln 1/\delta}{2n}}$
6             then Split leaf on best attribute
7                  for each branch
8                      do Start new leaf and initialize counts
9                  Create alternate subtree

CheckSplitValidity($HT, n, \delta$)
1  for each node $l$ in $HT$ that it is not a leaf
2        do for each tree $T_{alt}$ in ALT(l)
3              do CheckSplitValidity($T_{alt}, n, \delta$)
4           if exists a new promising attributes at node $l$
5              do Start an alternate subtree
```

Figure 7.1: The CVFDT algorithm

for every candidate M considered at every search step. After the first w examples, where w is the window width, it subtracts the oldest example from these statistics whenever a new one is added. After every Δn new examples, it determines again the best candidates at every previous search decision point. If one of them is better than an old winner by δ^* then one of two things has happened. Either the original decision was incorrect (which will happen a fraction δ of the time) or concept drift has occurred. In either case,it begins an alternate search starting from the new winners, while continuing to pursue the original search. Periodically it uses a number of new examples as a validation set to compare the performance of the models produced by the new and old searches. It prunes an old search (and replace it with the new one) when the new model is on average better than the old one, and it prunes the new search if after a maximum number of validations its models have failed to become more accurate on average than the old ones. If more than a maximum number of new searches is in progress, it prunes the lowest-performing ones.

UFFT: Gama

Gama, Medas and Rocha [73] presented the Ultra Fast Forest of Trees (UFFT) algorithm.

UFFT is an algorithm for supervised classification learning, that generates a forest of binary trees. The algorithm is incremental, processing each example in constant time, works on-line, UFFT is designed for continuous data. It uses analytical techniques to choose the splitting criteria, and the information gain to estimate the merit of each possible splitting-test. For multi-class problems, the algorithm builds a binary tree for each possible pair of classes leading to a forest-of-trees. During the training phase the algorithm maintains a short term memory. Given a data stream, a limited number of the most recent examples are maintained in a data structure that supports constant time insertion and deletion. When a test is installed, a leaf is transformed into a decision node with two descendant leaves. The sufficient statistics of the leaf are initialized with the examples in the short term memory that will fall at that leaf.

The UFFT algorithm maintains, at each node of all decision trees, a Naïve Bayes classifier. Those classifiers were constructed using the sufficient statistics needed to evaluate the splitting criteria when that node was a leaf. After the leaf becomes a node, all examples that traverse the node will be classified by the Naïve Bayes. The basic idea of the drift detection method is to control this error-rate. If the distribution of the examples is stationary, the error rate of Naïve-Bayes decreases. If there is a change on the distribution of the examples the Naïve Bayes error increases. The system uses DDM, the drift detection method explained in Section 1.5. When it detects an statistically significant increase of the Naïve-Bayes error in a given node, an indication of a change in the distribution of the examples, this suggest that the splitting-test that has been installed at this node is no longer appropriate. The subtree rooted at that node is pruned, and the node becomes a leaf. All the sufficient statistics of the leaf are initialized. When a new training example becomes available, it will cross the corresponding binary decision trees from the root node till a leaf. At each node, the Naïve Bayes installed at that node classifies the example. The example will be correctly or incorrectly classified. For a set of examples the error is a random variable from Bernoulli trials. The Binomial distribution gives the general form of the probability for the random variable that represents the number of errors in a sample of n examples.

The sufficient statistics of the leaf are initialized with the examples in the short term memory that maintains a limited number of the most recent examples. It is possible to observe an increase of the error reaching the warning level, followed by a decrease. This method uses the information already available to the learning algorithm and does not require additional computational resources. An advantage of this method is it continuously monitors the online error of Naïve Bayes. It can detect changes in the class-distribution of the examples at any time. All decision nodes contain Naïve Bayes to detect changes in the class distribution of the examples that traverse the node, that correspond to detect shifts in different regions of the instance space. Nodes near the root should be able to

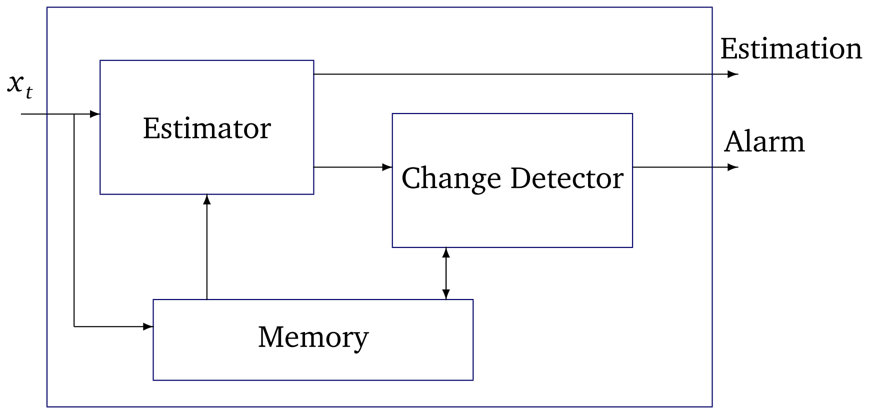


Figure 7.2: General Framework

detect abrupt changes in the distribution of the examples, while deeper nodes should detect smoothed changes.

A Methodology for Adaptive Stream Mining

In the data stream mining literature, most algorithms incorporate one or more of the following ingredients: windows to remember recent examples; methods for detecting distribution change in the input; and methods for keeping updated estimations for some statistics of the input. We see them as the basis for solving the three central problems of

- what to remember or forget,
- when to do the model upgrade, and
- how to do the model upgrade.

Our claim is that by basing mining algorithms on well-designed, well-encapsulated modules for these tasks, one can often get more generic and more efficient solutions than by using ad-hoc techniques as required.

Time Change Detectors and Predictors: A General Framework

Most approaches for predicting and detecting change in streams of data can be discussed in the general framework: The system consists of three modules: a Memory module, an Estimator Module, and a Change Detector or Alarm Generator module. These three modules interact as shown in Figure 7.2, which is analogous to Figure 8 in [172].
In general, the input to this algorithm is a sequence $x_1, x_2, \dots, x_t, \dots$ of data items whose distribution varies over time in an unknown way. The outputs of the algorithm are, at each time step

- an estimation of some important parameters of the input distribution, and

- a signal alarm indicating that distribution change has recently occurred.

We consider a specific, but very frequent case, of this setting: that in which all the x_t are real values. The desired estimation is usually the expected value of the current x_t, and less often another distribution statistics such as the variance. The only assumption on the distribution is that each x_t is drawn independently from each other.

Memory is the component where the algorithm stores all the sample data or summary that considers relevant at current time, that is, that presumably shows the current data distribution.

The Estimator component is an algorithm that estimates the desired statistics on the input data, which may change over time. The algorithm may or may not use the data contained in the Memory. The simplest Estimator algorithm for the expected is the *linear estimator,* which simply returns the average of the data items contained in the Memory. Other examples of run-time efficient estimators are Auto-Regressive, Auto Regressive Moving Average, and Kalman filters.

The change detector component outputs an alarm signal when it detects change in the input data distribution. It uses the output of the Estimator, and may or may not in addition use the contents of Memory.

In Table 7.1 we classify these predictors in four classes, depending on whether Change Detector and Memory modules exist:

	No memory	**Memory**
No Change Detector	*Type I* Kalman Filter	*Type III* Adaptive Kalman Filter
Change Detector	*Type II* Kalman Filter + CUSUM	*Type IV* `ADWIN` Kalman Filter + `ADWIN`

Table 7.1: Types of Time Change Predictor and some examples

- *Type I: Estimator only.* The simplest one is modelled by

$$\hat{x}_k = (1-\alpha)\hat{x}_{k-1} + \alpha \cdot x_k.$$

  The linear estimator corresponds to using $\alpha = 1/N$ where N is the width of a virtual window containing the last N elements we want to consider. Otherwise, we can give more weight to the last elements with an appropriate constant value of α. The Kalman filter tries to optimize the estimation using a non-constant α (the K value) which varies at each discrete time interval.

- *Type II: Estimator with Change Detector.* An example is the Kalman Filter together with a CUSUM test change detector algorithm, see for example [111].
- *Type III: Estimator with Memory.* We add Memory to improve the results of the Estimator. For example, one can build an Adaptive Kalman Filter that uses the data in Memory to compute adequate values for the process variance Q and the measure variance R. In particular, one can use the sum of the last elements stored into a memory window to model the Q parameter and the difference of the last two elements to estimate parameter R.
- *Type IV: Estimator with Memory and Change Detector.* This is the most complete type. Two examples of this type, from the literature, are:
  - A Kalman filter with a CUSUM test and fixed-length window memory, as proposed in [172]. Only the Kalman filter has access to the memory.
  - A linear Estimator over fixed-length windows that flushes when change is detected [122], and a change detector that compares the running windows with a reference window.

  In Chapter 8, we will present `ADWIN`, an adaptive sliding window method that works as a type IV change detector and predictor.

Window Management Models

Window strategies have been used in conjunction with mining algorithms in two ways: one, externally to the learning algorithm; the window system is used to monitor the error rate of the current model, which under stable distributions should keep decreasing or at most stabilize; when instead this rate grows significantly, change is declared and the base learning algorithm is invoked to revise or rebuild the model with fresh data. Note that in this case the window memory contains bits or real numbers (not full examples). Figure 7.3 shows this model.

The other way is to embed the window system *inside* the learning algorithm, to maintain the statistics required by the learning algorithm continuously updated; it is then the algorithm's responsibility to keep the model in synchrony with these statistics, as shown in Figure 7.4.

Learning algorithms that detect change, usually compare statistics of two windows. Note that the methods may be memoryless: they may keep window statistics without storing all their elements. There have been in the literature, some different window management strategies:

- Equal & fixed size subwindows: Kifer et al. [122] compares one reference, non-sliding, window of older data with a sliding window of the same size keeping the most recent data.
- Equal size adjacent subwindows: Dasu et al. [41] compares two adjacent sliding windows of the same size of recent data.

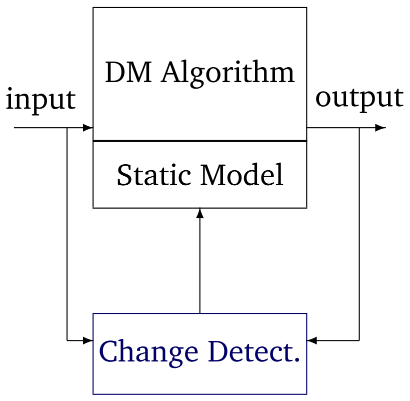


Figure 7.3: Data mining algorithm framework with concept drift.

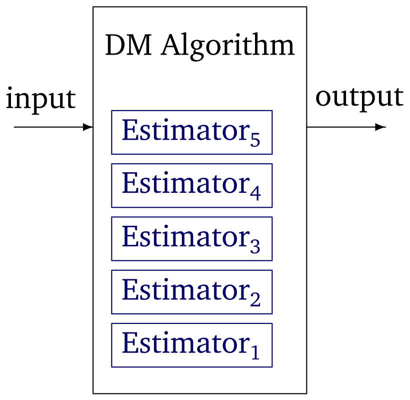


Figure 7.4: Data mining algorithm framework with concept drift using estimators replacing counters.

- Total window against subwindow: Gama et al. [72] compares the window that contains all the data with a subwindow of data from the beginning until it detects that the accuracy of the algorithm decreases.

The strategy of ADWIN, the method presented in next chapter, will be to compare all the adjacent subwindows in which is possible to partition the window containing all the data. Figure 7.5 shows these window management strategies.

Optimal Change Detector and Predictor

We have presented in section 7.2 a general framework for time change detectors and predictors. Using this framework we can establish the main properties of an optimal change detector and predictor system as the following:

- High accuracy
- Fast detection of change

Let $W = \boxed{101010110111111}$

- Equal & fixed size subwindows: $\boxed{1010}1011011\boxed{1111}$
- Equal size adjacent subwindows: $1010101\boxed{1011}\boxed{1111}$
- Total window against subwindow: $\boxed{\boxed{10101011011}1111}$
- ADWIN: All adjacent subwindows:

$$\boxed{1}\boxed{01010110111111}$$
$$\boxed{1010}\boxed{10110111111}$$
$$\boxed{1010101}\boxed{10111111}$$
$$\boxed{1010101101}\boxed{11111}$$
$$\boxed{10101011011111}\boxed{1}$$

Figure 7.5: Different window management strategies

- Low false positives and negatives ratios
- Low computational cost: minimum space and time needed
- Theoretical guarantees
- No parameters needed
- Detector of type IV: Estimator with Memory and Change Detector

In the next chapter we will present ADWIN, a change detector and predictor with these characteristics, using an adaptive sliding window model. ADWIN's window management strategy will be to compare all the adjacent subwindows in which is possible to partition the window containing all the data. It seems that this procedure may be the most accurate, since it looks at all possible subwindows partitions. On the other hand, time cost is the main disadvantage of this method. Considering this, we will provide another version working in the strict conditions of the Data Stream model, namely low memory and low processing per item.

8 Adaptive Sliding Windows

Dealing with data whose nature changes over time is one of the core problems in data mining and machine learning. In this chapter we present ADWIN, an adaptive sliding window algorithm, as an estimator with memory and change detector with the main properties of optimality explained in section 7.3. We study and develop also the combination of ADWIN with Kalman filters.

Introduction

Most strategies in the literature use variations of the sliding window idea: a window is maintained that keeps the most recently read examples, and from which older examples are dropped according to some set of rules. The contents of the window can be used for the three tasks: 1) to detect change (e.g., by using some statistical test on different subwindows), 2) obviously, to obtain updated statistics from the recent examples, and 3) to have data to rebuild or revise the model(s) after data has changed.

The simplest rule is to keep a window of some fixed size, usually determined *a priori* by the user. This can work well if information on the time-scale of change is available, but this is rarely the case. Normally, the user is caught in a tradeoff without solution: choosing a small size (so that the window reflects accurately the current distribution) and choosing a large size (so that many examples are available to work on, increasing accuracy in periods of stability). A different strategy uses a *decay function* to weight the importance of examples according to their age (see e.g. [39]): the relative contribution of each data item is scaled down by a factor that depends on elapsed time. In this case, the tradeoff shows up in the choice of a decay constant that should match the unknown rate of change.

Less often, it has been proposed to use windows of variable size. In general, one tries to keep examples as long as possible, i.e., while not proven stale. This delivers the users from having to guess *a priori* an unknown parameter such as the time scale of change. However, most works along these lines that we know of (e.g., [72, 125, 132, 192]) are heuristics and have no rigorous guarantees of performance. Some works in computational learning theory (e.g. [13, 97, 98]) describe strategies with rigorous performance bounds, but to our knowledge they have never been tried in real learning/mining contexts and often assume a known bound on the rate of change.

We will present `ADWIN`, a parameter-free adaptive size sliding window, with theoretical garantees. We will use Kalman filters at the last part of this Chapter, in order to provide an adaptive weight for each item.

Maintaining Updated Windows of Varying Length

In this section we describe algorithms for dynamically adjusting the length of a data window, make a formal claim about its performance, and derive an efficient variation.

We will use Hoeffding's bound in order to obtain formal guarantees, and a streaming algorithm. However, other tests computing differences between window distributions may be used.

Setting

The inputs to the algorithms are a confidence value $\delta \in (0,1)$ and a (possibly infinite) sequence of real values $x_1, x_2, x_3, \ldots, x_t, \ldots$ The value of x_t is available only at time t. Each x_t is generated according to some distribution D_t, independently for every t. We denote with μ_t and σ_t^2 the expected value and the variance of x_t when it is drawn according to D_t. We assume that x_t is always in $[0,1]$; by an easy rescaling, we can handle any case in which we know an interval $[a,b]$ such that $a \le x_t \le b$ with probability 1. Nothing else is known about the sequence of distributions D_t; in particular, μ_t and σ_t^2 are unknown for all t.

