IMPROVING ON-LINE LEARNING

BY JON CHRISTIAN MESTERHARM

A dissertation submitted to the
Graduate School—New Brunswick
Rutgers, The State University of New Jersey
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
Graduate Program in Computer Science

Written under the direction of
Dr. Haym Hirsh
and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

October, 2007

ABSTRACT OF THE DISSERTATION

Improving On-line Learning

by Jon Christian Mesterharm

Dissertation Director: Dr. Haym Hirsh

In this dissertation, we consider techniques to improve the performance and applicability of algorithms used for on-line learning. We organize these techniques according to the assumptions they make about the generation of instances. Our first assumption is that the instances are generated by a fixed distribution. Many algorithms are designed to perform well when instances are generated by an adversary; we give two techniques to modify these algorithms to improve performance when the instances are instead generated by a distribution. We validate these techniques with extensive experiments using a wide range of real world data sets. Our second assumption is that the target concept the algorithm is attempting to learn changes over time. We give a modification of the Winnow algorithm and show it has good bounds for tracking a shifting concept when instances are generated by an adversary. We also consider the case that the instances are generated by a shifting distribution. We apply variations of the previous fixed distribution techniques and show, with real data derived experiments, that these techniques continue to significantly improve performance. Last, we assume that the labels for instances may be delayed for a number of trials. We give techniques to modify an on-line algorithm so that it has good performance even when the labels are delayed. We derive upper-bounds on the performance of these modifications and show through lower-bounds that these modifications are close to optimal.

Acknowledgements

First and foremost, I want to thank my adviser Haym Hirsh for the time and effort he has given me over my years at Rutgers. Whether it be brain storming new ideas, critiquing my papers, or generously providing funding, my dissertation would not be possible without his assistance. I also want to thank the other committee members, Michael Littman, Rob Schapire, and Bill Steiger, for the effort and time they spent reviewing my work.

In addition, I want to thank all the machine learning students I worked with for for their stimulating discussions and input on my work. In particular, Sofus Macskassy, Greg Perkins, Brian Davidson and Ofer Melnik all helped by either proofreading my papers, proposing new ideas, or giving dissertation advice. I especially want to thank Arunava Banerjee. To this day, I am still influenced by his keen insights on research and life.

All the work in this dissertation was based on the research I did for Nick Littlestone at NEC. He was my mentor, and this dissertation would not be possible without his early guidance. Nick helped refine my thinking process in academics and beyond. I try to live by the standard he sets.

I want thank my family and friends have always been supportive through the many ups and the downs. In particular, I wish to thank my parents for their lifelong care and support, and I want to thank Judy for her persistent belief in my eventual success and the patience that implies.

Last, I want to thank the various professors I have interacted with at Rutgers. Through the course I have taken and the classes I have TAed, they have had a strong influence on my growth and education.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Machine learning is an area of computer science that deals with making computers learn. While there are many types of learning, one of the simplest is learning from examples. Traditionally, this is called inductive learning. Consider a collection of documents that we wish to classify into various categories. This is a useful task to automate given the large number of documents found electronically on the Internet. If the computer has access to a large set of previously classified documents then it can use these documents to compute a rule to classify future documents. For example, assume we want to classify whether or not a document is about basketball, and we have a large set C of previously classified documents. The computer needs to find a rule that works on future documents based on rules that work for C.

In this dissertation, we consider a particular type of induction called on-line learning [Lit89]. In this model of inductive learning, the computer starts with zero examples and builds its knowledge one example at a time. Each example allows the computer to learn more about the problem and refine its classification rule. However, each time an example appears, the algorithm must predict the category. Assume we want to predict whether a fund in the stock market goes up or down in price. Each day we get a new example for the stock price. If the algorithm guesses the wrong category then the algorithm makes a mistake. To minimize the number of mistakes, it is important that the computer quickly learn a good prediction rule.

Before the computer can consider possible classifying rules, it needs a way to represent the example. Returning to document classification, while a document is naturally represented as a string of characters, the computer needs a representation that will have elements in common between documents. We need a representation that allows a rule

to capture what a set of documents with a particular property has in common. For our example, we want a representation that allows the computer to form a rule that includes documents about basketball and excludes all other documents. One popular representation for documents is to use a vector where each element of the vector corresponds to a specific word. If the word is in a particular document then that element of the vector is set to 1 otherwise the element is set to 0. This is refereed to as the bag of words representation [LSCP96]. To classify documents about basketball, the computer can learn to look for words such as basketball, NBA, NCAA, dunk, or dribble.

Machine learning can be applied to many different problems, therefore it is useful to have a general terminology. In what follows, we define the terminology with respect to the document problem and then give some examples to see how it generalizes to other problems. In our document example, each of the elements of the word vector is an *attribute*. The full vector of attributes for a document is an *instance* of the learning problem and the document category is the *label* of that instance. For the purpose of learning, the vector of attribute values and the label is all that is necessary to represent a document. This type of representation is useful for a wide range of learning problems [Mit97]. One may want to predict the next day's weather. This can be done inductively by using a sequence of previous days where each day is an instance with attributes based on a series of meteorological tests. Another learning problem is to determine if a patient has a particular disease. A patient instance can be represented with the results from a standard set of medical tests.

One problem with induction is that, for many learning problems, a good representation is not obvious. Currently, a machine learning algorithm needs help. The algorithm relies on someone to devise a representation that allows the algorithm to learn a rule that includes most of the correct instances and excludes most of the wrong instances. Each learning problem requires some expertise in finding a good representation for instances and multiple representations are possible. While this is still an open problem, current machine learning algorithms do partially address this issue. By allowing a large number of attributes, the learning algorithm can explore a wide range of representations. Some of the attributes may not be relevant to the problem, but it is the job of the

learning algorithm to find a good prediction function in this large space of attributes.

Given our general terminology, we can give a more precise definition of on-line learning. In on-line learning, we assume that, soon after the learning algorithm makes a prediction, the algorithm is told the correct label for the instance. This is a reasonable assumption for many problems. Going back to our weather example, after the learning algorithm predicts tomorrow's weather, it only has to wait a day before it discovers the actual weather. This is a useful model of induction because it allows the learning algorithm to receive a potentially infinite stream of labeled instances. Each instance can be used to refine the prediction hypothesis of the algorithm.

Formally on-line learning consists of a sequence of trials where each trial is composed of three steps.

1. The algorithm receives the instance.
2. The algorithm predicts the label of the instance.
3. The algorithm receives the actual label.

The algorithm can use the actual label to update its hypothesis so that it can perform better on future trials. The goal of the algorithm is to minimize the total number of mistakes over the trials [Lit88].

The major difficulty with on-line learning is guaranteeing that the algorithm always receives label feedback. When the algorithm gets an instance, it does not have the correct label for the instance; however, after the algorithm makes the prediction, the label must become available. The main type of problem that satisfies this constraint is predicting the future. As soon as the future becomes the present, the algorithm can observe the environment and determine the correct label. Of course, there must be value in knowing the label today instead of waiting for tomorrow. For example, if a wedding planner knows it is going to rain tomorrow, he can prepare to move the wedding indoors today. If a CPU knows the outcome of a future branch instruction, it can start filling up the processor pipeline with the correct instructions. If a computer interface knows the next option a user is going to select, it can make that option easier to select and start early processing of that option.