First algorithm: `ADWIN0`

The first algorithm keeps a sliding window W with the most recently read x_i. Let n denote the length of W, $\hat{\mu}_W$ the (observed) average of the elements in W, and μ_W the (unknown) average of μ_t for $t \in W$. Strictly speaking, these quantities should be indexed by t, but in general t will be clear from the context.

Since the values of μ_t can oscillate wildly, there is no guarantee that μ_W or $\hat{\mu}_W$ will be anywhere close to the instantaneous value μ_t, even for long W. However, μ_W is the expected value of $\hat{\mu}_W$, so μ_W and $\hat{\mu}_W$ *do* get close as W grows.

Algorithm `ADWIN0` is presented in Figure 8.1. The idea is simple: whenever two "large enough" subwindows of W exhibit "distinct enough" averages, one can conclude that the corresponding expected values are different, and the older portion of the window is dropped. In other words, W is kept as long as possible while the null hypothesis "μ_t has remained constant in W" is sustainable up to confidence δ.[1] "Large enough" and "distinct enough" above are made precise by choosing an appropriate statistical test for distribution change, which in general involves the value of δ, the lengths of the subwindows, and their contents. We choose one particular statistical test for our implementation, but this is not the

[1]It would easy to use instead the null hypothesis "there has been no change greater than ϵ", for a user-specified ϵ expressing the smallest change that deserves reaction.

```
ADWIN0: ADAPTIVE WINDOWING ALGORITHM
1  Initialize Window $W$
2  for each $t > 0$
3        do $W \leftarrow W \cup \{x_t\}$ (i.e., add $x_t$ to the head of $W$)
4            repeat Drop elements from the tail of $W$
5              until $|\hat{\mu}_{W_0} - \hat{\mu}_{W_1}| < \epsilon_{cut}$ holds
6                  for every split of $W$ into $W = W_0 \cdot W_1$
7            output $\hat{\mu}_W$
```

Figure 8.1: Algorithm ADWIN0.

essence of our proposal – many other tests could be used. At every step, ADWIN0 simply outputs the value of $\hat{\mu}_W$ as an approximation to μ_W.

The value of ϵ_{cut} for a partition $W_0 \cdot W_1$ of W is computed as follows: Let n_0 and n_1 be the lengths of W_0 and W_1 and n be the length of W, so $n = n_0 + n_1$. Let $\hat{\mu}_{W_0}$ and $\hat{\mu}_{W_1}$ be the averages of the values in W_0 and W_1, and μ_{W_0} and μ_{W_1} their expected values. To obtain totally rigorous performance guarantees we define:

$$\begin{aligned} m &= \frac{1}{1/n_0 + 1/n_1} \text{ (harmonic mean of } n_0 \text{ and } n_1\text{)}, \\ \delta' &= \frac{\delta}{n}, \text{ and } \quad \epsilon_{cut} = \sqrt{\frac{1}{2m} \cdot \ln \frac{4}{\delta'}}\,. \end{aligned}$$

Our statistical test for different distributions in W_0 and W_1 simply checks whether the observed average in both subwindows differs by more than the threshold ϵ_{cut}. The role of δ' is to avoid problems with multiple hypothesis testing (since we will be testing n different possibilities for W_0 and W_1 and we want global error below δ). Later we will provide a more sensitive test based on the normal approximation that, although not 100% rigorous, is perfectly valid in practice.

Now we state our main technical result about the performance of ADWIN0:

Theorem 1. *At every time step we have*

1. (False positive rate bound). *If μ_t remains constant within W, the probability that* ADWIN0 *shrinks the window at this step is at most δ.*
2. (False negative rate bound). *Suppose that for* some *partition of W in two parts W_0W_1 (where W_1 contains the most recent items) we have $|\mu_{W_0} - \mu_{W_1}| > 2\epsilon_{cut}$. Then with probability $1 - \delta$* ADWIN0 *shrinks W to W_1, or shorter.*

Proof. **Part 1)** Assume $\mu_{W_0} = \mu_{W_1} = \mu_W$ as null hypothesis. We show that for any partition W as W_0W_1 we have probability at most δ/n that ADWIN0 decides to shrink W to W_1, or equivalently,

$$\Pr[\,|\hat{\mu}_{W_1} - \hat{\mu}_{W_0}| \geq \epsilon_{cut}\,] \leq \delta/n.$$

Since there are at most n partitions W_0W_1, the claim follows by the union bound. Note that, for every real number $k \in (0,1)$, $|\hat{\mu}_{W_1} - \hat{\mu}_{W_0}| \geq \epsilon_{cut}$ can be decomposed as

$$\Pr[\,|\hat{\mu}_{W_1} - \hat{\mu}_{W_0}| \geq \epsilon_{cut}\,] \;\leq\; \Pr[\,|\hat{\mu}_{W_1} - \mu_W| \geq k\epsilon_{cut}\,] + \Pr[\,|\mu_W - \hat{\mu}_{W_0}| \geq (1-k)\epsilon_{cut})\,].$$

Applying the Hoeffding bound, we have then

$$\Pr[\,|\hat{\mu}_{W_1} - \hat{\mu}_{W_0}| \geq \epsilon_{cut}\,] \;\leq\; 2\exp(-2(k\,\epsilon_{cut})^2\,n_0) + 2\exp(-2((1-k)\,\epsilon_{cut})^2\,n_1)$$

To approximately minimize the sum, we choose the value of k that makes both probabilities equal, i.e. such that

$$(k\,\epsilon_{cut})^2\,n_0 = ((1-k)\,\epsilon_{cut})^2\,n_1.$$

which is $k = \sqrt{n_1/n_0}/(1+\sqrt{n_1/n_0})$. For this k, we have precisely

$$(k\,\epsilon_{cut})^2\,n_0 = \frac{n_1 n_0}{(\sqrt{n_0}+\sqrt{n_1})^2}\,\epsilon_{cut}^2 \leq \frac{n_1 n_0}{(n_0+n_1)}\,\epsilon_{cut}^2 = m\,\epsilon_{cut}^2.$$

Therefore, in order to have

$$\Pr[\,|\hat{\mu}_{W_1} - \hat{\mu}_{W_0}| \geq \epsilon_{cut}\,] \leq \frac{\delta}{n}$$

it suffices to have

$$4\exp(-2m\,\epsilon_{cut}^2) \leq \frac{\delta}{n}$$

which is satisfied by

$$\epsilon_{cut} = \sqrt{\frac{1}{2m}\,\ln\frac{4n}{\delta}}\,.$$

Part 2) Now assume $|\mu_{W_0} - \mu_{W_1}| > 2\epsilon_{cut}$. We want to show that $\Pr[\,|\hat{\mu}_{W_1} - \hat{\mu}_{W_0}| \leq \epsilon_{cut}\,] \leq \delta$, which means that with probability at least $1-\delta$ change is detected and the algorithm cuts W to W_1. As before, for any $k \in (0,1)$, we can decompose $|\hat{\mu}_{W_0} - \hat{\mu}_{W_1}| \leq \epsilon_{cut}$ as

$$\begin{aligned}\Pr[\,|\hat{\mu}_{W_0} - \hat{\mu}_{W_1}| \leq \epsilon_{cut}\,] \;&\leq\; \Pr[\,(|\hat{\mu}_{W_0} - \mu_{W_0}| \geq k\epsilon_{cut}) \cup (|\hat{\mu}_{W_1} - \mu_{W_1}| \geq (1-k)\epsilon_{cut})\,]\\ &\leq\; \Pr[\,|\hat{\mu}_{W_0} - \mu_{W_0}| \geq k\epsilon_{cut}\,] + \Pr[\,|\hat{\mu}_{W_1} - \mu_{W_1}| \geq (1-k)\epsilon_{cut}\,].\end{aligned}$$

To see the first inequality, observe that if $|\hat{\mu}_{W_0} - \hat{\mu}_{W_1}| \leq \epsilon_{cut}$, $|\hat{\mu}_{W_0} - \mu_{W_0}| \leq k\epsilon_{cut}$, and $|\hat{\mu}_{W_1} - \mu_{W_1}| \leq (1-k)\epsilon_{cut}$ hold, by the triangle inequality we have

$$|\mu_{W_0} - \mu_{W_1}| \leq |\hat{\mu}_{W_0} + k\epsilon_{cut} - \hat{\mu}_{W_1} + (1-k)\epsilon_{cut}| \leq |\hat{\mu}_{W_0} - \hat{\mu}_{W_1}| + \epsilon_{cut} \leq 2\epsilon_{cut},$$

contradicting the hypothesis. Using the Hoeffding bound, we have then

$$\Pr[\,|\hat{\mu}_{W_0} - \hat{\mu}_{W_1}| \geq \epsilon_{cut}\,] \;\leq\; 2\,\exp(-2(k\,\epsilon_{cut})^2\,n_0) + 2\,\exp(-2((1-k)\,\epsilon_{cut})^2\,n_1).$$

Now, choose k as before to make both terms equal. By the calculations in Part 1 we have

$$\Pr[\,|\hat{\mu}_{W_0} - \hat{\mu}_{W_1}| \geq \epsilon_{cut}\,] \;\leq\; 4\exp(-2\,m\,\epsilon_{cut}^2) \leq \frac{\delta}{n} \leq \delta,$$

as desired.

$\square$

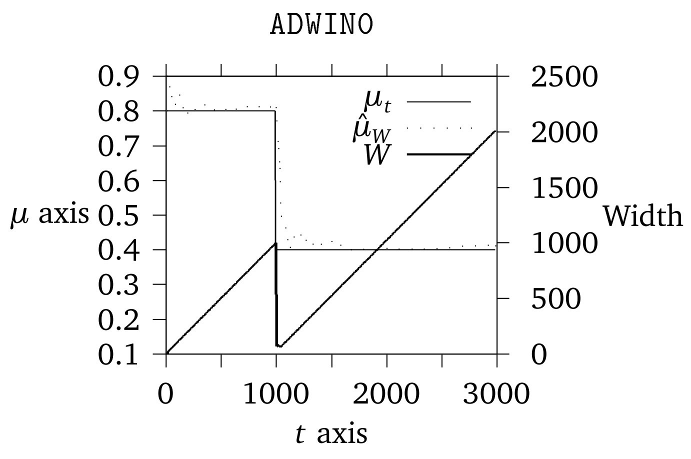


Figure 8.2: Output of algorithm ADWIN0 with abrupt change.

In practice, the definition of ϵ_{cut} as above is too conservative. Indeed, it is based on the Hoeffding bound, which is valid for all distributions but greatly overestimates the probability of large deviations for distributions of small variance; in fact, it is equivalent to assuming always the worst-case variance $\sigma^2 = 1/4$. In practice, one can observe that $\mu_{W_0} - \mu_{W_1}$ tends to a normal distribution for large window sizes, and use

$$\epsilon_{cut} = \sqrt{\frac{2}{m} \cdot \sigma_W^2 \cdot \ln \frac{2}{\delta'}} + \frac{2}{3\,m} \ln \frac{2}{\delta'}, \tag{8.1}$$

where σ_W^2 is the observed variance of the elements in window W. Thus, the term with the square root is essentially equivalent to setting ϵ_{cut} to k times the standard deviation, for k depending on the desired confidence δ, as is done in [72]. The extra additive term protects the cases where the window sizes are too small to apply the normal approximation, as an alternative to the traditional use of requiring, say, sample size at least 30; it can be formally derived from the so-called Bernstein bound. Additionally, one (somewhat involved) argument shows that setting $\delta' = \delta/(\ln n)$ is enough in this context to protect from the multiple hypothesis testing problem; anyway, in the actual algorithm that we will run (ADWIN), only $O(\log n)$ subwindows are checked, which justifies using $\delta' = \delta/(\ln n)$. Theorem 1 holds for this new value of ϵ_{cut}, up to the error introduced by the normal approximation. We have used these better bounds in all our implementations.

Let us consider how ADWIN0 behaves in two special cases: sudden (but infrequent) changes, and slow gradual changes. Suppose that for a long time μ_t has remained fixed at a value μ, and that it suddenly jumps to a value $\mu' = \mu + \epsilon$. By part (2) of Theorem 1 and Equation 8.1, one can derive that the window will start shrinking after $O(\mu \ln(1/\delta)/\epsilon^2)$ steps, and in fact will be shrunk to the point where only $O(\mu \ln(1/\delta)/\epsilon^2)$ examples prior to the change are left. From then on, if no further changes occur, no more examples will be dropped so the window will expand unboundedly.

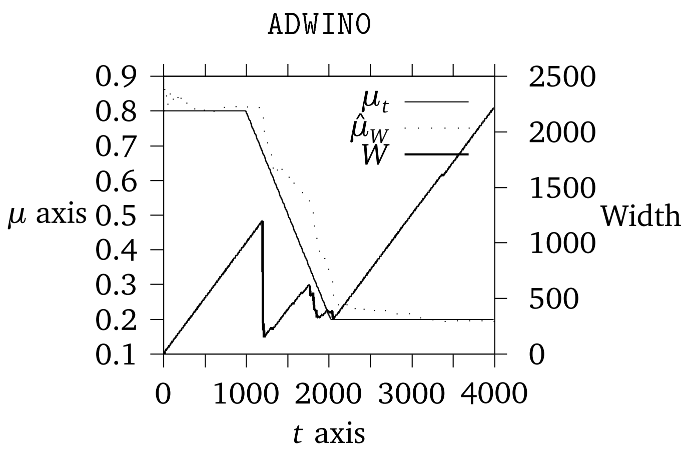


Figure 8.3: Output of algorithm ADWIN0 with slow gradual changes.

In case of a gradual change with slope α following a long stationary period at μ, observe that the average of W_1 after n_1 steps is $\mu + \alpha n_1/2$; we have $\epsilon(= \alpha n_1/2) \geq O(\sqrt{\mu \ln(1/\delta)/n_1})$ iff $n_1 = O(\mu \ln(1/\delta)/\alpha^2)^{1/3}$. So n_1 steps after the change the window will start shrinking, and will remain at approximately size n_1 from then on. A dependence on α of the form $O(\alpha^{-2/3})$ may seem odd at first, but one can show that this window length is actually optimal in this setting, even if α is known: it minimizes the sum of variance error (due to short window) and error due to out-of-date data (due to long windows in the presence of change). Thus, in this setting, ADWIN0 provably adjusts automatically the window setting to its optimal value, up to multiplicative constants.

Figures 8.2 and 8.3 illustrate these behaviors. In Figure 8.2, a sudden change from $\mu_{t-1} = 0.8$ to $\mu_t = 0.4$ occurs, at $t = 1000$. One can see that the window size grows linearly up to $t = 1000$, that ADWIN0 cuts the window severely 10 steps later (at $t = 1010$), and that the window expands again linearly after time $t = 1010$. In Figure 8.3, μ_t gradually descends from 0.8 to 0.2 in the range $t \in [1000..2000]$. In this case, ADWIN0 cuts the window sharply at t around 1200 (i.e., 200 steps after the slope starts), keeps the window length bounded (with some random fluctuations) while the slope lasts, and starts growing it linearly again after that. As predicted by theory, detecting the change is harder in slopes than in abrupt changes.

ADWIN0 for Poisson processes

A Poisson process is the stochastic process in which events occur continuously and independently of one another. A well-known example is radioactive decay of atoms. Many processes are not exactly Poisson processes, but similar enough that for certain types of analysis they can be regarded as such; e.g., telephone calls arriving at a switchboard, webpage requests to a search engine, or rainfall.

Using the Chernoff bound for Poisson processes [145]

$$\Pr\{X \geq cE[X]\} \leq \exp(-(c\ln(c) + 1 - c)E[X])$$

we find a similar ϵ_{cut} for Poisson processes.