A different approach to induction is to take a set of labeled instances and use them to learn a classifying function. This classifying function is then fixed and used for future prediction of labels. This creates two phases for the learning algorithm. First the algorithm trains on a set of labeled instances to create a hypothesis; second the hypothesis is used for future prediction. In the machine learning literature, this type of inductive learning is called batch learning [Lit88]. Batch learning is popular because the second phase does not require labels for instances. For many problems, it is difficult to guarantee the label will eventually become available. For example, if we want to train an algorithm to read books into a computerized format, we do not want to continually provide the correct text to the algorithm. The whole purpose of using the algorithm is to automate the text recognition. In this case, batch learning is more appropriate. We can train the algorithm until it reaches an acceptable level of performance and then use the hypothesis it generates for all future text recognition.

While batch learning is a useful model, for those problems where label feedback is available, on-line learning is a better choice. Why stop learning when more labels are always available? The longer an on-line algorithm is used, the more it can potentially improve its performance by exploiting the extra labels.

The purpose of this thesis is to improve upon the existing research in on-line learning and make on-line learning more practical. This includes improving existing on-line algorithms to decrease the number of mistakes for practical problems and extending the existing on-line algorithms to handle a wider range of problems. It even includes extending the on-line model to study and solve new types of practical problems. In general, our main purpose is to expand the applicability and performance of the on-line model.

1.1 Linear-threshold Algorithms

In this section, we describe the most popular class of algorithms for on-line learning. These algorithms represent a hypothesis with a hyperplane in the attribute space where each side of the hyperplane predicts a different label. For convenience, we use -1 and 1

as the two binary labels. This representation is also called a linear-threshold function, since the hyperplane can be represented by a linear function with a constant threshold term [Lit88]. For this reason, we call any algorithm that represents its hypothesis as a hyperplane a linear-threshold algorithm.

More formally, assume an instance is a vector $\mathbf{x}$ with n attributes. Let x_i be the ith attribute. We assume that each x_i is an element of $[0, 1]$.[1] A linear-threshold function uses $n + 1$ real valued weights to compute the prediction for an instance. There is one weight, w_i, for each attribute and an extra weight θ called the threshold. The dot product of $\mathbf{w}$ and $\mathbf{x}$ is compared to θ. If $\sum_{i=1}^{n} w_i x_i \geq \theta$ the prediction is 1 otherwise the prediction is -1

There are many different linear-threshold algorithms. The linear-threshold threshold algorithms we cover in this thesis include the Perceptron [Ros62], Winnow [Lit89], and ALMA [Gen01]. One advantage of these algorithms is that they are efficient for prediction and updating using only $O(n)$ time per trial. Another advantage is that they have good performance guarantees even when an adversary is allowed to pick the instances. The adversary is allowed to pick any sequence of instances, but the adversary is constrained to label most of the instances based on a linear-threshold function. Even with an adversary picking the instances, these algorithms have nontrivial upper-bounds on the total number of mistakes. In this thesis, we often call algorithms with these types of performance guarantees adversarial algorithms.

Since linear-threshold algorithms only use a hyperplane to represent the hypothesis, one may think they would do poorly on many real world problems; however, good accuracy is possible on many data-sets even with such a seemingly simple hypothesis. In [DP97], the authors show that Naive Bayes, which uses a linear-threshold function, is competitive with more expressive learning algorithms on data from the U. C. Irvine repository [DNMml]. The U. C. Irvine data repository is a diverse collection of machine learning problems that have been submitted by various machine learning researchers. This shows that a linear-threshold function is a reasonable representation for a wide

[1] While many linear-threshold algorithms can work with real valued attributes outside of $[0, 1]$, in practice it is useful to normalize the attributes to some interval.

range of learning problems.

Additional research has shown that linear-threshold algorithms work well on other real world problems. Much of this research has focused on the recent class of Winnow algorithms [Lit89]. The Winnow algorithms are desirable because they can perform well even if many of the attributes are not relevant to the target function. For example, the text learning problem, which we talked about earlier in this chapter, creates a large instance vector to describe documents. The maximum size of an instance could be close to the number of words in the dictionary. However, only a small fraction of these attributes may be useful. As we will see in the next chapter, the Winnow algorithms make few mistakes while finding these relevant attributes.

This advantage makes it possible to add many speculative features to the Winnow algorithms. For example, to take the text example even further, instead of just dealing with single words in a document, one can look at pairs of words. For every pair of words, have an attribute that is 1 when that pair of words appear next to each other in the document. This can cause an explosion in the number of features, but the computational efficiency of the Winnow algorithm and its ability to find the relevant attributes makes the problem tractable.

Based on this advantage, Winnow algorithms have been used on a wide range of problems. Winnow algorithms have been found effective for efficiently learning how to categorize text into different classes [CS96, LSCP96, DKR97, KMB03, BKV03]. This is particularly useful given the large amount of text information found on the Internet. The ability to handle large amounts of text is also useful for natural language processing (NLP). This includes tasks such as segmenting sentences and tagging words with different parts of speech [KR98, ZDJ01, TCS03, RZ98, RtY01, Sid02]. In a NLP related task, Winnow has been used to help with spell checking [GR99, BB01, MKCN98]. It has even been used on tasks as diverse as predicting instruction branches in a CPU [LH02], determining when to stop spinning a hard drive for mobile applications [HLSS00], predicting calendar appointments [Blu95], and predicting which links a user should follow on a website [AFJM95].

Interestingly, many of these problems do not directly fit into the on-line model.

They may be either batch problems, or they may make or need extra assumptions not included in the traditional on-line model. To handle these differences, much of this work makes modifications to the original Winnow algorithms to handle the particular problem and improve performance. In this thesis, our goal is similar but broader. We want to find general ways to take existing on-line algorithms and improve them based on the extra assumptions one is likely to encounter in a real problem. Much of our thesis will deal with linear-threshold algorithms with a focus on Winnow algorithms. Some of our results will be more general, applying to other on-line algorithms, but we limit our testing to linear-threshold algorithms.

1.2 Thesis Overview

The purpose of this dissertation is to improve the performance and applicability of on-line learning algorithms. Most of our results deal with modifying existing on-line algorithms to improve performance when certain explicit assumptions are true for a problem. The goal of all our modifications is to improve the algorithm while maintaining the benefits of the original on-line learning algorithm. To help distinguish between the original and the modified algorithms, we often refer to the original on-line algorithm as the basic algorithm.

The second chapter gives information about the basic on-line linear-threshold algorithms we use in the thesis. An advantage of these on-line learning algorithms is that they are designed to perform well even when an adversary is allowed to pick the instances. In Chapter 2, we describe the algorithms and give bounds on the maximum number of mistakes for problems where instances are generated by an adversary. In many cases, these bounds are improvements over previously published bounds. At the end of the chapter, we give a common notation that allows us to compare the mistake bounds for all the algorithms.

While it is impressive that adversarial algorithms can perform well under such adverse conditions, for many real world problems an adversary is not a realistic assumption. Chapter 3 and Chapter 4 give techniques to improve the performance of

adversarial algorithms when instances are generated by something weaker than an adversary. The technique in Chapter 3 predicts using a weighted vote from a collection of hypotheses. This voting technique is designed to improve performance when instances are generated by a fixed distribution. In Chapter 4, we improve performance by reusing instances from previous trials. Since adversarial algorithms make a bounded number of mistakes, any mistake made on a saved instance could prevent a mistake on a real instance. We perform experiments with both of these techniques on real world data. In Chapter 5, we combine both of these techniques and show that the combined algorithm gives even better performance on real world data.

In on-line learning, it is possible to learn a target function that changes over time. For example, assume a learning algorithm is trying to predict if the price of a stock will go up or down during a day of trading. The stock market has many influences that gradually or infrequently change over time. A prediction function that works well one month may not work well next month because of influences that are not captured in the instance representation. Because an on-line learning algorithm continually receives newly labeled instances, it can potentially modify its hypothesis to track these changes. In Chapter 6, we give a modified version of the Winnow algorithm that allows the tracking of changing target functions. We compare this algorithm to a version of ALMA, another recent algorithm that can track target functions. In Chapter 7, we perform experiments on these and other linear-threshold algorithms when learning a shifting target function. To improve performance, we use some of the techniques from Chapter 3 and Chapter 4.

The greatest restriction of the on-line model is the need for the correct label at the end of a trial. For many problems, it is not possible to get the correct label before the start of the next trial. For example, a doctor may wait a week before he gets the results of a blood test. While waiting for the result, he may need to diagnose several other patients. In the last chapter, we show how to modify the on-line model to handle delayed labels. We give different algorithms to handle fixed or shifting target functions and to handle instances generated by an adversary or a distribution. We show that these new algorithms are close to optimal for delayed feedback.

The final chapter reviews the contributions of this dissertation and considers possible future work. Some of this future work is based on on discoveries made during the course of our research. Other parts are based on interesting extensions of on-line learning that we have not yet had a chance to explore.

Chapter 2

On-line Linear-threshold Algorithms

In this chapter, we describe the linear-threshold learning algorithms used in this thesis. All these algorithms have been previously published, so we only give a short description of each algorithm along with references where further details can be found. In addition, we give some information that may be useful in determining when to use the algorithm, such as a bound on the number of mistakes and conditions for when that bound is valid. Previously these bounds have used different notation making them difficult to compare. Therefore, towards the end of the chapter, we convert all the bounds to a single notation.

We use these linear-threshold algorithms as basic algorithms on which the techniques given in this thesis will be applied. We cover many popular algorithms in order to show how our techniques work on a wide range of linear-threshold algorithms. For the most part, our results suggest not using the algorithms in the forms given in this chapter, but to instead use a modified algorithm as discussed in later chapters.

This chapter is organized as follows. We start with some essential notation and then work into a proof of an upper-bound on the number of mistake for the Perceptron algorithm. In Section 2.3, we then give mistake bounds and other information for all the linear-threshold algorithms used in this dissertation. In Section 2.4, we convert these bounds into a single notation and compare the bounds. Last, we explain a popular technique used to deal with the many parameters found in several of the algorithms.

2.1 Notation and Terminology

The notation we use in this chapter builds off the notation given in Chapter 1. Vector $\mathbf{x}_t$ represents the instance the algorithm receives on trial t. This vector is of size n

and for most algorithms we assume that each component is an element of $[0, 1]$. Often, we need to deal with the components of the instance vector; we use $x_{i,t}$ to refer to the ith component of instance $\mathbf{x}_t$. Each instance has a label $y_t \in \{-1, 1\}$. Let s be a sequence of these instances. The set of trials where on-line algorithm B makes a mistake on sequence s is denoted by $M(B, s)$. When the algorithm and sequence can be determined from context, we use the notation M_T where T refers to the length of the sequence. To refer to the number of elements in a set or sequence A, we use the standard notation $|A|$. For example, the number of mistakes made by an algorithm can be represented as $|M_T|$.

The linear-threshold functions in this chapter learn a function from the instances to the labels. This binary function is often called a concept. The weights of the linear-threshold function, at trial t, are represented by the vector $\mathbf{w_t}$. Again, we use $w_{i,t}$ to represent the ith component of the vector. The weight vector is of the same size, n, as the instance vector and predictions are made by taking the dot product of the weight vector and the instance vector and comparing the result to $\theta \in \mathbf{R}$. If the dot product is greater than or equal to θ then the algorithm predicts 1. If the dot product is less than θ then the algorithm predicts -1.

The algorithms we consider in this chapter only change their state after they have made a mistake on a prediction. These are called mistake-driven algorithms. A related definition, that we use later in the dissertation, is a conservative algorithm. This is an algorithm that only changes its prediction hypothesis after a mistake.[1] The key difference between these definitions is that a mistake-driven algorithm is never affected by instances that are predicted correctly. A conservative algorithm does not change its prediction hypothesis when it makes a correct prediction, but the instance can still affect the state of the algorithm and therefore can have an influence on future predictions. Notice that mistake-driven algorithms are a subset of conservative algorithms. More information on mistake-driven algorithms can be found in Appendix F.

[1]There is some discrepancy in the literature with the definition of a conservative algorithm. Some people use the term conservative and mistake-driven interchangeably.

Perceptron
Initialization
$t \leftarrow 1$ is the current trial.
$\forall i \in \{1, \ldots, n\}$ $w_{i,1} = 0$.
Trials
Instance: $\mathbf{x}_t \in \mathbf{R}^n$.
Prediction: If $\mathbf{w_t} \cdot \mathbf{x_t} \geq 0$ then
predict $\hat{y}_t = 1$ else predict $\hat{y}_t = -1$.
Update: Let $y_t \in \{-1, 1\}$ be the correct label.
If $y_t(\mathbf{w_t} \cdot \mathbf{x_t}) \leq 0$ then
$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = w_{i,t} + y_t x_{i,t}$.
Else
$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = w_{i,t}$.
$t \leftarrow t + 1$.

Figure 2.1: Pseudo-code for the Perceptron algorithm.

2.2 Mistake-bounds

One advantage of the linear-threshold algorithms used in this thesis is that we can bound the maximum number of mistakes the algorithm makes even when instances are selected by an adversary. To help explain the mistake bounds of linear-threshold algorithms, we start with an in-depth example. One of the first algorithms able to learn arbitrary linear-threshold functions was the Perceptron [Ros62]. The code and the proof for the Perceptron algorithm are fairly straightforward, so it is a good choice to help explain some of the common properties of the mistake bounds used in this thesis. We give the code for the Perceptron algorithm in Figure 2.1 This code is similar to the algorithm published in [DH73].