First, we look for a simpler form of this bound. Let $c = 1 + \epsilon$ then

$$c\ln(c) - c + 1 = (1+\epsilon)\cdot\ln(1+\epsilon) - \epsilon$$

Using the Taylor expansion of $\ln(x)$

$$\ln(1+x) = \sum(-1)^{n+1}\cdot\frac{x^n}{n} = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots$$

we get the following simpler expression:

$$\Pr\{X \geq (1+\epsilon)E[X]\} \leq \exp(-\epsilon^2 E[X]/2)$$

Now, let S_n be the sum of n Poisson processes. As S_n is also a Poisson process

$$E[S_n] = \lambda_{S_n} = nE[X] = n\cdot\lambda_X$$

and then we obtain

$$\Pr\{S_n \geq (1+\epsilon)E[S_n]\} \leq \exp(-\epsilon^2 E[S_n]/2)$$

In order to obtain a formula for ϵ_{cut}, let $Y = S_n/n$

$$\Pr\{Y \geq (1+\epsilon)E[Y]\} \leq \exp(-\epsilon^2\cdot n\cdot E[Y]/2)$$

And finally, with this bound we get the following ϵ_{cut} for ADWIN0

$$\epsilon_{cut} = \sqrt{\frac{2\lambda}{m}\ln\frac{2}{\delta}}$$

where $1/m = 1/n_0 + 1/n_1$, and λ is the mean of the window data.

Improving time and memory requirements

The first version of ADWIN0 is computationally expensive, because it checks exhaustively all "large enough" subwindows of the current window for possible cuts. Furthermore, the contents of the window is kept explicitly, with the corresponding memory cost as the window grows. To reduce these costs we present a new version ADWIN using ideas developed in data stream algorithmics [10, 149, 11, 42] to find a good cutpoint quickly. Figure 8.4 shows the ADWIN algorithm. We next provide a sketch of how this algorithm and these data structures work.

Our data structure is a variation of exponential histograms [42], a data structure that maintains an approximation of the number of 1's in a sliding window of length W with logarithmic memory and update time. We adapt this data structure in a way that can provide this approximation simultaneously for about $O(\log W)$ subwindows whose lengths follow a geometric law, *with no memory overhead* with respect to keeping the count for a single window. That is, our data structure will be able to give the number of 1s among the most recently

```
ADWIN: ADAPTIVE WINDOWING ALGORITHM
1  Initialize $W$ as an empty list of buckets
2  Initialize WIDTH, VARIANCE and TOTAL
3  for each $t > 0$
4       do SETINPUT($x_t, W$)
5          output $\hat{\mu}_W$ as TOTAL/WIDTH and ChangeAlarm

SETINPUT(item e, List W)
1  INSERTELEMENT($e, W$)
2  repeat DELETEELEMENT($W$)
3     until $|\hat{\mu}_{W_0} - \hat{\mu}_{W_1}| < \epsilon_{cut}$ holds
4        for every split of $W$ into $W = W_0 \cdot W_1$

INSERTELEMENT(item e, List W)
1  create a new bucket $b$ with content $e$ and capacity 1
2  $W \leftarrow W \cup \{b\}$ (i.e., add $e$ to the head of $W$)
3  update WIDTH, VARIANCE and TOTAL
4  COMPRESSBUCKETS($W$)

DELETEELEMENT(List W)
1  remove a bucket from tail of List W
2  update WIDTH, VARIANCE and TOTAL
3  ChangeAlarm ← true

COMPRESSBUCKETS(List W)
1  Traverse the list of buckets in increasing order
2       do If there are more than $M$ buckets of the same capacity
3              do merge buckets
4                 COMPRESSBUCKETS(sublist of W not traversed)
```

Figure 8.4: Algorithm ADWIN.

$t-1, t-\lfloor c\rfloor, t-\lfloor c^2\rfloor, \ldots, t-\lfloor c^i\rfloor, \ldots$ read bits, with the same amount of memory required to keep an approximation for the whole W. Note that keeping exact counts for a fixed-window size is provably impossible in sublinear memory. We go around this problem by shrinking or enlarging the window strategically so that what would otherwise be an approximate count happens to be exact.

More precisely, to design the algorithm one chooses a parameter M. This parameter controls both 1) the amount of memory used (it will be $O(M \log W/M)$ words, and 2) the closeness of the cutpoints checked (the basis c of the geometric series above, which will be about $c = 1 + 1/M$). Note that the choice of M does *not* reflect any assumption about the time-scale of change: Since points are checked at a geometric rate anyway, this policy is essentially scale-independent.

More precisely, in the boolean case, the information on the number of 1's is kept as a series of buckets whose size is always a power of 2. We keep at most M buckets of each size 2^i, where M is a design parameter. For each bucket we record two (integer) elements: *capacity* and *content* (size, or number of 1s it contains).

Thus, we use about $M \cdot \log(W/M)$ buckets to maintain our data stream sliding window. ADWIN checks as a possible cut every border of a bucket, i.e., window lengths of the form $M(1+2+\cdots+2^{i-1}) + j \cdot 2^i$, for $0 \le j \le M$. It can be seen that these $M \cdot \log(W/M)$ points follow approximately a geometric law of basis $\cong 1 + 1/M$.

Let's look at an example: a sliding window with 14 elements. We register it as:

	1010101	101	11	1	1
Content:	4	2	2	1	1
Capacity:	7	3	2	1	1

Each time a new element arrives, if the element is "1", we create a new bucket of *content* 1 and *capacity* the number of elements arrived since the last "1". After that we compress the rest of buckets: When there are $M+1$ buckets of size 2^i, we merge the two oldest ones (adding its capacity) into a bucket of size 2^{i+1}. So, we use $O(M \cdot \log W/M)$ memory words if we assume that a word can contain a number up to W. In [42], the window is kept at a fixed size W. The information missing about the last bucket is responsible for the approximation error. Here, each time we detect change, we reduce the window's length deleting the last bucket, instead of (conceptually) dropping a single element as in a typical sliding window framework. This lets us keep an exact counting, since when throwing away a whole bucket we know that we are dropping exactly 2^i "1"s.

We summarize these results with the following theorem.

Theorem 2. *The* ADWIN *algorithm maintains a data structure with the following properties:*

- *It uses $O(M \cdot \log(W/M))$ memory words (assuming a memory word can contain numbers up to W).*
- *It can process the arrival of a new element in $O(1)$ amortized time and $O(\log W)$ worst-case time.*

- *It can provide the exact counts of* 1*'s for all the subwindows whose lengths are of the form* $\lfloor(1+1/M)^i\rfloor$*, in* $O(1)$ *time per query.*

Since ADWIN tries $O(\log W)$ cutpoints, the total processing time per example is $O(\log W)$ (amortized) and $O(\log W)$ (worst-case).

In our example, suppose $M = 2$, if a new element "1" arrives then

$\boxed{1010101}\ \boxed{101}\ \boxed{11}\ \boxed{1}\ \boxed{1}\ \boxed{1}$

Content: 4 2 2 1 1 1

Capacity: 7 3 2 1 1 1

There are 3 buckets of 1, so we compress it:

$\boxed{1010101}\ \boxed{101}\ \boxed{11}\ \boxed{11}\ \boxed{1}$

Content: 4 2 2 2 1

Capacity: 7 3 2 2 1

and now as we have 3 buckets of size 2, we compress it again

$\boxed{1010101}\ \boxed{10111}\ \boxed{11}\ \boxed{1}$

Content: 4 4 2 1

Capacity: 7 5 2 1

And finally, if we detect change, we reduce the size of our sliding window deleting the last bucket:

$\boxed{10111}\ \boxed{11}\ \boxed{1}$

Content: 4 2 1

Capacity: 5 2 1

In the case of real values, we also maintain buckets of two elements: *capacity* and *content*. We store at *content* the sum of the real numbers we want to summarize. We restrict *capacity* to be a power of two. As in the boolean case, we use $O(\log W)$ buckets, and check $O(\log W)$ possible cuts. The memory requirement for each bucket is $\log W + R + \log\log W$ bits per bucket, where R is number of bits used to store a real number.

Figure 8.5 shows the output of ADWIN to a sudden change, and Figure 8.6 to a slow gradual change. The main difference with ADWIN output is that as ADWIN0 reduces one element by one each time it detects changes, ADWIN deletes an entire bucket, which yields a slightly more jagged graph in the case of a gradual change. The difference in approximation power between ADWIN0 and ADWIN is almost negligible, so we use ADWIN exclusively for our experiments.

Finally, we state our main technical result about the performance of ADWIN, in a similar way to the Theorem 1:

Theorem 3. *At every time step we have*

1. (False positive rate bound). *If* μ_t *remains constant within* W*, the probability that* ADWIN *shrinks the window at this step is at most* $M/n \cdot \log(n/M) \cdot \delta$.

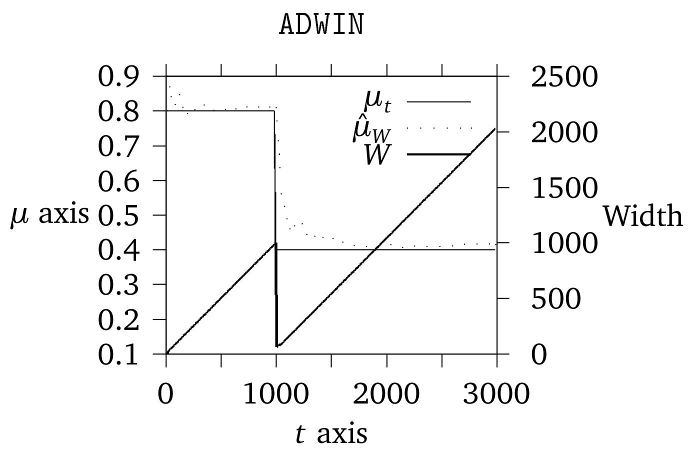


Figure 8.5: Output of algorithm ADWIN with abrupt change

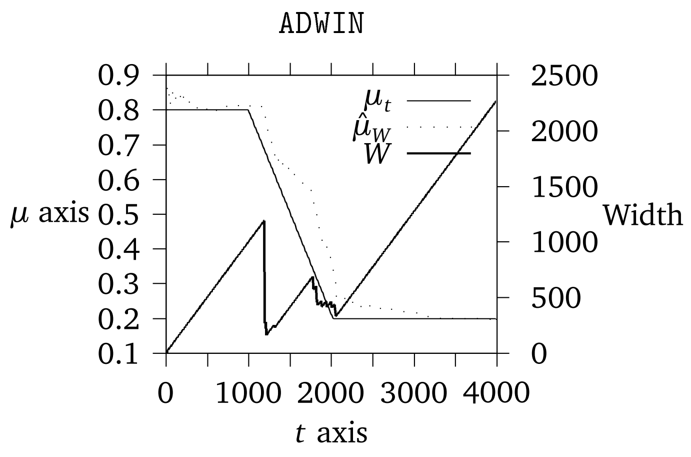


Figure 8.6: Output of algorithm ADWIN with slow gradual changes

2. (False negative rate bound). *Suppose that for* some *partition of W in two parts W_0W_1 (where W_1 contains the most recent items) we have $|\mu_{W_0} - \mu_{W_1}| > 2\epsilon_{cut}$. Then with probability $1-\delta$ ADWIN shrinks W to W_1, or shorter.*

Proof. **Part 1)** Assume $\mu_{W_0} = \mu_{W_1} = \mu_W$ as null hypothesis. We have shown in the proof of Theorem 1 that for any partition W as W_0W_1 we have probability at most δ/n that ADWIN0 decides to shrink W to W_1, or equivalently,

$$\Pr[\, |\hat{\mu}_{W_1} - \hat{\mu}_{W_0}| \ge \epsilon_{cut}\,] \le \delta/n.$$

Since ADWIN checks at most $M\log(n/M)$ partitions W_0W_1, the claim follows.

Part 2) The proof is similar to the proof of Part 2 of Theorem 1. □

K-ADWIN = ADWIN + Kalman Filtering

One of the most widely used Estimation algorithms is the Kalman filter. We give here a description of its essentials; see [190] for a complete introduction.

The Kalman filter addresses the general problem of trying to estimate the state $x \in \Re^n$ of a discrete-time controlled process that is governed by the linear stochastic difference equation

$$x_t = Ax_{t-1} + Bu_t + w_{t-1}$$

with a measurement $z \in \Re^m$ that is

$$Z_t = Hx_t + v_t.$$

The random variables w_t and v_t represent the process and measurement noise (respectively). They are assumed to be independent (of each other), white, and with normal probability distributions

$$p(w) \sim N(0,Q)$$

$$p(v) \sim N(0,R).$$

In essence, the main function of the Kalman filter is to estimate the state vector using system sensors and measurement data corrupted by noise.

The Kalman filter estimates a process by using a form of feedback control: the filter estimates the process state at some time and then obtains feedback in the form of (noisy) measurements. As such, the equations for the Kalman filter fall into two groups: time update equations and measurement update equations. The time update equations are responsible for projecting forward (in time) the current state and error covariance estimates to obtain the a priori estimates for the next time step.

$$x_t^- = Ax_{t-1} + Bu_t$$

$$P_t^- = AP_{t-1}A^T + Q$$

The measurement update equations are responsible for the feedback, i.e. for incorporating a new measurement into the a priori estimate to obtain an improved a posteriori estimate.

$$K_t = P_t^- H^T (HP_t^- H^T + R)^{-1}$$

$$x_t = x_t^- + K_t(z_t - Hx_t^-)$$

$$P_t = (I - K_t H)P_t^-.$$

There are extensions of the Kalman filter (Extended Kalman Filters, or EKF) for the cases in which the process to be estimated or the measurement-to-process relation is nonlinear. We do not discuss them here.

In our case we consider the input data sequence of real values $z_1, z_2, \ldots, z_t, \ldots$ as the measurement data. The difference equation of our discrete-time controlled process is the simpler one, with $A = 1, H = 1, B = 0$. So the equations are simplified to:

$$K_t = P_{t-1}/(P_{t-1} + R)$$

$$X_t = X_{t-1} + K_t(z_t - X_{t-1})$$
$$P_t = P_t(1 - K_t) + Q.$$

Note the similarity between this Kalman filter and an EWMA estimator, taking $\alpha = K_t$. This Kalman filter can be considered as an adaptive EWMA estimator where $\alpha = f(Q, R)$ is calculated optimally when Q and R are known.

The performance of the Kalman filter depends on the accuracy of the a-priori assumptions:

- linearity of the difference stochastic equation
- estimation of covariances Q and R, assumed to be fixed, known, and follow normal distributions with zero mean.

When applying the Kalman filter to data streams that vary arbitrarily over time, both assumptions are problematic. The linearity assumption for sure, but also the assumption that parameters Q and R are fixed and known – in fact, estimating them from the data is itself a complex estimation problem.

ADWIN is basically a linear Estimator with Change Detector that makes an efficient use of Memory. It seems a natural idea to improve its performance by replacing the linear estimator by an adaptive Kalman filter, where the parameters Q and R of the Kalman filter are computed using the information in ADWIN's memory.

We have set $R = W^2/50$ and $Q = 200/W$, where W is the length of the window maintained by ADWIN. While we cannot rigorously prove that these are the optimal choices, we have informal arguments that these are about the "right" forms for R and Q, on the basis of the theoretical guarantees of ADWIN.

Let us sketch the argument for Q. Theorem 1, part (2) gives a value ϵ for the maximum change that may have occurred within the window maintained by ADWIN. This means that the process variance within that window is at most ϵ^2, so we want to set $Q = \epsilon^2$. In the formula for ϵ, consider the case in which $n_0 = n_1 = W/2$, then we have

$$\epsilon \geq 4 \cdot \sqrt{\frac{3(\mu_{W_0} + \epsilon)}{W/2} \cdot \ln \frac{4W}{\delta}}$$

Isolating from this equation and distinguishing the extreme cases in which $\mu_{W_0} \gg \epsilon$ or $\mu_{W_0} \ll \epsilon$, it can be shown that $Q = \epsilon^2$ has a form that varies between c/W and d/W^2. Here, c and d are constant for constant values of δ, and $c = 200$ is a reasonable estimation. This justifies our choice of $Q = 200/W$. A similar, slightly more involved argument, can be made to justify that reasonable values of R are in the range W^2/c to W^3/d, for somewhat large constants c and d.

When there is no change, ADWIN window's length increases, so R increases too and K decreases, reducing the significance of the most recent data arrived. Otherwise, if there is change, ADWIN window's length reduces, so does R, and K increases, which means giving more importance to the last data arrived.

9 Adaptive Hoeffding Trees

In this chapter we propose and illustrate a method for developing decision trees algorithms that can adaptively learn from data streams that change over time. We take the Hoeffding Tree learner, an incremental decision tree inducer for data streams, and use as a basis it to build two new methods that can deal with distribution and concept drift: a sliding window-based algorithm, Hoeffding Window Tree, and an adaptive method, Hoeffding Adaptive Tree. Our methods are based on the methodology explained in Chapter 7. We choose ADWIN as an implementation with theoretical guarantees in order to extend such guarantees to the resulting adaptive learning algorithm. A main advantage of our methods is that they require no guess about how fast or how often the stream will change; other methods typically have several user-defined parameters to this effect.

Introduction

We apply the framework presented in Chapter 7 to give two decision tree learning algorithms that can cope with concept and distribution drift on data streams: Hoeffding Window Trees in Section 9.2 and Hoeffding Adaptive Trees in Section 9.3. Decision trees are among the most common and well-studied classifier models. Classical methods such as C4.5 are not apt for data streams, as they assume all training data are available simultaneously in main memory, allowing for an unbounded number of passes, and certainly do not deal with data that changes over time. In the data stream context, a reference work on learning decision trees is the Hoeffding Tree for fast, incremental learning [52]. The Hoeffding Tree was described in Chapter 3. The methods we propose are based on VFDT, enriched with the change detection and estimation building blocks mentioned above.

We try several such building blocks, although the best suited for our purposes is the ADWIN algorithm, described in Chapter 8. This algorithm is parameter-free in that it automatically and continuously detects the rate of change in the data streams rather than using apriori guesses, thus allowing the client algorithm to react adaptively to the data stream it is processing. Additionally, ADWIN has rigorous guarantees of performance (Theorem 1 in Section 8.2.2). We show that these guarantees can be transferred to decision tree learners as follows: if a change is followed by a long enough stable period, the classification error of the learner will tend, and the same rate, to the error rate of VFDT.

Decision Trees on Sliding Windows

We propose a general method for building incrementally a decision tree based on a sliding window keeping the last instances on the stream. To specify one such method, we specify how to:

- place one or more change detectors at every node that will raise a hand whenever something worth attention happens at the node
- create, manage, switch and delete alternate trees
- maintain estimators of only relevant statistics at the nodes of the current sliding window

We call *Hoeffding Window Tree* any decision tree that uses Hoeffding bounds, maintains a sliding window of instances, and that can be included in this general framework. Figure 9.1 shows the pseudo-code of HOEFFDING WINDOW TREE. Note that δ' should be the *Bonferroni correction* of δ to account for the fact that many tests are performed and we want all of them to be simultaneously correct with probability $1-\delta$. It is enough e.g. to divide δ by the number of tests performed so far. The need for this correction is also acknowledged in [52], although in experiments the more convenient option of using a lower δ was taken. We have followed the same option in our experiments for fair comparison.

HWT-ADWIN : Hoeffding Window Tree using ADWIN

We use ADWIN to design HWT-ADWIN, a new Hoeffding Window Tree that uses ADWIN as a change detector. The main advantage of using a change detector as ADWIN is that it has theoretical guarantees, and we can extend this guarantees to the learning algorithms.

Example of performance Guarantee

Let HWT*ADWIN be a variation of HWT-ADWIN with the following condition: every time a node decides to create an alternate tree, an alternate tree is also started at the root. In this section we show an example of performance guarantee about the error rate of HWT*ADWIN. Informally speaking, it states that after a change followed by a stable period, HWT*ADWIN's error rate will decrease at the same rate as that of VFDT, after a transient period that depends only on the magnitude of the change.

We consider the following scenario: Let C and D be arbitrary concepts, that can differ both in example distribution and label assignments. Suppose the input data sequence S is generated according to concept C up to time t_0, that it abruptly changes to concept D at time t_0+1, and remains stable after that. Let HWT*ADWIN be run on sequence S, and e_1 be error(HWT*ADWIN,S,t_0), and e_2 be error(HWT*ADWIN,S,t_0+1), so that e_2-e_1 measures how much worse the error of HWT*ADWIN has become after the concept change.

```
HOEFFDING WINDOW TREE($Stream, \delta$)
 1  Let HT be a tree with a single leaf(root)
 2  Init estimators $A_{ijk}$ at root
 3  for each example $(x, y)$ in Stream
 4        do HWTREEGROW($(x, y), HT, \delta$)
```

```
HWTREEGROW($(x, y), HT, \delta$)
 1  Sort $(x, y)$ to leaf $l$ using $HT$
 2  Update estimators $A_{ijk}$
 3      at leaf $l$ and nodes traversed in the sort
 4  if current node $l$ has an alternate tree $T_{alt}$
 5      HWTREEGROW($(x, y), T_{alt}, \delta$)
 6  Compute $G$ for each attribute
    ▷ Evaluate condition for splitting leaf $l$
 7  if $G$(Best Attr.)$-G$(2nd best) $> \epsilon(\delta', \ldots)$
 8      then Split leaf on best attribute
 9  for each branch of the split
10          do Start new leaf
11                  and initialize estimators
12  if one change detector has detected change
13      then Create an alternate subtree $T_{alt}$ at leaf $l$ if there is none
14  if existing alternate tree $T_{alt}$ is more accurate
15      then replace current node $l$ with alternate tree $T_{alt}$
```

Figure 9.1: Hoeffding Window Tree algorithm

Here error(HWT*ADWIN,S,t) denotes the classification error of the tree kept by HWT*ADWIN at time t on S. Similarly, error(VFDT,D,t) denotes the expected error rate of the tree kept by VFDT after being fed with t random examples coming from concept D.

Theorem 4. *Let S, t_0, e_1, and e_2 be as described above, and suppose t_0 is sufficiently large w.r.t. $e_2 - e_1$. Then for every time $t > t_0$, we have*

$$\textit{error(HWT*}\texttt{ADWIN}, S, t) \leq \min\{ e_2, e_{VFDT} \}$$

with probability at least $1 - \delta$, where

- $e_{VFDT} = error(VFDT, D, t - t0 - g(e_2 - e_1)) + O(\frac{1}{\sqrt{t-t_0}})$
- $g(e_2 - e_1) = 8/(e_2 - e_1)^2 \ln(4t_0/\delta)$

The following corollary is a direct consequence, since $O(1/\sqrt{t - t_0})$ tends to 0 as t grows.

Corollary 1. *If error(VFDT,D,t) tends to some quantity $\epsilon \leq e_2$ as t tends to infinity, then error(HWT*ADWIN ,S,t) tends to ϵ too.*

Proof. We know by the ADWIN False negative rate bound that with probability $1 - \delta$, the ADWIN instance monitoring the error rate at the root shrinks at time $t_0 + n$ if

$$|e_2 - e_1| > 2\epsilon_{cut} = \sqrt{2/m \ln(4(t - t_0)/\delta)}$$

where m is the harmonic mean of the lengths of the subwindows corresponding to data before and after the change. This condition is equivalent to

$$m > 4/(e_1 - e_2)^2 \ln(4(t - t_0)/\delta)$$

If t_0 is sufficiently large w.r.t. the quantity on the right hand side, one can show that m is, say, less than $n/2$ by definition of the harmonic mean. Then some calculations show that for $n \geq g(e_2 - e_1)$ the condition is fulfilled, and therefore by time $t_0 + n$ ADWIN will detect change.

After that, HWT*ADWIN will start an alternative tree at the root. This tree will from then on grow as in VFDT, because HWT*ADWIN behaves as VFDT when there is no concept change. While it does not switch to the alternate tree, the error will remain at e_2. If at any time $t_0 + g(e_1 - e_2) + n$ the error of the alternate tree is sufficiently below e_2, with probability $1-\delta$ the two ADWIN instances at the root will signal this fact, and HWT*ADWIN will switch to the alternate tree, and hence the tree will behave as the one built by VFDT with t examples. It can be shown, again by using the False Negative Bound on ADWIN , that the switch will occur when the VFDT error goes below $e_2 - O(1/\sqrt{n})$, and the theorem follows after some calculation. □

CVFDT

As an extension of VFDT to deal with concept change Hulten, Spencer, and Domingos presented Concept-adapting Very Fast Decision Trees CVFDT [106] algorithm. We have presented it on Section 7.1. We review it here briefly and compare it to our method.

CVFDT works by keeping its model consistent with respect to a sliding window of data from the data stream, and creating and replacing alternate decision subtrees when it detects that the distribution of data is changing at a node. When new data arrives, CVFDT updates the sufficient statistics at its nodes by incrementing the counts n_{ijk} corresponding to the new examples and decrementing the counts n_{ijk} corresponding to the oldest example in the window, which is effectively forgotten. CVFDT is a Hoeffding Window Tree as it is included in the general method previously presented.

Two external differences among CVFDT and our method is that CVFDT has no theoretical guarantees (as far as we know), and that it uses a number of parameters, with default values that can be changed by the user - but which are fixed for a given execution. Besides the example window length, it needs:

1. T_0: after each T_0 examples, CVFDT traverses all the decision tree, and checks at each node if the splitting attribute is still the best. If there is a better splitting attribute, it starts growing an alternate tree rooted at this node, and it splits on the currently best attribute according to the statistics in the node.

2. T_1: after an alternate tree is created, the following T_1 examples are used to build the alternate tree.

3. T_2: after the arrival of T_1 examples, the following T_2 examples are used to test the accuracy of the alternate tree. If the alternate tree is more accurate than the current one, CVDFT replaces it with this alternate tree (we say that the alternate tree is promoted).

The default values are $T_0 = 10,000$, $T_1 = 9,000$, and $T_2 = 1,000$. One can interpret these figures as the preconception that often about the last $50,000$ examples are likely to be relevant, and that change is not likely to occur faster than every $10,000$ examples. These preconceptions may or may not be right for a given data source.

The main internal differences of HWT-ADWIN respect CVFDT are:

- The alternates trees are created as soon as change is detected, without having to wait that a fixed number of examples arrives after the change. Furthermore, the more abrupt the change is, the faster a new alternate tree will be created.

- HWT-ADWIN replaces the old trees by the new alternates trees as soon as there is evidence that they are more accurate, rather than having to wait for another fixed number of examples.

These two effects can be summarized saying that HWT-ADWIN adapts to the scale of time change in the data, rather than having to rely on the *a priori* guesses by the user.

Hoeffding Adaptive Trees

In this section we present Hoeffding Adaptive Tree as a new method that evolving from Hoeffding Window Tree, adaptively learn from data streams that change over time without needing a fixed size of sliding window. The optimal size of the sliding window is a very difficult parameter to guess for users, since it depends on the rate of change of the distribution of the dataset.

In order to avoid to choose a size parameter, we propose a new method for managing statistics at the nodes. The general idea is simple: we place instances of estimators of frequency statistics at every node, that is, replacing each n_{ijk} counters in the Hoeffding Window Tree with an instance A_{ijk} of an estimator.

More precisely, we present three variants of a *Hoeffding Adaptive Tree* or HAT, depending on the estimator used:

- HAT-INC: it uses a linear incremental estimator
- HAT-EWMA: it uses an Exponential Weight Moving Average (EWMA)
- HAT-ADWIN : it uses an ADWIN estimator. As the ADWIN instances are also change detectors, they will give an alarm when a change in the attribute-class statistics at that node is detected, which indicates also a possible concept change.

The main advantages of this new method over a Hoeffding Window Tree are:

- All relevant statistics from the examples are kept in the nodes. There is no need of an optimal size of sliding window for all nodes. Each node can decide which of the last instances are currently relevant for it. There is no need for an additional window to store current examples. For medium window sizes, this factor substantially reduces our memory consumption with respect to a Hoeffding Window Tree.
- A Hoeffding Window Tree, as CVFDT for example, stores in main memory only a bounded part of the window. The rest (most of it, for large window sizes) is stored in disk. For example, CVFDT has one parameter that indicates the amount of main memory used to store the window (default is 10,000). Hoeffding Adaptive Trees keeps all its data in main memory.

Example of performance Guarantee

In this subsection we show a performance guarantee on the error rate of HAT-ADWIN on a simple situation. Roughly speaking, it states that after a distribution and concept change in the data stream, followed by a stable period, HAT-ADWIN will start, in reasonable time, growing a tree identical to the one that

VFDT would grow if starting afresh from the new stable distribution. Statements for more complex scenarios are possible, including some with slow, gradual, changes.

Theorem 5. *Let D_0 and D_1 be two distributions on labelled examples. Let S be a data stream that contains examples following D_0 for a time T, then suddenly changes to using D_1. Let t be the time that until VFDT running on a (stable) stream with distribution D_1 takes to perform a split at the node. Assume also that VFDT on D_0 and D_1 builds trees that differ on the attribute tested at the root. Then with probability at least $1-\delta$:*

- *By time $t' = T + c \cdot V^2 \cdot t \log(tV)$, HAT-*`ADWIN` *will create at the root an alternate tree labelled with the same attribute as VFDT(D_1). Here $c \leq 20$ is an absolute constant, and V the number of values of the attributes.*[1]
- *this alternate tree will evolve from then on identically as does that of VFDT(D_1), and will eventually be promoted to be the current tree if and only if its error on D_1 is smaller than that of the tree built by time T.*

If the two trees do not differ at the roots, the corresponding statement can be made for a pair of deeper nodes.

Lemma 1. *In the situation above, at every time $t+T>T$, with probability $1-\delta$ we have at every node and for every counter (instance of* `ADWIN`*) $A_{i,j,k}$*

$$|A_{i,j,k} - P_{i,j,k}| \leq \sqrt{\frac{\ln(1/\delta')\,T}{t(t+T)}}$$

where $P_{i,j,k}$ is the probability that an example arriving at the node has value j in its ith attribute and class k.

Observe that for fixed δ' and T this bound tends to 0 as t grows.

To prove the theorem, use this lemma to prove high-confidence bounds on the estimation of $G(a)$ for all attributes at the root, and show that the attribute $best$ chosen by VFDT on D_1 will also have maximal $G(best)$ at some point, so it will be placed at the root of an alternate tree. Since this new alternate tree will be grown exclusively with fresh examples from D_1, it will evolve as a tree grown by VFDT on D_1.

Memory Complexity Analysis

Let us compare the memory complexity Hoeffding Adaptive Trees and Hoeffding Window Trees. We take CVFDT as an example of Hoeffding Window Tree. Denote with

- E : size of an example

[1]This value of t' is a very large overestimate, as indicated by our experiments. We are working on an improved analysis, and hope to be able to reduce t' to $T + c \cdot t$, for $c < 4$.

- A : number of attributes
- V : maximum number of values for an attribute
- C : number of classes
- T : number of nodes

A Hoeffding Window Tree as CVFDT uses memory $O(WE + TAVC)$, because it uses a window W with E examples, and each node in the tree uses AVC counters. A Hoeffding Adaptive Tree does not need to store a window of examples, but uses instead memory $O(\log W)$ at each node as it uses an `ADWIN` as a change detector, so its memory requirement is $O(TAVC + T\log W)$. For medium-size W, the $O(WE)$ in CVFDT can often dominate. HAT-`ADWIN` has a complexity of $O(TAVC\log W)$.