The algorithm works according to the on-line model explained in Chapter 1. The algorithm has n weights; one weight for each attribute of the learning problem. The weights are initialized to 0. After initialization, the algorithm starts to predict the labels of instances. This is broken down into three steps. First the algorithm receives the instance. Second the prediction is made by taking the dot product of the weight vector and the instance. If this dot product is at least 0 then the algorithm predicts 1; otherwise, the algorithm predicts -1. Last, the algorithm receives the correct label. If the algorithm makes a mistake on the instance ($y_t \mathbf{w}_t \cdot \mathbf{x}_t \leq 0$) then the weights are

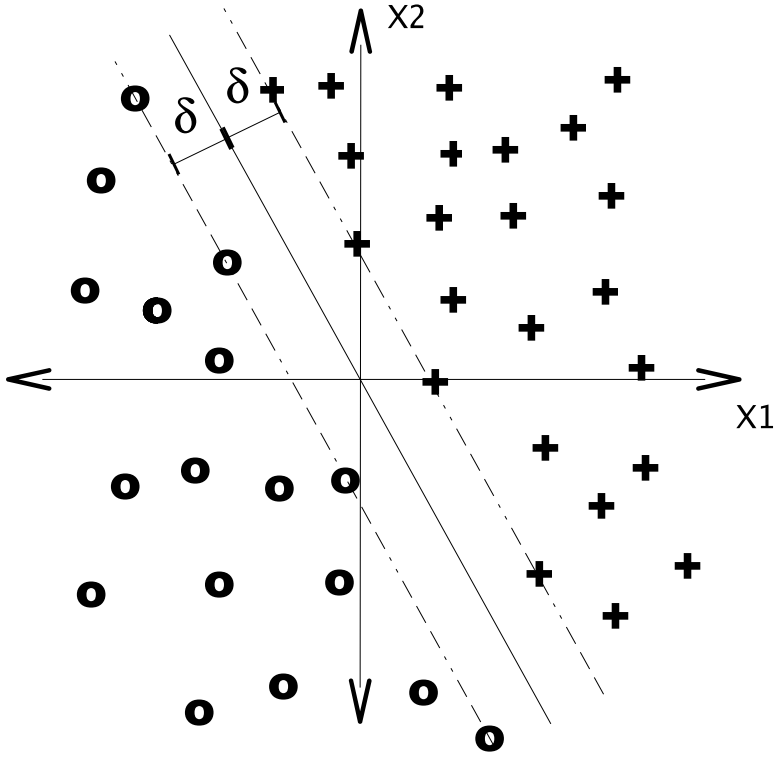

Figure 2.2: Linear-threshold target function where no instance is allowed to occur within a δ margin around the target function.

adjusted to counteract the mistake. If the dot product is too small, the algorithm changes the weights to increase the dot product for the instance. If the dot product is too big then the algorithm changes the weights to decrease the dot product for the instance. The weights are moved by adding or subtracting the instance vector from the weight vector.

As is the standard practice, we do not include an explicit threshold term in the Perceptron algorithm. To learn linear-threshold functions that have a non-zero threshold, an extra attribute with a fixed value of 1 is added to all instances [DH73]. Its weight is updated along with the other weights allowing the algorithm to effectively learn the threshold. This is a standard technique we will see with other linear-threshold learning algorithms; we add extra attributes to expand the set of functions the algorithm can represent.

Next we want to prove an upper-bound on the number of mistakes made by the Perceptron algorithm when the instances are generated by an adversary. For a first

effort, we assume that no instance generated by the adversary can get too close to the hyperplane that separates the two labels. Later, we will relax this assumption and allow noisy instances. In Figure 2.2, we have drawn a hyperplane where all the instances are predicted correctly, but no instances are allowed to occur within a distance $\delta > 0$ of the linear-threshold function. More formally, let $\mathbf{u}$ be the weights of the target function. All instances must satisfy $y_t \mathbf{u} \cdot \mathbf{x_t} \geq \delta$. This is the only assumption we make on the instances.[2] The adversary is allowed to know all the details of the learning algorithm and can generate any instance that satisfies the above separation condition.

Using the above idea of a margin, we are almost ready to give a proof of the Perceptron mistake bound. The proof is typical of proofs used for linear-threshold algorithms and is similar to the proof given in [DH73]. It shows that a certain "progress" function of the learning algorithm weights must decrease by at least a constant, positive amount every time the learning algorithm makes a mistake. It also shows that this function has a minimum value. This gives the finite mistake bound. In the case of the Perceptron, the progress function is just the squared two-norm of the difference between the current weight vector and a multiple of the target vector. In other words, the progress function is $\|\mathbf{w_t} - a\mathbf{u}\|_2^2$. Intuitively this is a reasonable choice since we want the weight vector to get close to the target vector. The constant a is needed since any multiple of the target weights makes identical predictions. We use the constant a to create the target function that the algorithm weights approach.

Before we give the proof, we need some new notation. Let β^2 be the average square of the two-norm of the instance vectors over all trials where there is a mistake. More formally, let M_T be the set of trials that have a mistake up to trial T. Let $\beta^2 = \sum_{t \in M_T} \|\mathbf{x}_t\|_2^2 / |M_T|$.

Theorem 2.1 *When all instances satisfy $y_t \mathbf{u} \cdot \mathbf{x_t} \geq \delta$, the number of mistakes made by Perceptron is at most $\beta^2 \|\mathbf{u}\|_2^2 / \delta^2$.*

Proof Assume there is a mistake made on trial T. Based on the update procedure of

[2] This description of the target function is not unique. We can multiply $\mathbf{u}$ and δ by any constant and get the same target function.

the Perceptron algorithm

$$(\mathbf{w}_{T+1} - a\mathbf{u}) = (\mathbf{w}_T - a\mathbf{u}) + y_T\mathbf{x_T}.$$

Taking the two norm of both sides and squaring both sides gives,

$$\|\mathbf{w}_{T+1} - a\mathbf{u}\|_2^2 = \|\mathbf{w}_T - a\mathbf{u}\|_2^2 + 2(\mathbf{w}_T - a\mathbf{u}) \cdot (y_T\mathbf{x_T}) + \|\mathbf{x}_T\|_2^2.$$

Because $\mathbf{x}_T$ was misclassified, we know that $y_T\mathbf{w_T} \cdot \mathbf{x_T} \leq 0$. Also because of the restriction on the adversary, $y_T\mathbf{u} \cdot \mathbf{x_T} \geq \delta$. Using these facts, we get

$$\|\mathbf{w}_{T+1} - a\mathbf{u}\|_2^2 - \|\mathbf{w}_T - a\mathbf{u}\|_2^2 \leq -2a\delta + \|\mathbf{x}_T\|_2^2.$$

This means that the $\|\mathbf{w}_T - a\mathbf{u}\|^2$ must change by $-2a\delta + \|\mathbf{x}_T\|_2^2$ on a mistake. Summing over all mistakes up to trial T gives

$$\|\mathbf{w}_{T+1} - a\mathbf{u}\|_2^2 - \|\mathbf{w}_1 - a\mathbf{u}\|_2^2 \leq -2a\delta|M_T| + \sum_{t \in M_T} \|\mathbf{x}_t\|_2^2.$$

We can drop the first norm term and use the fact that $\mathbf{w}_1$ is all zeros to get

$$-a^2\|\mathbf{u}\|_2^2 \leq -2a\delta|M_T| + \sum_{t \in M_T} \|\mathbf{x_t}\|_2^2.$$

Using the fact that $\beta^2 = \sum_{t \in M_T} \|\mathbf{x_t}\|_2^2/|M_T|$ gives

$$-a^2\|\mathbf{u}\|_2^2 \leq -2a\delta|M_T| + \beta^2|M_T|.$$

As long as $a > \beta^2/(2\delta)$, we can use the above equation to bound $|M_T|$ with

$$|M_T| \leq \frac{a^2\|\mathbf{u}\|_2^2}{2a\delta - \beta^2}.$$

Setting $a = \beta^2/\delta$ minimizes this bound and proves the theorem. ■

This new proof is similar but slightly stronger than the proof of [DH73]; the primary difference is that we define β^2 based on the average value of $\|\mathbf{x_t}\|_2^2$ during mistakes as opposed to the maximum value. This shows that the Perceptron will not be unduly affected if a single instance has an exceptionally large $\|\mathbf{x_t}\|_2^2$; it is the average value that matters. Of course, this average is an average over all the mistakes. An adversary can