10
Adaptive Ensemble Methods

Online mining when data streams evolve over time, that is when concepts drift or change completely, is becoming one of the core issues of advanced analysis of data streams. When tackling non-stationary concepts, ensembles of classifiers have several advantages over single classifier methods: they are easy to scale and parallelize, they can adapt to change quickly by pruning under-performing parts of the ensemble, and they therefore usually also generate more accurate concept descriptions. This chapter presents two new variants of Bagging: `ADWIN` Bagging and Adaptive-Size Hoeffding Tree (ASHT) Bagging.

New method of Bagging using trees of different size

In this section, we introduce the Adaptive-Size Hoeffding Tree (ASHT). It is derived from the Hoeffding Tree algorithm with the following differences:

- it has a maximum number of split nodes, or *size*
- after one node splits, if the number of split nodes of the ASHT tree is higher than the maximum value, then it deletes some nodes to reduce its size

The intuition behind this method is as follows: smaller trees adapt more quickly to changes, and larger trees do better during periods with no or little

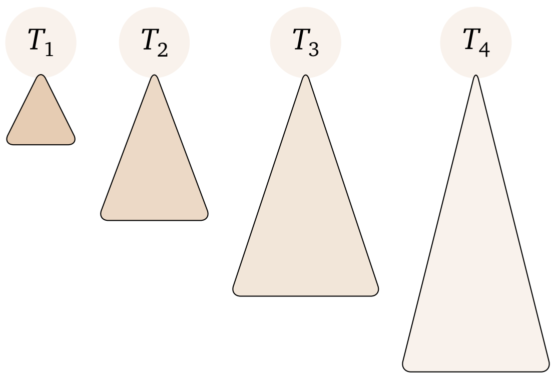
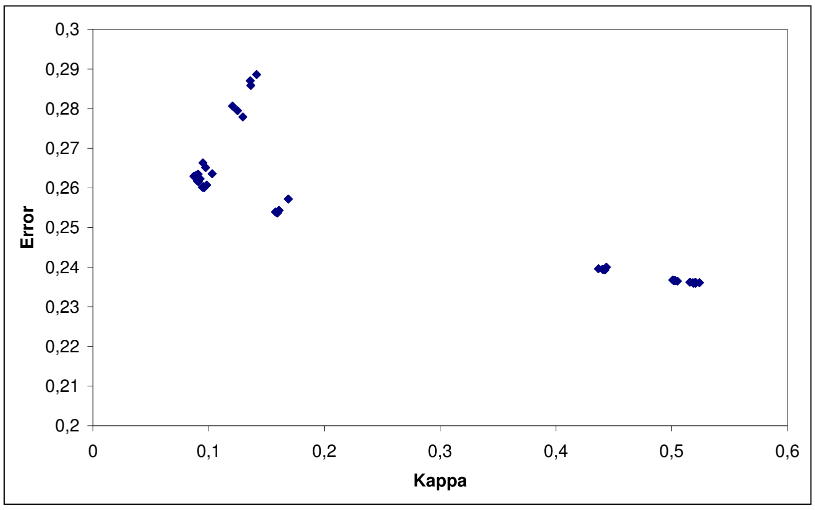


Figure 10.1: An ensemble of trees of different size



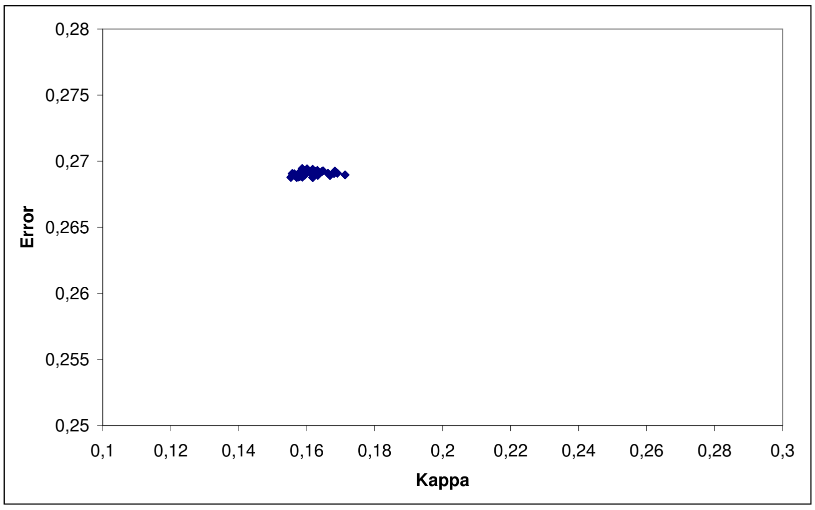


Figure 10.2: Kappa-Error diagrams for ASHT bagging (left) and bagging (right) on dataset RandomRBF with drift, plotting 90 pairs of classifiers.

change, simply because they were built on more data. Trees limited to size s will be reset about twice as often as trees with a size limit of $2s$. This creates a set of different reset-speeds for an ensemble of such trees, and therefore a subset of trees that are a good approximation for the current rate of change. It is important to note that resets will happen all the time, even for stationary datasets, but this behaviour should not have a negative impact on the ensemble's predictive performance.

When the tree size exceeds the maximun size value, there are two different delete options:

- delete the oldest node, the root, and all of its children except the one where the split has been made. After that, the root of the child not deleted becomes the new root delete the oldest node, the root, and all of its children.

- delete all the nodes of the tree, i.e., restart from a new root.

We present a new bagging method that uses these Adaptive-Size Hoeffding Trees and that sets the size for each tree (Figure 10.1). The maximum allowed size for the n-th ASHT tree is twice the maximum allowed size for the $(n-1)$-th tree. Moreover, each tree has a weight proportional to the inverse of the square of its error, and it monitors its error with an exponential weighted moving average (EWMA) with $\alpha = .01$. The size of the first tree is 2.

With this new method, we attempt to improve bagging performance by increasing tree diversity. It has been observed that boosting tends to produce a more diverse set of classifiers than bagging, and this has been cited as a factor in increased performance [140].

We use the Kappa statistic κ to show how using trees of different size, we increase the diversity of the ensemble. Let's consider two classifiers h_a and h_b, a data set containing m examples, and a contingency table where cell C_{ij} contains the number of examples for which $h_a(x) = i$ and $h_b(x) = j$. If h_a and h_b are identical on the data set, then all non-zero counts will appear along the diagonal. If h_a and h_b are very different, then there should be a large number of counts off the diagonal. We define

$$\Theta_1 = \frac{\sum_{i=1}^{L} C_{ii}}{m}$$

$$\Theta_2 = \sum_{i=1}^{L} \left(\sum_{j=1}^{L} \frac{C_{ij}}{m} \cdot \sum_{j=1}^{L} \frac{C_{ji}}{m} \right)$$

We could use Θ_1 as a measure of agreement, but in problems where one class is much more common than others, all classifiers will agree by chance, so all pair of classifiers will obtain high values for Θ_1. To correct this, the κ statistic is defined as follows:

$$\kappa = \frac{\Theta_1 - \Theta_2}{1 - \Theta_2}$$

κ uses Θ_2, the probability that two classifiers agree by chance, given the observed counts in the table. If two classifiers agree on every example then $\kappa = 1$, and if their predictions coincide purely by chance, then $\kappa = 0$.

We use the Kappa-Error diagram to compare the diversity of normal bagging with bagging using trees of different size. The Kappa-Error diagram is a scatterplot where each point corresponds to a pair of classifiers. The x coordinate of the pair is the κ value for the two classifiers. The y coordinate is the average of the error rates of the two classifiers.

Figure 10.2 shows the Kappa-Error diagram for the Random RBF dataset with drift parameter or change speed equal to 0.001.We observe that bagging classifiers are very similar to one another and that the decision tree classifiers of different size are very diferent from one another.

New method of Bagging using `ADWIN`

`ADWIN`[18], as seen in Chapter 8 is a change detector and estimator that solves in a well-specified way the problem of tracking the average of a stream of bits or real-valued numbers. `ADWIN` keeps a variable-length window of recently seen items, with the property that the window has the maximal length statistically consistent with the hypothesis “there has been no change in the average value inside the window".

More precisely, an older fragment of the window is dropped if and only if there is enough evidence that its average value differs from that of the rest of the window. This has two consequences: one, that change reliably declared whenever the window shrinks; and two, that at any time the average over the existing window can be reliably taken as an estimation of the current average in the stream (barring a very small or very recent change that is still not statistically visible).

`ADWIN` is parameter- and assumption-free in the sense that it automatically detects and adapts to the current rate of change. Its only parameter is a confidence bound δ, indicating how confident we want to be in the algorithm's output, inherent to all algorithms dealing with random processes.

Also important for our purposes, `ADWIN` does not maintain the window explicitly, but compresses it using a variant of the exponential histogram technique in [11]. This means that it keeps a window of length W using only $O(\log W)$ memory and $O(\log W)$ processing time per item, rather than the $O(W)$ one expects from a naïve implementation.

`ADWIN` Bagging is the online bagging method implemented in MOA with the addition of the `ADWIN` algorithm as a change detector and as an estimator for the weights of the boosting method. When a change is detected, the worst classifier of the ensemble of classifiers is removed and a new classifier is added to the ensemble.

Adaptive Hoeffding Option Trees

Hoeffding Option Trees [158] explained in Section 6.2.3 are regular Hoeffding trees containing additional option nodes that allow several tests to be applied, leading to multiple Hoeffding trees as separate paths. They consist of a single structure that efficiently represents multiple trees. A particular example can travel down multiple paths of the tree, contributing, in different ways, to different options.

An *Adaptive Hoeffding Option Tree* is a Hoeffding Option Tree with the following improvement: each leaf stores an estimation of the current error. It uses an EWMA estimator with $\alpha = .2$. The weight of each node in the voting process is proportional to the square of the inverse of the error.

Method performance

Bagging is clearly the best method in terms of accuracy. This superior position is, however, achieved at high cost in terms of memory and time. `ADWIN` Bagging and ASHT Bagging are the most accurate methods for most datasets, but they are slow. `ADWIN` Bagging is slower than ASHT Bagging and for some datasets it needs more memory. ASHT Bagging using weighted classifiers and replacing oversized trees with new ones seems to be the most accurate ASHT bagging method. We observe that bagging using 5 trees of different size may be sufficient, as its error is not much higher than for 10 trees, but it is nearly twice as fast. Also Hoeffding trees using drift detection methods are faster but less accurate methods.

In [158], a range of option limits were tested and averaged across all datasets without concept drift to determine the optimal number of paths. This optimal number of options was five. Dealing with concept drift, we observe that increasing the number of options to 50, a significant improvement is obtained in accuracy for some datasets.

11 Restricted Hoeffding Trees Stacking

When applying boosting algorithms to build ensembles of decision trees using a standard tree induction algorithm, the tree inducer has access to all the attributes in the data. Thus, each tree can model arbitrarily complex interactions between attributes in principle. However, often the full modeling power of unrestricted decision tree learners is not necessary to yield good accuracy with boosting, and it may even be harmful because it can lead to overfitting. Hence, it is common to apply boosting in conjunction with depth-limited decision trees, where the growth of each tree is restricted to a certain level. In this way, only a limited set of attributes can be used in each tree, but the risk of overfitting is reduced. In the extreme case, only a single split is allowed and the decision trees degenerate to decision stumps. Despite the fact that each restricted tree can only model interactions between a limited set of attributes, the overall ensemble is often highly accurate.

The method presented in this chapter is based on this observation [17]. We present an algorithm that produces a classification model based on an ensemble of restricted decision trees, where each tree is built from a distinct subset of the attributes. The overall model is formed by combining the log-odds of the predicted class probabilities of these trees using sigmoid perceptrons, with one perceptron per class. In contrast to the standard boosting approach, which forms an ensemble classifier in a greedy fashion, building each tree in sequence and assigning corresponding weights as a by-product, our method generates each tree in parallel and combines them using perceptron classifiers by adopting the stacking approach [193]. Because we are working in a data stream scenario, Hoeffding trees [52] are used as the ensemble members. They, as well as the perceptrons, can be trained incrementally, and we also show how ADWIN-based change detection [18] can be used to apply the method to evolving data streams.

There is existing work on boosting for data streams [154], but the algorithm has been found to yield inferior accuracy compared to ADWIN bagging. Moreover, it is unclear how a boosting algorithm can be adapted to data streams that evolve over time: because bagging generates models independently, a model can be replaced when it is no longer accurate, but the sequential nature of boosting prohibits this simple and elegant solution. Because our method generates models independently, we can apply this simple strategy, and we show that it yields more accurate classifications than Online Bagging [154] on the real-world and artificial data streams that we consider in our experiments.

Our method is also related to work on building ensembles using random subspace models [101]. In the random subspace approach, each model in an ensemble is trained based on a randomly chosen subset of attributes. The models' predictions are then combined in an unweighted fashion. Because of this latter property, the number of attributes available to each model must necessarily be quite large in general, so that each individual model is powerful enough to yield accurate classifications. In contrast, in our method, we exhaustively consider all subsets of a given, small, size, build a tree for each subset of attributes, and then combine their predictions using stacking. In random forests [28], the random subspace approach is applied locally at each node of a decision tree in a bagged ensemble. Random forests have been applied to data streams, but did not yield substantial improvements on bagging in this scenario.

Combining Restricted Hoeffding Trees using Stacking

The basic method we present in this chapter is very simple: enumerate all attribute subsets of a given user-specified size k, learn a Hoeffding tree from each subset based on the incoming data stream, gather the trees' predictions for each incoming instance, and use these predictions to train simple perceptron classifiers. The question that needs to be addressed is how exactly to prepare the "meta" level data for the perceptrons and what shape it should take.

Rather than using discrete classifications to build the meta level model in stacking, it is common to use class probability estimates instead because they provide more information due to the fact that they represent the degree of confidence that each model has in its predictions. Hence, we also adopt this approach in our method: the meta-level data is formed by collecting the class probability estimates for an incoming new instance, obtained from the Hoeffding trees built from the data observed previously.

The meta level combiner we use is based on simple perceptrons with sigmoid activation functions, trained using stochastic gradient descent to minimize the squared loss with respect to the actual observed class labels in the data stream. We train one perceptron per class value and use the Hoeffding trees' class probability estimates for the corresponding class value to form the input data for each perceptron.

There is one caveat. Because the sigmoid activation function is used in the perceptrons, we do not use the raw class probability estimates as the input values for training them. Rather, we use the log-odds of these probability estimates instead. Let $\hat{p}(c_{ij}|\vec{x})$ be the probability estimate for class i and instance $\vec{x}$ obtained from Hoeffding tree j in the ensemble. Then we use

$$a_{ij} = \log(\hat{p}(c_{ij}|\vec{x})/(1-\hat{p}(c_{ij}|\vec{x}))$$

as the value of input attribute j for the perceptron associated with class value i.

Let $\vec{a}_i$ be the vector of log-odds for class i. The output of the sigmoid perceptron for class i is then $f(\vec{a}_i) = 1/(1+e^{-(\vec{w}_i\vec{a}_i+b_i)})$, based on per-class coefficients $\vec{w}_i$ and b_i. We use the log-odds as the inputs for the perceptron because application of the sigmoid function presupposes a linear relationship between $\log(f(\vec{a}_i)/(1-f(\vec{a}_i)))$ and $\vec{a}_i$.

To avoid the zero-frequency problem, we slightly modify the probability estimates obtained from a Hoeffding tree by adding a small constant ϵ to the probability for each class, and then renormalize. In our experiments, we use $\epsilon = 0.001$, but smaller values lead to very similar results.

The perceptrons are trained using stochastic gradient descent: the weight vector is updated each time a new training instance is obtained from the data stream. Once the class probability estimates for that instance have been obtained from the ensemble of Hoeffding trees, the input data for the perceptrons can be formed, and the gradient descent update rule can be used to perform the update. The weight values are initialized to the reciprocal of the size of the ensemble, so that the perceptrons give equal weight to each ensemble member initially.

A crucial aspect of stochastic gradient descent is an appropriately chosen learning rate, which determines the magnitude of the update. If it is chosen too large, there is a risk that the learning process will not converge. A common strategy is to decrease it as the amount of training data increases. Let n be the number of training instances seen so far in the data stream, and let m be the number of attributes. We set the learning rate α based on the following equation:

$$\alpha = \frac{2}{2+m+n}.$$

However, there is a problem with this approach for setting the learning rate in the context we consider here: it assumes that the training data is identically distributed. This is not actually the case in our scenario because the training data for the perceptrons is derived from the probability estimates obtained from the Hoeffding trees, and these change over time, generally becoming more accurate. Setting the learning rate based on the above equation means that the perceptrons will adapt too slowly once the initial data in the data stream has been processed.

There is a solution to this problem: the stream of predictions from the Hoeffding trees can be viewed as an *evolving* data stream (regardless of whether the underlying data stream forming the training data for the Hoeffding trees is actually evolving) and we can use an existing change detection method for evolving data streams to detect when the learning rate should be reset to a larger value. We do this very simply by setting the value of n to zero when change has been detected. To detect change, we use the ADWIN change detector [18], which is discussed in more detail in the next section. It detects when the accuracy of a classifier increases or decreases significantly as a data stream is processed. We apply it to monitor the accuracy of each Hoeffding tree in the ensemble. When accuracy changes significantly for one of the trees, the learning rate is reset by setting n to zero. The value of n is then incremented for each new instance in the stream until a new change is detected. This has the effect that the learning rate will be kept relatively large while the learning curve for the Hoeffding trees has a significant upward trend.

ADWIN-based Change Detection

The ADWIN change detector comes with nice theoretical guarantees. In addition to using it to reset the learning rate when necessary, we also use it to make our ensemble classifier applicable to evolving data stream, where the original data stream, used to train the Hoeffding trees, changes over time.

The strategy we use to cope with evolving data streams using ADWIN is very simple: the idea is to replace ensemble members when they start to perform poorly. To implement this, we use ADWIN to detect when the accuracy of one of the Hoeffding trees in the ensemble has dropped significantly. To do this, we can use the same ADWIN change detectors that are also applied to detect when the learning rate needs to be reset. When one of the change detectors associated with a particular tree reports a significant drop in accuracy, the tree is reset and the coefficients in the perceptrons that are associated with this tree (one per class value) are set to zero. A new tree is then generated from new data in the data stream, so that the ensemble can adapt to changes.

Note that all trees for which a significant drop is detected are replaced. Note also that a change detection event automatically triggers a reset of the learning rate, which is important because the perceptrons need to adapt to the changed ensemble of trees.

A Note on Computational Complexity

This new approach is based on generating trees for all possible attribute subsets of size k. If there are m attributes in total, there are $\binom{m}{k}$ of these subsets. Clearly, only moderate values of k, or values of k that are very close to m, are feasible. When k is one, there is no penalty with respect to computational complexity compared to building a single Hoeffding tree, because then there is only one tree per attribute, and an unrestricted tree also scales linearly in the number of attributes. If k is two, there is an extra factor of m in the computational complexity compared to building a single unrestricted Hoeffding tree, i.e. overall effort becomes quadratic in the number of attributes.

$k = 2$ is very practical even for datasets with a relatively large number of attributes, although certainly not for very high-dimensional data (for which linear classifiers are usually sufficient anyway). Larger values of k are only practical for small numbers of attributes, unless k is very close to m (e.g. $k = m - 1$). We have used $k = 4$ for datasets with 10 attributes in our experiments, with very acceptable runtimes. It is important to keep in mind that many practical classification problems appear to exhibit only very low-dimensional interactions, which means small values of k are sufficient to yield high accuracy.

12 Leveraging Bagging

Bagging and Boosting are ensemble methods used to improve the accuracy of classifier methods. Non-streaming bagging [23] builds a set of M base models, training each model with a bootstrap sample of size N created by drawing random samples with replacement from the original training set. Each base model's training set contains each of the original training example K times where $P(K = k)$ follows a binomial distribution. This binomial distribution for large values of N tends to a Poisson(1) distribution, where Poisson(1)= $\exp(-1)/k!$. Using this fact, Oza and Russell [154, 153] proposed *Online Bagging*, an online method that instead of sampling with replacement, gives each example a weight according to Poisson(1).

Boosting algorithms combine multiple base models to obtain a small generalization error. Non-streaming boosting builds a set of models sequentially, with the construction of each new model depending on the performance of the previously constructed models. The intuitive idea of boosting is to give more weight to misclassified examples, and reducing the weight of the correctly classified ones.

From studies appearing in the literature [154, 153, 20], Online Bagging seems to perform better than online boosting methods. Why bagging outperforms boosting in the data stream setting is still an open question. Adding more random weight to all instances seems to improve accuracy more than adding weight to misclassified instances. In this chapter we focus on randomization as a powerful tool to increase accuracy and diversity when constructing an ensemble of classifiers. There are three ways of using randomization:

- Manipulating the input data
- Manipulating the classifier algorithms
- Manipulating the output targets

In this chapter we focus on randomizing the input data and the output prediction of online bagging.

Bagging Literature

Breiman [23] introduced bagging classification using the notion of an *order-correct* learner. An order-correct learner ϕ at the input x is a predictor that if input x results in a class more often than any other class, then ϕ will also predict this class at x more often than any other class. An order-correct learner is not necessarily an accurate predictor but its aggregated predictor is optimal. If a predictor is good because it is order-correct for most inputs x then aggregation can transform it into a nearly optimal predictor. The vital element to gain accuracy is the instability of the prediction method. A learner is unstable if a small change in the input data leads to large changes in the output.

Friedman [69] explained that bagging works by reducing variance without changing the bias. There are several definitions of bias and variance for classification, but the common idea is that bias measures average error over many different training sets, and variance measures the additional error due to the variation in the model produced by using different training sets.

Domingos [51] claimed that Breiman's line of reasoning is limited, since we may never know *a priori* whether a learner is order-correct for a given example or not, and what regions of the instance space will be order-correct or not. He explained bagging's success showing that bagging works by effectively changing a single-model learner to another single-model learner, with a different implicit prior distribution over models, one that is less biased in favor of simple models.

Some work in the literature shows that bagging asymptotically performs some smoothing on the estimate. Friedman and Hall [67] used an asymptotic truncated Taylor series of the estimate to show that in the limit of infinite samples, bagging reduces the variance of non-linear components.

Bühlmann and Yu [31], analyzed bagging using also asymptotic limiting distributions, and they proposed *subagging* as a less expensive alternative to bagging. Subagging uses subsampling as an alternative aggregation scheme. They claimed that subagging is as accurate as bagging but uses less computation.

Grandvalet[84] explained the performance of bagging by the goodness and badness of highly influential examples, in situations where the usual variance reduction argument is questionable. He presented an experiment showing that bagging increases the variance of decision trees, and claimed that bagging does not simply reduce variance in its averaging process.

In Chapter 10 two new state-of-the-art bagging methods were presented: ASHT Bagging using trees of different sizes, and `ADWIN` Bagging using a change detector to decide when to discard underperforming ensemble members.

Breiman [28] proposed Random Forests as a method to use randomization on the input and on the internal construction of the decision trees. Random Forests are ensembles of trees with the following characteristics: the input training set is obtained by sampling with replacement, the nodes of the tree only may use a fixed number of random attributes to split, and the trees are grown without pruning. Abdulsalam et al. [2] presented a streaming version of random forests and Saffari et al. [166] presented an online version.

Leveraging Bagging

In this section, we present a new online leveraging bagging algorithm, improving Online Bagging of Oza and Russell. The pseudo-code of Online Bagging of Oza and Russell is listed in Algorithm 9.

We leverage the performance of bagging, with two randomization improvements: increasing resampling and using output detection codes.

Algorithm 13 *Leveraging Bagging* for M models

```
1: Initialize base models $h_m$ for all $m \in \{1, 2, ..., M\}$
2: Compute coloring $\mu_m(y)$
3: for all training examples $(x, y)$ do
4:    for $m = 1, 2, ..., M$ do
5:       Set $w = Poisson(\lambda)$
6:       Update $h_m$ with the current example with weight $w$ and class $\mu_m(y)$
7:    end for
8: end for
9: if ADWIN detects change in error of one of the classifiers then
10:    Replace classifier with higher error with a new one
11: end if
12: anytime output:
13: return hypothesis: $h_{fin}(x) = \arg\max_{y \in Y} \sum_{t=1}^{T} I(h_t(x) = \mu_t(y))$
```

Resampling with replacement is done in Online Bagging using Poisson(1). There are other sampling mechanisms:

- Lee and Clyde [134] uses the Gamma distribution (Gamma(1,1)) to obtain a Bayesian version of Bagging. Note that Gamma(1,1) is equal to Exp(1).
- Bulhman and Yu [31] proposes subagging, using resampling without replacement.

Our proposal is to increase the weights of this resampling using a larger value λ to compute the value of the Poisson distribution. The Poisson distribution is used to model the number of events occurring within a given time interval.

Figure 12.1 shows the probability function mass of the distribution of Poisson for several values of λ. The mean and variance of a Poisson distribution is λ. For $\lambda = 1$ we see that 37% of the values are zero, 37% are one, and 26% are values greater than one. Using a weight of Poisson(1) we are taking out 37% of the examples, and repeating 26% of the examples, in a similar way to non streaming bagging. For $\lambda = 6$ we see that 0.25% of the values are zero, 45% are lower than six, 16% are six, and 39% are values greater than six. Using a value of $\lambda > 1$ for Poisson(λ) we are increasing the diversity of the weights and modifying the input space of the classifiers inside the ensemble. However, the optimal value of λ may be different for each dataset.

Our second improvement is to add randomization at the output of the ensemble using output codes. Dietterich and Bakiri [49] introduced a method based

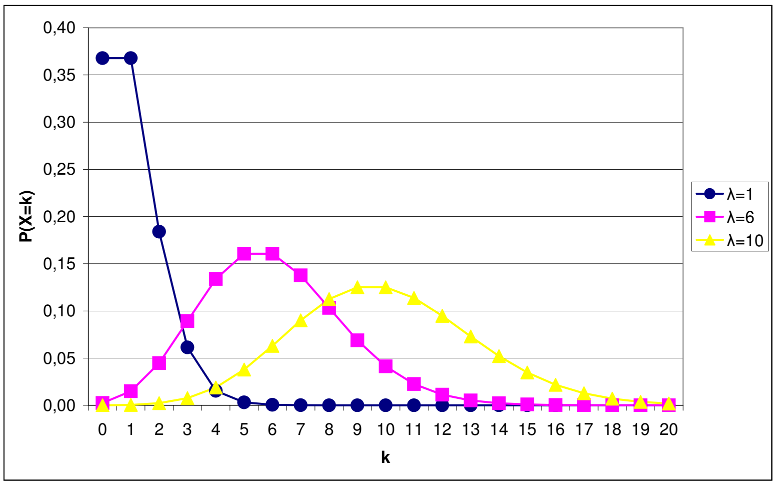


Figure 12.1: Poisson Distribution.

on error-correcting output codes, which handles multiclass problems using only a binary classifier. The classes assigned to each example are modified to create a new binary classification of the data induced by a mapping from the set of classes to {0,1}. A variation of this method by Schapire [168] presented a form of boosting using output codes.

We assign to each class a binary string of length n and then build an ensemble of n binary classifiers. Each of the classifiers learns one bit for each position in this binary string. When a new instance arrives, we assign x to the class whose binary code is closest. We can view an error-correcting code as a form of voting in which a number of incorrect votes can be corrected.

We use random output codes instead of deterministic codes. In standard ensemble methods, all classifiers try to predict the same function. However, using output codes each classifier will predict a different function. This may reduce the effects of correlations between the classifiers, and increase diversity of the ensemble.

We implement random output codes in the following way: we choose for each classifier m and class c a binary value $\mu_m(c)$ in a uniform, independent, and random way. We ensure that exactly half of the classes are mapped to 0. The output of the classifier for an example is the class which has more votes of its binary mapping classes. Table 12.1 shows an example for an ensemble of 6 classifiers in a classification task of 3 classes.

We use the same strategy as in Chapter 10. ADWIN [18] is a change detector and estimator that solves in a well-specified way the problem of tracking the average of a stream of bits or real-valued numbers. ADWIN keeps a variable-length window of recently seen items, with the property that the window has

Table 12.1: Example matrix of random output codes for 3 classes and 6 classifiers

	Class 1	Class 2	Class 3
Classifier 1	0	0	1
Classifier 2	0	1	1
Classifier 3	1	0	0
Classifier 4	1	1	0
Classifier 5	1	0	1
Classifier 6	0	1	0

the maximal length statistically consistent with the hypothesis "there has been no change in the average value inside the window".

ADWIN is parameter- and assumption-free in the sense that it automatically detects and adapts to the current rate of change. Its only parameter is a confidence bound δ, indicating how confident we want to be in the algorithm's output, inherent to all algorithms dealing with random processes.

Also important for our purposes, ADWIN does not maintain the window explicitly, but compresses it using a variant of the exponential histogram technique. This means that it keeps a window of length W using only $O(\log W)$ memory and $O(\log W)$ processing time per item.

Algorithm 13 shows the pseudo-code of our Leveraging Bagging. First we build a matrix with the values of μ for each classifier and class. For each new instance that arrives, we give it a random weight of $Poisson(k)$. We train the classifier with this weight, and when a change is detected, the worst classifier of the ensemble of classifiers is removed and a new classifier is added to the ensemble. To predict the class of an example, we compute for each class c the sum of the votes for $\mu(c)$ of all the ensemble classifiers, and we output as a prediction the class with the most votes.

13 Twitter Stream Mining

Twitter is a "what's-happening-right-now" tool that enables interested parties to follow individual users' thoughts and commentary on events in their lives—in almost real-time [173]. It is a potentially valuable source of data that can be used to delve into the thoughts of millions of people as they are uttering them. Twitter makes these utterances immediately available in a data stream, which can be mined using appropriate stream mining techniques. In principle, this could make it possible to infer people's opinions, both at an individual level as well as in aggregate, regarding potentially any subject or event [173].

At the official Twitter Chirp developer conference in April 2010 [194], the company presented some statistics about its site and its users. In April 2010, Twitter had 106 million registered users, and 180 million unique visitors every month. New users were signing up at a rate of 300,000 per day. Twitter's search engine received around 600 million search queries per day, and Twitter received a total of 3 billion requests a day via its API. Of Twitter's active users, 37 percent used their phone to tweet.

Twitter data follows the data stream model. In this model, data arrive at high speed, and algorithms that process them must be able to predict in real time and do so under very strict constraints of space and time. Data streams pose several challenges for algorithm design. First, they must make use of limited resources (time and memory). Second, they must deal with data whose nature or distribution changes over time.

The main Twitter data stream that provides all messages from every user in real-time is called Firehose [116] and was made available to developers in 2010. To deal with this large amount of data, and to use it for sentiment analysis and opinion mining—the task considered in this chapter—streaming techniques are needed. However, to the best of our knowledge, data stream algorithms, in conjunction with appropriate evaluation techniques, have so far not been considered for this task.

Evaluating data streams in real time is a challenging task. Most work in the literature considers only how to build a picture of accuracy over time. Two main approaches arise [19]:

- **Holdout**: Performance is measured using on single hold-out set.