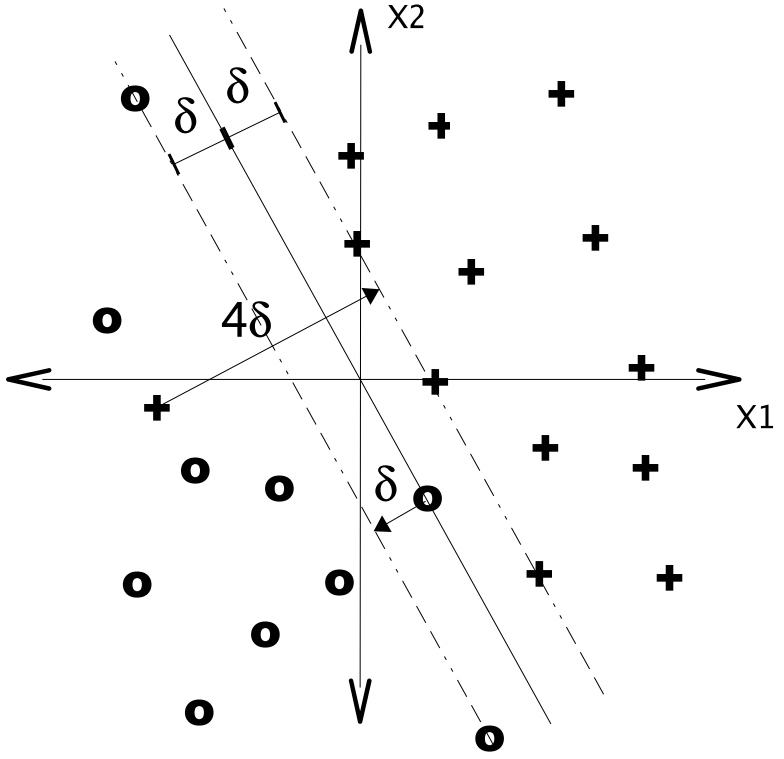

Figure 2.3: Linear-threshold target function with instances with δ noise and 4δ noise.

maximize the mistake bound by creating mistakes on instances with large $\|\mathbf{x_t}\|_2^2$. However, the β^2 term is still useful because we sometimes apply the Perceptron algorithm on problems where the instances are generated by something weaker than an adversary.

The Perceptron mistake bound grows rapidly as δ shrinks. In addition, the mistake bound does not apply if there does not exist a hyperplane that correctly classifies the data. Fortunately, because of the nature of the proof technique, it is generally easy to modify these proofs to include noisy instances that are either inside the margin, allowing for a larger margin, or on the wrong side of the target hyperplane. We just need to keep track of how these noisy instances affect the progress function.[3] Generally, a mistake on a noisy instance will move the progress function in the wrong direction. If we can bound the amount these noisy instances change the progress function, we can bound the extra mistakes that are needed to correct these changes.

Because $y_t\mathbf{u} \cdot \mathbf{x_t} \geq \delta$ for a non-noisy instance, the amount of noise can be measured

[3]Littlestone gives a general technique to modify progress function based proofs to handle noise in instances [Lit89]. However, he assumes each noisy instance has the same effect on the progress function.

by the size of $\delta - y_t \mathbf{u} \cdot \mathbf{x_t}$. The larger this number, the larger the noise. Let $\nu_t = \max(0, \delta - y_t \mathbf{u} \cdot \mathbf{x_t})$ be the noise in trial t. This noise value is often called the hinge-loss [GW99]. If an instance is non-noisy then ν_t will be 0. If an instance is noisy then ν_t is the perpendicular distance of the instance from the side of the margin that corresponds to the correct classification of the instance. See Figure 2.3 for an example. Notice that the syntactic form of the noise function contains all the information about the target function: the target weights $\mathbf{u}$ and the margin δ. Therefore, for most algorithms we only specify the definition of the noise function; the target function is implied.

We can use this measure of noise to give a new mistake bound based on the amount of noise in the instances. Let N be an upper-bound on the total amount of noise up to trial T; in other words, $\sum_{t=1}^{T} \nu_t \leq N$. Using this definition of noise, we are ready to give the proof. The proof is similar to the no noise case except we let the value of a remain a variable.

Theorem 2.2 *For $a > \beta^2/(2\delta)$, the number of mistakes made by Perceptron is at most*

$$\frac{a^2 \|\mathbf{u}\|_2^2 + 2aN}{2a\delta - \beta^2}.$$

Proof Assume there is a mistake made on trial t. Based on the update procedure of the Perceptron algorithm

$$(\mathbf{w}_{t+1} - a\mathbf{u}) = (\mathbf{w}_t - a\mathbf{u}) + y_t \mathbf{x_t}.$$

Taking the two norm of both sides and squaring both sides gives,

$$\|\mathbf{w}_{t+1} - a\mathbf{u}\|_2^2 = \|\mathbf{w}_t - a\mathbf{u}\|_2^2 + 2(\mathbf{w}_t - a\mathbf{u}) \cdot (y_t \mathbf{x_t}) + \|\mathbf{x}_t\|_2^2.$$

Because $\mathbf{x}_t$ was misclassified, we know that $y_t \mathbf{w_t} \cdot \mathbf{x_t} \leq 0$. Using this fact, we get

$$\|\mathbf{w}_{t+1} - a\mathbf{u}\|_2^2 \leq \|\mathbf{w}_t - a\mathbf{u}\|_2^2 - 2a y_t \mathbf{u} \cdot \mathbf{x_t} + \|\mathbf{x}_t\|_2^2.$$

Next we use the fact that $y_t \mathbf{u} \cdot \mathbf{x_t} \geq \delta - \nu_t$. We get

$$\|\mathbf{w}_{t+1} - a\mathbf{u}\|_2^2 - \|\mathbf{w}_t - a\mathbf{u}\|_2^2 \leq -2a\delta + 2a\nu_t + \|\mathbf{x}_t\|_2^2.$$

This means that the $\|\mathbf{w}_t - a\mathbf{u}\|_2^2$ must change by $-2a\delta + 2a\nu_t + \|\mathbf{x_t}\|_2^2$ on every mistake. Summing over all mistakes up to trial T gives

$$\|\mathbf{w}_{t+1} - a\mathbf{u}\|_2^2 - \|\mathbf{w}_1 - a\mathbf{u}\|_2^2 \leq -2a\delta|M_T| + \sum_{t \in M_T} 2a\nu_t + \sum_{t \in M_T} \|\mathbf{x_t}\|_2^2.$$

We can drop the first norm term and use the fact that $\mathbf{w}_1$ is all zeros, to get

$$-a^2\|\mathbf{u}\|_2^2 \leq -2a\delta|M_T| + 2a \sum_{t \in M_T} \nu_t + \sum_{t \in M_T} \|\mathbf{x_t}\|_2^2.$$

Using the definition of noise, N and the definition of β^2 gives,

$$-a^2\|\mathbf{u}\|_2^2 \leq -2a\delta|M_T| + 2aN + \beta^2|M_T|.$$

As long as $a > \beta^2/(2\delta)$, we can use the above equation to bound $|M_T|$ with

$$|M_T| \leq \frac{a^2\|\mathbf{u}\|_2^2 + 2aN}{2a\delta - \beta^2}.$$

■

The value of a that minimizes the bound gives a somewhat complicated bound. Therefore we start with a simpler bound that, while not optimal, is easy to relate to the no noise case.