- **Interleaved Test-Then-Train or Prequential**: Each individual example is used to test the model before it is used for training, and accuracy is incrementally updated.

A common problem is that for unbalanced data streams with, for example, 90% of the instances in one class, the simplest classifiers will have high accuracies of at least 90%. To deal with this type of data stream, we propose the Kappa statistic, based on a sliding window, as a measure for classifier performance in unbalanced class streams [16].

Mining Twitter Data: Challenges

Twitter has its own conventions that renders it distinct from other textual data. Consider the following Twitter example message ("tweet"): `RT @toni has a cool #job`. It shows that users may reply to other users by indicating user names using the character @, as in, for example, `@toni`. Hashtags (#) are used to denote subjects or categories, as in, for example `#job`. `RT` is used at the beginning of the tweet to indicate that the message is a so-called "retweet", a repetition or reposting of a previous tweet.

In the knowledge discovery context, there are two fundamental data mining tasks that can be considered in conjunction with Twitter data: (a) graph mining based on analysis of the links amongst messages, and (b) text mining based on analysis of the messages' actual text.

Twitter graph mining has been used to tackle several interesting problems:

- **Measuring user influence and dynamics of popularity**. Direct links indicate the flow of information, and thus a user's influence on others. There are three measures of influence: indegree, retweets and mentions. Cha et al. [36] show that popular users who have high indegree are not necessarily influential in terms of retweets or mentions, and that influence is gained through concerted effort such as limiting tweets to a single topic.

- **Community discovery and formation**. Java et al. [114] found communities using HyperText Induced Topic Search (HITS) [124], and the Clique Percolation Method [44]. Romero and Kleinberg [165] analyze the formation of links in Twitter via the directed closure process.

- **Social information diffusion**. De Choudhury et al. [43] study how data sampling strategies impact the discovery of information diffusion.

There are also a number of interesting tasks that have been tackled using Twitter text mining: sentiment analysis, which is the application we consider in this chapter, classification of tweets into categories, clustering of tweets and trending topic detection.

Considering sentiment analysis [137, 157], O'Connor et al. [152] found that surveys of consumer confidence and political opinion correlate with sentiment word frequencies in tweets, and propose text stream mining as a substitute for traditional polling. Jansen et al. [113] discuss the implications for organizations of using micro-blogging as part of their marketing strategy. Pak et al. [156] used classification based on the multinomial naïve Bayes classifier for sentiment analysis. Go et al. [83] compared multinomial naïve Bayes, a maximum entropy classifier, and a linear support vector machine; they all exhibited broadly comparable accuracy on their test data, but small differences could be observed depending on the features used.

The Twitter Streaming API

The Twitter Application Programming Interface (API) [1] currently provides a Streaming API and two discrete REST APIs. The Streaming API [116] provides real-time access to Tweets in sampled and filtered form. The API is HTTP based, and GET, POST, and DELETE requests can be used to access the data.

In Twitter terminology, individual messages describe the "status" of a user. The streaming API allows near real-time access to subsets of public status descriptions, including replies and mentions created by public accounts. Status descriptions created by protected accounts and all direct messages are not available. An interesting property of the streaming API is that it can filter status descriptions using quality metrics, which are influenced by frequent and repetitious status updates, etc.

The API uses basic HTTP authentication and requires a valid Twitter account. Data can be retrieved as XML and the more succinct JSON format. Parsing JSON data from the streaming API is simple: every object is returned on its own line, and ends with a carriage return.

Twitter Sentiment Analysis

Sentiment analysis can be cast as a classification problem where the task is to classify messages into two categories depending on whether they convey positive or negative feelings.

Twitter sentiment analysis is not an easy task because a tweet can contain a significant amount of information in very compressed form, and simultaneously carry positive and negative feelings. Consider the following example:

```
I currently use the Nikon D90 and love it, but not as
much as the Canon 40D/50D. I chose the D90 for the video
feature.  My mistake.
```

Also, some tweets may contain sarcasm or irony [35] as in the following example:

```
After a whole 5 hours away from work, I get to go back
again, I'm so lucky!
```

To build classifiers for sentiment analysis, we need to collect training data so that we can apply appropriate learning algorithms. Labeling tweets manually as positive or negative is a laborious and expensive, if not impossible, task. However, a significant advantage of Twitter data is that many tweets have author-provided sentiment indicators: changing sentiment is implicit in the use of various types of emoticons. Hence we may use these to label our training data.

Smileys or *emoticons* are visual cues that are associated with emotional states [162, 35]. They are constructed using the characters available on a standard keyboard, representing a facial expression of emotion. When the author of a tweet uses an emoticon, they are annotating their own text with an emotional state. Annotated tweets can be used to train a sentiment classifier.

Streaming Data Evaluation with Unbalanced Classes

In data stream mining, the most frequently used measure for evaluating predictive accuracy of a classifier is prequential accuracy [79]. We argue that this measure is only appropriate when all classes are balanced, and have (approximately) the same number of examples. In this section, we propose the Kappa statistic as a more sensitive measure for quantifying the predictive performance of streaming classifiers. For example, considering the particular target domain in this chapter, the rate in which the Twitter Streaming API delivers positive or negative tweets may vary over time; we cannot expect it to be 50% all the time. Hence, a measure that automatically compensates for changes in the class distribution should be preferable.

Just like accuracy, Kappa needs to be estimated using some sampling procedure. Standard estimation procedures for small datasets, such as cross-validation, do not apply. In the case of very large datasets or data streams, there are two basic evaluation procedures: holdout evaluation and prequential evaluation. Only the latter provides a picture of performance over time. In prequential evaluation (also known as interleaved test-then-train evaluation), each example in a data stream is used for testing before it is used for training.

We argue that prequential accuracy is not well-suited for data streams with unbalanced data, and that a prequential estimate of Kappa should be used instead. Let p_0 be the classifier's prequential accuracy, and p_c the probability that a chance classifier—one that assigns the same number of examples to each class as the classifier under consideration—makes a correct prediction. Consider the simple confusion matrix shown in Table 13.1. From this table, we see that Class+ is predicted correctly 75 out of 100 times, and Class- is predicted correctly 10 times. So accuracy p_0 is 85%. However a classifier predicting solely by chance—in the given proportions—will predict Class+ and Class- correctly in 68.06% and 3.06% of cases respectively. Hence, it will have an accuracy p_c of 71.12% as shown in Table 13.2.

Comparing the classifier's observed accuracy to that of a chance predictor renders its performance far less impressive than it first seems. The problem is that one class is much more frequent than the other in this example and plain accuracy does not compensate for this. The Kappa statistic, which normalizes a classifier's accuracy by that of a chance predictor, is more appropriate in scenarios such as this one.

The Kappa statistic κ was introduced by Cohen [40]. We argue that it is particularly appropriate in data stream mining due to potential changes in the class distribution. Consider a classifier h, a data set containing m examples and L

	Predicted Class+	Predicted Class-	Total
Correct Class+	75	8	83
Correct Class-	7	10	17
Total	82	18	100

Table 13.1: Simple confusion matrix example

	Predicted Class+	Predicted Class-	Total
Correct Class+	68.06	14.94	83
Correct Class-	13.94	3.06	17
Total	82	18	100

Table 13.2: Confusion matrix for chance predictor based on example in Table 13.1

classes, and a contingency table where cell C_{ij} contains the number of examples for which $h(x) = i$ and the class is j. If $h(x)$ correctly predicts all the data, then all non-zero counts will appear along the diagonal. If h misclassifies some examples, then some off-diagonal elements will be non-zero.

We define

$$p_0 = \frac{\sum_{i=1}^{L} C_{ii}}{m}$$

$$p_c = \sum_{i=1}^{L} \left(\sum_{j=1}^{L} \frac{C_{ij}}{m} \cdot \sum_{j=1}^{L} \frac{C_{ji}}{m} \right)$$

In problems where one class is much more common than the others, any classifier can easily yield a correct prediction by chance, and it will hence obtain a high value for p_0. To correct for this, the κ statistic is defined as follows:

$$\kappa = \frac{p_0 - p_c}{1 - p_c}$$

If the classifier is always correct then $\kappa = 1$. If its predictions coincide with the correct ones as often as those of the chance classifier, then $\kappa = 0$.

The question remains as to how exactly to compute the relevant counts for the contingency table: using all examples seen so far is not useful in time-changing data streams. Gama et al. [79] propose to use a forgetting mechanism for estimating prequential accuracy: a sliding window of size w with the most recent observations, or fading factors that weigh observations using a decay factor α. As the output of the two mechanisms is very similar (every window of size w_0 may be approximated by some decay factor α_0), we propose to use the Kappa statistic measured using a sliding window. Note that, to calculate the statistic for an n_c class problem, we need to maintain only $2n_c + 1$ estimators. We store the sum of all rows and columns in the confusion matrix ($2n_c$ values) to compute p_c, and we store the prequential accuracy p_0. The ability to calculate it efficiently is an important reason why the Kappa statistic is more appropriate for data streams than a measure such as the area under the ROC curve.

Data Stream Mining Methods

There are three fast incremental methods that are well-suited to deal with text data streams: multinomial naïve Bayes, stochastic gradient descent, and the Hoeffding tree.

Multinomial Naïve Bayes

The multinomial naïve Bayes classifier is a popular classifier for document classification that often yields good performance. It can be trivially applied to data streams because it is straightforward to update the counts required to estimate conditional probabilities..

Multinomial naive Bayes considers a document as a bag-of-words. For each class c, $P(w|c)$, the probability of observing word w given this class, is estimated from the training data, simply by computing the relative frequency of each word in the collection of training documents of that class. The classifier also requires the prior probability $P(c)$, which is straightforward to estimate.

Assuming n_{wd} is the number of times word w occurs in document d, the probability of class c given a test document is calculated as follows:

$$P(c|d) = \frac{P(c)\prod_{w\in d} P(w|c)^{n_{wd}}}{P(d)},$$

where $P(d)$ is a normalization factor. To avoid the zero-frequency problem, it is common to use the Laplace correction for all conditional probabilities involved, which means all counts are initialized to value one instead of zero.

Stochastic Gradient Descent

Stochastic gradient descent (SGD) has experienced a revival since it has been discovered that it provides an efficient means to learn some classifiers even if they are based on non-differentiable loss functions, such as the hinge loss used in support vector machines. In our experiments we use an implementation of vanilla stochastic gradient descent with a fixed learning rate, optimizing the hinge loss with an L_2 penalty that is commonly applied to learn support vector machines. With a linear machine, which is frequently applied for document classification, the loss function we optimize is:

$$\frac{\lambda}{2}||\mathbf{w}||^2 + \sum[1-(y\mathbf{xw}+b)]_+,$$

where w is the weight vector, b the bias, λ the regularization parameter, and the class labels y are assumed to be in $\{+1,-1\}$.

We compared the performance of our vanilla implementation to that of the Pegasos method [176], which does not require specification of an explicit learning rate, but did not observe a gain in performance using the latter. On the contrary, the ability to specify an explicit learning rate turned out to be crucial to deal with time-changing Twitter data streams : setting the learning rate to a value that was too small meant the classifier adapted too slowly to local changes in the distribution. In our experiments, we used $\lambda = 0.0001$ and set the learning rate for the per-example updates to the classifier's parameters to 0.1.

Bibliography

[1] Twitter API. http://apiwiki.twitter.com/, 2010.

[2] Hanady Abdulsalam, David B. Skillicorn, and Patrick Martin. Streaming random forests. In *IDEAS '07: Proceedings of the 11th International Database Engineering and Applications Symposium*, pages 225–232, Washington, DC, USA, 2007. IEEE Computer Society.

[3] 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.

[4] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, 1993.

[5] Rakesh Agrawal and Arun Swami. A one-pass space-efficient algorithm for finding quantiles. In *International Conference on Management of Data*, 1995.

[6] Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102–116, 2002.

[7] 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.

[8] S. Ansari, S.G. Rajeev, and H.S. Chandrashekar. Packet sniffing: A brief introduction. *IEEE Potentials*, 21(5):17–19, 2002.

[9] 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.

[10] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. 21st ACM Symposium on Principles of Database Systems*, 2002.

[11] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2002.

[12] Manuel Baena-García, José del Campo-Ávila, Raúl Fidalgo, Albert Bifet, Ricard Gavaldá, and Rafael Morales-Bueno. Early drift detection method. In *Fourth International Workshop on Knowledge Discovery from Data Streams*, 2006.

[13] P.L. Bartlett, S. Ben-David, and S.R. Kulkarni. Learning changing concepts by exploiting the structure of change. *Machine Learning*, 41(2):153–174, 2000.

[14] Michèle Basseville and Igor V. Nikiforov. *Detection of abrupt changes: theory and application*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[15] 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.

[16] Albert Bifet and Eibe Frank. Sentiment knowledge discovery in twitter streaming data. In *Discovery Science*, pages 1–15, 2010.

[17] Albert Bifet, Eibe Frank, Geoffrey Holmes, and Bernhard Pfahringer. Accurate ensembles for data streams: Combining restricted hoeffding trees using stacking. *Journal of Machine Learning Research - Proceedings Track*, 13:225–240, 2010.

[18] Albert Bifet and Ricard Gavaldà. Learning from time-changing data with adaptive windowing. In *SIAM International Conference on Data Mining*, 2007.

[19] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Moa: Massive online analysis. *Journal of Machine Learning Research*, 11:1601–1604, 2010.

[20] Albert Bifet, Geoff Holmes, Bernhard Pfahringer, Richard Kirkby, and Ricard Gavaldà. New ensemble methods for evolving data streams. In *15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2009.

[21] Remco R. Bouckaert. Choosing between two learning algorithms based on calibrated tests. In *International Conference on Machine Learning*, pages 51–58, 2003.

[22] Remco R. Bouckaert. Voting massive collections of bayesian network classifiers for data streams. In *Australian Joint Conference on Artificial Intelligence*, pages 243–252, 2006.

[23] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[24] Leo Breiman. Arcing classifiers. *The Annals of Statistics*, 26(3):801–824, 1998.

[25] Leo Breiman. Rejoinder to discussion of the paper “arcing classifiers”. *The Annals of Statistics*, 26(3):841–849, 1998.

[26] Leo Breiman. Pasting bites together for prediction in large data sets and on-line. *Machine Learning*, 36(1/2):85–103, 1999.

[27] Leo Breiman. Prediction games and arcing algorithms. *Neural Computation*, 11(7):1493–1517, 1999.

[28] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[29] Leo Breiman, Jerome Friedman, R. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.

[30] Nader H. Bshouty and Dmitry Gavinsky. On boosting with polynomially bounded distributions. *Journal of Machine Learning Research*, 3:483–506, 2002.

[31] P. Bühlmann and B. Yu. Analyzing bagging. *Annals of Statistics*, 2003.

[32] Wray Buntine. Learning classification trees. *Statistics and Computing*, 2(2):63–73, 1992.

[33] Christopher J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

[34] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *International Conference on Machine Learning*, pages 161–168, 2006.

[35] Paula Carvalho, Luís Sarmento, Mário J. Silva, and Eugénio de Oliveira. Clues for detecting irony in user-generated contents: oh...!! it's "so easy" ;-). In *Proceeding of the 1st International CIKM Workshop on Topic-sentiment Analysis for Mass Opinion*, pages 53–56, 2009.

[36] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In *Proceedings of the 4th International AAAI Conference on Weblogs and Social Media*, pages 10–17, 2010.

[37] Tony F. Chan and John Gregg Lewis. Computing standard deviations: Accuracy. *Communications of the ACM*, 22(9):526–531, 1979.

[38] 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.

[39] E. Cohen and M. Strauss. Maintaining time-decaying stream aggregates. In *Proc. 22nd ACM Symposium on Principles of Database Systems*, 2003.