Corollary 2.3 *The number of mistakes made by Perceptron is at most*

$$\frac{\beta^2\|\mathbf{u}\|_2^2}{\delta^2} + \frac{2N}{\delta}.$$

Proof Just set $a = \beta^2/\delta$ in Theorem 2.2. ■

This bound is the same as the bound given in Theorem 2.1 except that we add the term $2N/\delta$ to account for the noisy instances. Next we give the optimal bound.

Corollary 2.4 *Let* $\xi = \sqrt{1 + \frac{4N\delta}{\beta^2\|\mathbf{u}\|_2^2}}$. *The number of mistakes made by Perceptron is at most*

$$\frac{\beta^2\|\mathbf{u}\|_2^2}{2\delta^2}\left(1 + \frac{1}{\xi}\right) + \frac{N}{\delta}\left(1 + \frac{2}{\xi}\right).$$

Proof Using $a = \left(\|\mathbf{u}\|_2^2 \beta^2 + \sqrt{\|\mathbf{u}\|_2^4 \beta^4 + 4\|\mathbf{u}\|_2^2 \delta N \beta^2} \right) / (2\|\mathbf{u}\|_2^2 \delta)$ minimizes the bound in Theorem 2.2. Plugging this value into Theorem 2.2 gives the result. ■

This bound shows that, as the noise starts to dominate, the Perceptron algorithm makes N/δ, plus lower order terms, mistakes. Therefore our simple bound, in Corollary 2.3, is somewhat deceptive as it does not give the correct long term behavior of the algorithm with respect to noisy instances. However, for ξ close to 1, the optimal bound becomes deceptive. In the limit, the concept term goes to $\beta^2 \|\mathbf{u}\|_2^2 / \delta^2$ while the noise term goes to $3N/\delta$. This bound appears to be greater than our simple bound. In truth, the bound is still optimal. The interaction of the two terms gives a bound that is close to the bound in Corollary 2.3 for ξ values close to 1.[4]

It is interesting to note that the Perceptron bound depends on the representation used for the instances. For example, binary attributes which are normally represented as either $\{0, 1\}$ or $\{-1, 1\}$. For some binary problems, the $\{0, 1\}$ representation will give a better bound because the β^2 factor will be much smaller when many attributes have value 0. This is partially offset because δ is smaller when using a $\{0, 1\}$ representation. In this thesis, many of the experiments have few non-zero attributes. Therefore we always use a $\{0, 1\}$ representation for binary attributes.

Also notice that the Perceptron algorithm allows attributes $\mathbf{x}_t \in R^n$. Most of the other proofs, we cover in this thesis, assume $\mathbf{x}_t \in [0, 1]^n$. While it is possible to extend the other proofs, we feel it is a reasonable assumption to assume the attributes are normalized to be in the $[0, 1]$ interval.

While the Perceptron algorithm is well studied, as far as we know, the above proof has not been previously published. In [FS98], an alternative proof is given for the Perceptron with noisy instances using the technique of [KS95]. This technique handles noisy instances by allowing an extra attribute for each trial. The target weight of this attribute is used to correct for any noise on the trial and allow a hyperplane that

[4] This can be seen by looking at the Taylor series of the bound around $N = 0$.

perfectly separates the instances. This is a general technique that can convert a linear-threshold learning algorithm with a no noise proof into an algorithm that can handle noisy instances. In the case of the Perceptron, this technique does not change the behavior of the learning algorithm, so it gives a proof that Perceptron can learn noisy instances. To describe the bound, we need some new notation. Let $\beta' = \max_{t \in T} \|\mathbf{x_t}\|_2$. Clearly $\beta \leq \beta'$. Let $N' = \sqrt{\sum_{t=1}^{T} \nu_t^2}$. The new bound is $(\beta' \|\mathbf{u}\|_2 + N')^2 / \delta^2$. Even assuming that $\beta' = \beta$, this bound is always higher than our bound in Corollary 2.4.

At this point, we want to clarify the generality of the noise model based on hinge-loss. This noise model allows us to give mistake bounds that covers many types of noise simply by making assumptions about ν_t. For example, a common assumption is to allow only a fixed amount of noise by assuming $\sum_{t=1}^{\infty} \nu_t \leq K$ for some constant $K \in \mathbf{R}$. Another possibility is to allow $\sum_{t=1}^{T} \nu_t \leq KT$. This allows the noise to grow linearly with the trial number. We can even assume that the various ν_t are random variables allowing something similar to the noise model in [Lit91]. In general, the noise component of the mistake bound at trial T will depend on $\sum_{t \in M} \nu_t$ no matter how the noisy instances are generated.

A subtle aspect of the previous Perceptron mistake-bound is that any value of $\mathbf{u}$ and δ gives a legal mistake bound for a sequence of instances. Of course, if a given $\mathbf{u}$ misclassifies most of the instances then the noise will be large. If another $\mathbf{u}$ exists that correctly classifies most of the instances, it will most likely give a better bound. Another possibility is that $\mathbf{u}$ correctly classifies the instances, but many of the instances are within δ of the hyperplane. In that case, a smaller δ is likely to give a better bound. Since all target functions are valid, what we are really interested in is a target function that gives the minimum mistake bound. However, any target function we use gives an upper-bound on that minimum.

2.3 Algorithms

In this section, we give the details on the remaining basic algorithms we use in this thesis.[5] This includes pseudo-code, upper-bounds on mistakes, and some relevant facts for each algorithm. Most of the mistake bounds are refinements of existing bounds and have not been previously published. We include the proofs of these bounds in the appendix. In Section 2.4, we compare the bounds of the various algorithms.

2.3.1 Unnormalized Winnow Algorithm

The Unnormalized Winnow algorithm is the first of a series of "Winnow" algorithms introduced by Littlestone [Lit88, Lit89, Lit91]. Its original name is Winnow, but we follow the convention of [HPW99] and call it Unnormalized Winnow so as not to confuse it with the other Winnow style algorithms that we also discuss. Unnormalized Winnow, like Perceptron, can learn arbitrary linear-threshold functions, but it has the advantage of making fewer mistakes for problems with a large number of attributes where only a small fraction of the attributes are used in the target function. This can be seen in the mistake bounds given below and will be further clarified in Section 2.4.2.

The code for Unnormalized Winnow is given in Figure 2.4. The main difference between the Winnow algorithms and Perceptron is that the Winnow algorithms perform a multiplicative update on the weights instead of an additive update. Specifically, Perceptron uses $w_{i,t+1} = w_{i,t} + y_t x_{i,t}$, and Unnormalized Winnow uses $w_{i,t+1} = w_{i,t}\alpha^{y_t x_{i,t}}$. The constant $\alpha > 1$ is used to control the strength of the updates.

Next, we specify the noise function used in our mistake-bound for Unnormalized Winnow. Since Unnormalized Winnow always multiplies its weights by $\alpha > 1$, and the initial weights have value σ, it can not directly learn negative weights. Therefore all u_i must be greater or equal to zero.[6] Let $\delta \in (0, 1]$ be the margin. The noise on instance $\mathbf{x_t}$ is defined as $\nu_t = \max(0, \delta - y_t(\mathbf{u} \cdot \mathbf{x_t} - 1))$, and the total noise, up to trial T, is $N = \sum_{t=1}^{T} \nu_t$.