[40] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, April 1960.

[41] Tamraparni Dasu, Shankar Krishnan, Suresh Venkatasubramanian, and Ke Yi. An information-theoretic approach to detecting changes in multi-dimensional data streams. In *Proc. Interface*, 2006.

[42] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, 14(1):27–45, 2002.

[43] Munmun De Choudhury, Yu-Ru Lin, Hari Sundaram, K. Selcuk Candan, Lexing Xie, and Aisling Kelliher. How does the data sampling strategy impact the discovery of information diffusion in social media? In *Proceedings of the 4th International AAAI Conference on Weblogs and Social Media*, pages 34–41, 2010.

[44] Imre Derenyi, Gergely Palla, and Tamas Vicsek. Clique percolation in random networks. *Physical Review Letters*, 94(16), 2005.

[45] Thomas G. Dietterich. Approximate statistical test for comparing supervised classification learning algorithms. *Neural Computation*, 10(7):1895–1923, 1998.

[46] Thomas G. Dietterich. Machine learning research: Four current directions. *The AI Magazine*, 18(4):97–136, 1998.

[47] Thomas G. Dietterich. Ensemble methods in machine learning. *Lecture Notes in Computer Science*, 1857:1–15, 2000.

[48] 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.

[49] Thomas G. Dietterich and Ghulum Bakiri. Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research*, 2:263–286, 1995.

[50] Carlos Domingo and Osamu Watanabe. MadaBoost: A modification of AdaBoost. In *ACM Annual Workshop on Computational Learning Theory*, pages 180–189, 2000.

[51] Pedro Domingos. Why does bagging work? A bayesian account and its implications. In *KDD*, pages 155–158, 1997.

[52] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *International Conference on Knowledge Discovery and Data Mining*, pages 71–80, 2000.

[53] 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.

[54] James Dougherty, Ron Kohavi, and Mehran Sahami. Supervised and unsupervised discretization of continuous features. In *International Conference on Machine Learning*, pages 194–202, 1995.

[55] Peter F. Drucker. *Managing for the Future: The 1990s and Beyond*. Dutton Adult, 1992.

[56] 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.

[57] 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.

[58] 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.

[59] Alan Fern and Robert Givan. Online ensemble learning: An empirical study. *Machine Learning*, 53(1/2):71–109, 2003.

[60] 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.

[61] 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.

[62] Yoav Freund. Boosting a weak learning algorithm by majority. *Information and Computation*, 121(2):256–285, 1995.

[63] Yoav Freund. An adaptive version of the boost by majority algorithm. *Machine Learning*, 43(3):293–318, 2001.

[64] Yoav Freund and Llew Mason. The alternating decision tree learning algorithm. In *International Conference on Machine Learning*, pages 124–133, 1999.

[65] 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.

[66] Yoav Freund and Robert E. Schapire. Discussion of the paper "arcing classifiers" by Leo Breiman. *The Annals of Statistics*, 26(3):824–832, 1998.

[67] Jerome Friedman and Peter Hall. On bagging and nonlinear estimation. Technical report, Stanford University, 1999.

[68] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting. *The Annals of Statistics*, 28:337–374, 2000.

[69] Jerome H. Friedman. On bias, variance, 0/1—loss, and the curse-of-dimensionality. *Data Min. Knowl. Discov.*, 1(1):55–77, 1997.

[70] 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.

[71] 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.

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

[73] J. Gama, P. Medas, and R. Rocha. Forest trees for on-line data. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 632–636, New York, NY, USA, 2004. ACM Press.

[74] João Gama and Mohamed Medhat Gaber, editors. *Learning from Data Streams: Processing Techniques in Sensor Networks*. Springer, 2007.

[75] João Gama, Pedro Medas, and Ricardo Rocha. Forest trees for on-line data. In *ACM Symposium on Applied Computing*, pages 632–636, 2004.

[76] 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.

[77] 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.

[78] 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.

[79] João Gama, Raquel Sebastião, and Pedro Pereira Rodrigues. Issues in evaluation of stream learning algorithms. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 329–338, 2009.

[80] John F. Gantz, David Reinsel, Christopeher Chute, Wolfgang Schlichting, Stephen Minton, Anna Toncheva, and Alex Manfrediz. The expanding digital universe: An updated forecast of worldwide information growth through 2011. March 2008.

[81] 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.

[82] Stuart Geman, Elie Bienenstock, and René Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4(1):1–58, 1992.

[83] Alec Go, Lei Huang, and Richa Bhayani. Twitter sentiment classification using distant supervision. In *CS224N Project Report, Stanford*, 2009.

[84] Yves Grandvalet. Bagging equalizes influence. *Machine Learning*, 55(3):251–270, 2004.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[90] Fredrik Gustafsson. *Adaptive Filtering and Change Detection*. Wiley, 2000.

[91] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, 2009.

[92] Jiawei Han and Kevin Chang. Data mining for web intelligence. *IEEE Computer*, 35(11):64–70, 2002.

[93] 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.

[94] Lars Kai Hansen and Peter Salamon. Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(10):993–1001, 1990.

[95] Michael Harries. Splice-2 comparative evaluation: Electricity pricing. Technical report, The University of South Wales, 1999.

[96] J. A. Hartigan and M. A. Wong. A k-means clustering algorithm. *Applied Statistics*, 28(1):100–108, 1979.

[97] D.P. Helmbold and P.M. Long. Tracking drifting concepts by minimizing disagreements. *Machine Learning*, 14(1):27–45, 1994.

[98] M. Herbster and M. K. Warmuth. Tracking the best expert. In *Intl. Conf. on Machine Learning*, pages 286–294, 1995.

[99] 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.

[100] J. Hilden. Statistical diagnosis based on conditional independence does not require it. *Computers in Biology and Medicine*, 14(4):429–435, 1984.

[101] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, 1998.

[102] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[103] 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.

[104] 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.

[105] 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.

[106] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *7th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, pages 97–106, San Francisco, CA, 2001. ACM Press.

[107] 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.

[108] Geoff Hulten and Pedro Domingos. VFML – a toolkit for mining high-speed time-changing data streams [http://www.cs.washington.edu/dm/vfml/]. 2003.

[109] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is np-complete. *Information Processing Letters*, 5(1):15–17, 1976.

[110] Tomasz Imielinski and Badri Nath. Wireless graffiti – data, data everywhere. In *International Conference on Very Large Databases*, pages 9–19, 2002.

[111] K. Jacobsson, N. Möller, K.-H. Johansson, and H. Hjalmarsson. Some modeling and estimation issues in control of heterogeneous networks. In *16th Intl. Symposium on Mathematical Theory of Networks and Systems (MTNS2004)*, 2004.

[112] 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.

[113] Bernard J. Jansen, Mimi Zhang, Kate Sobel, and Abdur Chowdury. Microblogging as online word of mouth branding. In *Proceedings of the 27th International Conference Extended Abstracts on Human Factors in Computing Systems*, pages 3859–3864, 2009.

[114] Akshay Java, Xiaodan Song, Tim Finin, and Belle Tseng. Why we twitter: understanding microblogging usage and communities. In *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 Workshop on Web Mining and Social Network Analysis*, pages 56–65, 2007.

[115] Ruoming Jin and Gagan Agrawal. Efficient decision tree construction on streaming data. In *Knowledge Discovery and Data Mining*, pages 571–576, 2003.

[116] John Kalucki. Twitter streaming API. http://apiwiki.twitter.com/Streaming-API-Documentation, 2010.

[117] Gopal K Kanji. *100 Statistical Tests*. Sage Publications Ltd, 2006.

[118] 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.

[119] 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.

[120] Michael Kearns and Leslie Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *Journal of the ACM*, 41(1):67–95, 1994.

[121] 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.

[122] D. Kifer, S. Ben-David, and J. Gehrke. Detecting change in data streams. In *Proc. 30th VLDB Conf., Toronto, Canada*, 2004.

[123] Richard Kirkby. *Improving Hoeffding Trees*. PhD thesis, University of Waikato, November 2007.

[124] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.

[125] R. Klinkenberg and T. Joachims. Detecting concept drift with support vector machines. In *Proc. 17th Intl. Conf. on Machine Learning*, pages 487 – 494, 2000.

[126] 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.

[127] 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.

[128] Ron Kohavi and Clayton Kunz. Option decision trees with majority votes. In *International Conference on Machine Learning*, pages 161–169, 1997.

[129] Ron Kohavi and Foster J. Provost. Applications of data mining to electronic commerce. *Data Mining and Knowledge Discovery*, 5(1/2):5–10, 2001.

[130] 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.

[131] Pat Langley, Wayne Iba, and Kevin Thompson. An analysis of bayesian classifiers. In *National Conference on Artificial Intelligence*, pages 223–228, 1992.

[132] Mark Last. Online classification of nonstationary data streams. *Intelligent Data Analysis*, 6(2):129–147, 2002.

[133] 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.

[134] Herbert K. H. Lee and Merlise A. Clyde. Lossless online bayesian bagging. *J. Mach. Learn. Res.*, 5:143–151, 2004.

[135] 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.

[136] Nick Littlestone and Manfred K. Warmuth. The weighted majority algorithm. *Information and Computation*, 108(2):212–261, 1994.

[137] Bing Liu. *Web data mining; Exploring hyperlinks, contents, and usage data*. Springer, 2006.

[138] 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.

[139] Dragos D. Margineantu and Thomas G. Dietterich. Pruning adaptive boosting. In *International Conference on Machine Learning*, pages 211–218, 1997.

[140] Dragos D. Margineantu and Thomas G. Dietterich. Pruning adaptive boosting. In *ICML '97*, pages 211–218, 1997.

[141] 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.

[142] 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.

[143] Manish Mehta, Rakesh Agrawal, and Jorma Rissanen. SLIQ: A fast scalable classifier for data mining. In *Extending Database Technology*, pages 18–32, 1996.

[144] 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.

[145] Albert Meyer and Radhika Nagpal. Mathematics for computer science. In *Course Notes*, Cambridge, Massachusetts, 2002. Massachusetts Institute of Technology.

[146] Thomas Mitchell. *Machine Learning*. McGraw-Hill Education (ISE Editions), October 1997.

[147] J. Ian Munro and Mike Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12:315–323, 1980.

[148] Sreerama K. Murthy and Steven Salzberg. Lookahead and pathology in decision tree induction. In *International Joint Conference on Artificial Intelligence*, pages 1025–1033, 1995.

[149] S. Muthukrishnan. Data streams: Algorithms and applications. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2003.

[150] Anand Narasimhamurthy and Ludmila I. Kuncheva. A framework for generating data to simulate changing environments. In *AIAP'07*, pages 384–389, 2007.

[151] 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.

[152] Brendan O'Connor, Ramnath Balasubramanyan, Bryan R. Routledge, and Noah A. Smith. From tweets to polls: Linking text sentiment to public opinion time series. In *Proceedings of the International AAAI Conference on Weblogs and Social Media*, pages 122–129, 2010.

[153] 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.

[154] Nikunj C. Oza and Stuart Russell. Online bagging and boosting. In *Artificial Intelligence and Statistics*, pages 105–112, 2001.

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

[156] Alexander Pak and Patrick Paroubek. Twitter as a corpus for sentiment analysis and opinion mining. In *Proceedings of the Seventh Conference on International Language Resources and Evaluation*, pages 1320–1326, 2010.

[157] Bo Pang and Lillian Lee. Opinion mining and sentiment analysis. *Foundations and Trends in Information Retrieval*, 2(1-2):1–135, 2008.

[158] Bernhard Pfahringer, Geoffrey Holmes, and Richard Kirkby. New options for hoeffding trees. In *AI*, pages 90–99, 2007.

[159] J. Ross Quinlan. Miniboosting decision trees, submitted for publication 1998, available at http://citeseer.ist.psu.edu/quinlan99miniboosting.html.

[160] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann, San Francisco, 1993.

[161] J. Ross Quinlan. Improved use of continuous attributes in C4.5. *Journal of Artificial Intelligence Research*, 4:77–90, 1996.

[162] Jonathon Read. Using emoticons to reduce dependency in machine learning techniques for sentiment classification. In *Proceedings of the ACL Student Research Workshop*, pages 43–48, 2005.

[163] Jorma Rissanen. A universal prior for integers and estimation by minimum description length. *The Annals of Statistics*, 11(2):416–431, 1983.

[164] S. W. Roberts. Control chart tests based on geometric moving averages. *Technometrics*, 42(1):97–101, 2000.

[165] Daniel M. Romero and Jon Kleinberg. The directed closure process in hybrid social-information networks, with an analysis of link formation on twitter. In *Proceedings of the 4th International AAAI Conference on Weblogs and Social Media*, pages 138–145, 2010.

[166] Amir Saffari, Christian Leistner, Jakob Santner, Martin Godec, and Horst Bischof. On-line random forests. In *3rd IEEE - ICCV Workshop on On-line Learning for Computer Vision*, 2009.

[167] Robert E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2):197–227, 1990.

[168] Robert E. Schapire. Using output codes to boost multiclass learning problems. In *ICML '97: Proceedings of the Fourteenth International Conference on Machine Learning*, pages 313–321, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[169] 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.

[170] Robert E. Schapire and Yoram Singer. Improved boosting using confidence-rated predictions. *Machine Learning*, 37(3):297–336, 1999.

[171] Jeffrey C. Schlimmer and Richard H. Granger. Incremental learning from noisy data. *Machine Learning*, 1(3):317–354, 1986.

[172] T. Schön, A. Eidehall, and F. Gustafsson. Lane departure detection for improved road geometry estimation. Technical Report LiTH-ISY-R-2714, Dept. of Electrical Engineering, Linköping University, SE-581 83 Linköping, Sweden, Dec 2005.

[173] Erick Schonfeld. Mining the thought stream. TechCrunch Weblog Article, http://techcrunch.com/2009/02/15/mining-the-thought-stream/, 2009.

[174] 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.

[175] 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.

[176] Shai Shalev-Shwartz, Yoram Singer, and Nathan Srebro. Pegasos: Primal Estimated sub-GrAdient SOlver for SVM. In *Proceedings of the 24th International Conference on Machine learning*, pages 807–814, 2007.

[177] 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.

[178] 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.

[179] Kenneth Stanley. Learning concept drift with a committee of decision trees. Technical Report AI Technical Report 03-302, Department of Computer Science, University of Texas at Austin, Trinity College, 2003.

[180] 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.

[181] 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.

[182] Alexey Tsymbal. The problem of concept drift: Definitions and related work. Technical Report TCD-CS-2004-15, Department of Computer Science, University of Dublin, Trinity College, 2004.

[183] Kagan Tumer and Joydeep Ghosh. Error correlation and error reduction in ensemble classifiers. *Connection Science*, 8(3/4):385–403, 1996.

[184] Paul E. Utgoff. Incremental induction of decision trees. *Machine Learning*, 4(2):161–186, 1989.

[185] 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.

[186] Leslie G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[187] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.

[188] 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.

[189] 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.

[190] G. Welch and G. Bishop. An introduction to the Kalman Filter. Technical report, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1995.

[191] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.

[192] G. Widmer and M. Kubat. Learning in the presence of concept drift and hidden contexts. *Machine Learning*, 23(1):69–101, 1996.

[193] David H. Wolpert. Stacked generalization. *Neural Networks*, 5(2):241–259, 1992.

[194] Jay Yarow. Twitter finally reveals all its secret stats. BusinessInsider Weblog Article, http://www.businessinsider.com/twitter-stats-2010-4/, 2010.

[195] Peng Zhang, Xingquan Zhu, and Yong Shi. Categorizing and mining concept drifting data streams. In *KDD '08*, 2008.