[5] Information on the Perceptron algorithm can be found in Section 2.2.

[6] Later in the section, we show how to change the representation to effectively allow negative target weights.

Unnormalized Winnow(α, σ)
Parameters
$\alpha > 1$ is the update multiplier.
$\sigma > 0$ is the initial weight value.
Initialization
$t \leftarrow 1$ is the current trial.
$\forall i \in \{1, \ldots, n\}$ $w_{i,1} = \sigma$ are the weights.
Trials
Instance: $\mathbf{x}_t \in [0,1]^n$.
Prediction: If $\mathbf{w_t} \cdot \mathbf{x_t} \geq 1$
predict $\hat{y}_t = 1$ else predict $\hat{y}_t = -1$.
Update: Let $y_t \in \{-1, 1\}$ be the correct label.
If $y_t(\mathbf{w_t} \cdot \mathbf{x_t}) \leq 0$ then
$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = \alpha^{y_t x_{i,t}} w_{i,t}$.
Else
$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = w_{i,t}$.
$t \leftarrow t + 1$.

Figure 2.4: Pseudo-code for the Unnormalized Winnow algorithm.

This parallels the noise definition used for the Perceptron algorithm. Any instance that is within δ distance of the hyperplane specified by $\mathbf{u} \cdot \mathbf{x_t} - 1$ will have a positive noise. The further the instance is from the side of the half-space that corresponds to its label, the larger the noise. The noise function explicitly includes the threshold term -1 to match the negative of the threshold used in Unnormalized Winnow. For most cases, an algorithm that predicts 1 if and only if $\mathbf{w_t} \cdot \mathbf{x_t} \geq c$ uses a noise function of the form $\nu_t = \max(0, \delta - y_t(\mathbf{u} \cdot \mathbf{x_t} - c))$.

Theorem 2.5 *The number of mistakes made by Unnormalized Winnow when* $\alpha = 1+\delta$ *and* $\sigma = 1/n$ *is at most*

$$\frac{(2+\delta)\left(1 + \sum_{i=1}^{n} u_i \ln u_i + \ln(n) \sum_{i=1}^{n} u_i - \sum_{i=1}^{n} u_i\right)}{\delta^2} + \frac{(2+\delta^2/5)N}{\delta}.$$

The proof of this result can be found in Appendix A along with an explanation for the parameter choices. The proof is a small refinement of previous proofs [Lit88, Lit89, Lit91]. The refinement primarily consists of lowering the constants on the bound and changing the noise definition.

In order for any Unnormalized Winnow bound to apply, the α and σ parameter need

to be set appropriately. For most problems, the number of attributes in an instance is fixed, so one can set $\sigma = 1/n$. The α parameter is more difficult. For most problems we do not know the size of δ, therefore we may choose a sub-optimal multiplier. This can greatly increase the number of mistakes made by the algorithm. We consider possible ways to set α when we talk about parameter selection in Section 2.5.

A general problem with Winnow algorithms is that they can only represent positive weights because the weights start out positive and are always multiplied by a positive number. Fortunately there is a simple solution for learning negative weights. For every attribute x_i, we add the complemented attribute $\bar{x}_i = 1 - x_i$. This complemented attribute allows the algorithm to use positive weight to create negative weights [Lit88]. Let w_i be the weight on attribute x_i and w_i^c be the weight on attribute $\bar{x}_i$. If w_i is 0 then any weight $w_i^c > 0$ effectively becomes a negative weight for x_i. Therefore anytime we need a negative weight for a target function, we just use 0 weight for the normal attribute and place all the weight on the complemented attribute. The negative weight technique doubles the number of attributes, but because of the logarithmic nature of the bound it has a small effect on the mistake bound.

This negative weight technique has some beneficial consequences. Because of the 1 term in the complemented attribute, the target function gets some extra constant weight. This extra weight is useful because it allows the target to represent a negative threshold. For example, it allows the algorithm to represent a target function that always predicts 1. In this case, just set the weight on all attributes, including the complements, to $2/n$. This creates a target function that always predicts 1 with a mistake-bound of at most $7.63 + 2.2N$.

2.3.2 Complemented Unnormalized Winnow Algorithm

The Complemented Unnormalized Winnow Algorithm is a slight variation of the Unnormalized Winnow Algorithm. It was originally mentioned in [Lit88] as a technique to convert a target function consisting of a conjunction of attributes into a disjunction of attributes. As we will see, this significantly lowers the mistake-bound on conjunctions. We give the code for Complemented Unnormalized Winnow in Figure 2.5.

Complemented Unnormalized Winnow(α, σ)

Parameters

$\alpha > 1$ is the update multiplier.

$\sigma > 0$ is the initial weight value.

Initialization

$t \leftarrow 1$ is the current trial.

$\forall i \in \{1, \ldots, n\}$ $w_{i,1} = \sigma$ are the weights.

Trials

Instance: $\mathbf{x}_t \in [0,1]^n$. Let $\bar{x}_{i,t} = 1 - x_{i,t}$

Prediction: If $\mathbf{w_t} \cdot \bar{\mathbf{x}}_\mathbf{t} \geq 1$

predict $\hat{y}_t = -1$ else predict $\hat{y}_t = 1$.

Update: Let $y_t \in \{-1, 1\}$ be the correct label and $\bar{y}_t = -y_t$.

If $\bar{y}_t(\mathbf{w_t} \cdot \bar{\mathbf{x}}_\mathbf{t}) \leq 0$ then

$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = \alpha^{\bar{y}_t \bar{x}_{i,t}} w_{i,t}$.

Else

$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = w_{i,t}$.

$t \leftarrow t + 1$.

Figure 2.5: Pseudo-code for the Complemented Unnormalized Winnow algorithm.

Complemented Unnormalized Winnow uses the same noise function as Unnormalized Winnow. Let the target weights $u_i \geq 0$. Let $\delta \in (0, 1]$ be the margin. The noise on instance $\mathbf{x_t}$ is defined as $\nu_t = \max(0, \delta - y_t(\mathbf{u} \cdot \mathbf{x_t} - 1))$, and the total noise is $N = \sum_{t=1}^{T} \nu_t$.

To help motivate the complemented algorithm, it is useful to understand how Unnormalized Winnow behaves with conjunctions and disjunctions. Given our notation, if the target function is composed of a disjunction of k literals then the target weights can be set to 2. This allows us to set $\delta = 1$; its maximum value. If the target function is composed of a conjunction of k literals then each target weight can be set to $2/(2k-1)$ and $\delta = 1/k$. According to Theorem 2.5, this gives disjunctions a mistake-bound of roughly $6k \ln n$ and conjunctions a mistake-bound of roughly $2k^2 \ln n$. However, there is not much difference between a conjunction and a disjunction. The only difference is whether -1 or 1 is called true.

Therefore, we can modify any algorithm to learn conjunctions as if they were disjunctions. All we need to do is flip the values of all the attributes and labels. For attributes, every 0 becomes a 1 and every 1 becomes a 0. For labels, the new label is

just the negative of the old label. This new problem turns a conjunction into a disjunction. To get the answer to the original problem, we just run the algorithm on the new problem and flip the label of the algorithm's prediction. [Lit88]

We can generalize this technique to more than just conjunctions. Recall the definition of a complemented attribute is $\bar{x_i} = 1 - x_i$. This flips a $\{0, 1\}$ attribute, but can also handle attributes in $[0, 1]$ [Lit89]. Define the flipping of a label as $\tilde{y} = -y$. The Complemented Unnormalized Winnow algorithm is the same algorithm as Winnow except that every attribute is complemented and every label is flipped before they are accepted as input by the algorithm. This allows the algorithm to work on a different target function where the -1 and 1 labels are flipped. When the algorithm returns a prediction, the label must be flipped again to return to the label syntax of the environment.

Before we give the mistake-bound for Complemented Unnormalized Winnow, we want to simplify its statement with some new notation. Let $h = (\sum_{i=1}^{n} u_i) - 1$. We assume $h > 0$ because a target function with $h \leq 0$ must always predict -1. This causes the complemented problem to always predict 1. This is a special case that we mentioned in the section on Unnormalized Winnow. It can be handled separately and gives a mistake-bound of at most $7.63 + 2.2N$. Also notice that because $\sum_{i=1}^{n} u_i \geq 1 + \delta$, $h \geq \delta$.

Corollary 2.6 *Assume* $h = (\sum_{i=1}^{n} u_i) - 1 > 0$. *The number of mistakes made by Complemented Unnormalized Winnow when* $\sigma = 1/n$ *and* $\alpha = 1 + \frac{\delta}{(\sum_{i=1}^{n} u_i) - 1}$ *is at most*

$$\frac{(2h + \delta)\left(\sum_{i=1}^{n} u_i \ln u_i - \ln(h) \sum_{i=1}^{n} u_i + \ln(n) \sum_{i=1}^{n} u_i\right)}{\delta^2} + \frac{(2 + \frac{\delta^2}{5h^2})N}{\delta}.$$

Proof Our goal is to manipulate the noise function into a form that corresponds to a related but different target function that works with the complimented attributes.

$$\begin{aligned} \nu_t &= \max(0, \delta - y_t(\mathbf{u} \cdot \mathbf{x_t} - 1)) \\ &= \max(0, \delta + y_t(\sum_{i=1}^{n} u_i - \mathbf{u} \cdot \mathbf{x_t} + 1 - \sum_{i=1}^{n} u_i)) \\ &= \max(0, \delta + y_t(\sum_{i=1}^{n} u_i(1 - x_i) - h)) \end{aligned}$$

$$= \max(0, \delta - \tilde{y}_t(\mathbf{u} \cdot \bar{\mathbf{x}} - h)).$$

This is close to a legal noise function for the Unnormalized Winnow algorithm. We only need to make the threshold term 1 by dividing by h.

$$\frac{\nu_t}{h} = \max(0, \frac{\delta}{h} - \tilde{y}_t(\frac{\mathbf{u} \cdot \bar{\mathbf{x}}}{h} - 1)).$$

This is a noise function that can be used by Unnormalized Winnow with complemented attributes. Let $\mathbf{u}^c$ be the target weights and δ^c be the margin for this noise function. Let N^c be the sum of the noise term over all the trials. Therefore $u_i^c = u_i/h$, $\delta^c = \delta/h$, and $N^c = N/h$.

We can use these instances on the Unnormalized Winnow algorithm with $\alpha = 1+\delta^c$ and $\sigma = 1/n$. However, we are learning the flipped label. Therefore, every time the Unnormalized Winnow returns a label y_t, the complemented algorithm returns -y_t. This algorithm only makes a mistake if the Unnormalized Winnow algorithm makes a mistake on a flipped label. Using Theorem 2.5 with this noise function and these parameters gives an upper-bound on the number of mistakes. This proves the theorem. ■

The $(2h+\delta)$ factor is the primary difference between the mistake-bounds of Complemented Unnormalized Winnow and Unnormalized Winnow. The Unnormalized Winnow algorithm has a similar factor of $(2+\delta)$. For h values close to 0, the Complemented Unnormalized algorithm has a clear advantage. For $h = 1$ the bounds are similar. For large h, the Unnormalized Winnow should be superior. Notice that our previous example of a disjunction with k literals has $h = 2k - 1$, and a conjunction with k literals has $h = \frac{1}{2k-1}$.

Complemented Unnormalized Winnow suffers from the same strengths and weakness as Unnormalized Winnow. The α and σ parameters still need to be set appropriately. Again, setting $\sigma = 1/n$ is a good choice, but see Appendix A for more possibilities. Setting α is more difficult because for most problems one does not know δ. See Section 2.5 for information on how to deal with parameter selection. Also negative weights are only possible if the complement of the attributes are included. In this case, the

complements are the original attributes. Therefore, to deal with positive and negative weights both the complemented and normal attributes must be used by the algorithm. In this case, the Complemented Unnormalized Winnow only needs to implement the label flipping.

2.3.3 Normalized Winnow Algorithm

The Normalized Winnow algorithm was first proven to learn linear-threshold functions in [Lit89]. It is similar to the Unnormalized Winnow algorithm except that it effectively forces the weights of the algorithm to always sum to 1. The Normalized Winnow algorithm was originally called the Weighted Majority Algorithm (WMA) since it is almost identical to the original WMA [LW94]. The only difference is that Normalized Winnow only updates on mistakes while WMA can update on all trials. We will talk about WMA more extensively in Section 2.5 when we talk about parameter selection.

In this thesis, we give a more general form of Normalized Winnow. It uses an extra parameter θ that controls the target function threshold. The original version of Normalized Winnow has $\theta = 1/2$. This more general algorithm was originally designed by Nick Littlestone but never published [Lit94]. We give the code for Normalized Winnow in Figure 2.6. Notice that one can normalize the weights to sum to 1 at the end of any trial without affecting the behavior of the algorithm.

Let $u_i \geq 0$ be the weights for the target function where $\sum_{i=1}^{n} u_i = 1$. Let $\tau = \min(\theta, 1-\theta)$ and let $\delta \in (0, \tau]$ be the margin. The noise on instance $\mathbf{x_t}$ is defined as $\nu_t = \max(0, \delta - y_t(\mathbf{u} \cdot \mathbf{x_t} - \theta))$. Therefore, the hyperplane defined by this noise function is $\mathbf{u} \cdot \mathbf{x_t} - \theta$.

Theorem 2.7 *When* $\alpha = \min(6, 1 + \frac{\delta}{\tau(1-\tau-\delta)})$*, the number of mistakes made by Normalized Winnow is at most*

$$\frac{2(\theta(1-\theta) + \delta|1-2\theta|)(\ln n + \sum_{i=1}^{n} u_i \ln u_i)}{\delta^2} + \frac{2.8N}{\delta}.$$

The original proof for this algorithm is by Nick Littlestone [Lit94]. However, his proof did not clarify the improvements of this algorithm over Normalized Winnow when

Normalized Winnow(α, θ)
Parameters
$\alpha > 1$ is the update multiplier.
$0 < \theta < 1$ is the threshold.
Initialization
$t \leftarrow 1$ is the current trial.
$\forall i \in \{1, \ldots, n\}$ $w_{i,1} = 1$ are the weights.
Trials
Instance: $\mathbf{x}_t \in [0,1]^n$.
Prediction: If $\mathbf{w_t} \cdot \mathbf{x_t} / \sum_{i=1}^{n} w_{i,t} \geq \theta$
predict $\hat{y}_t = 1$ else predict $\hat{y}_t = -1$.
Update: Let $y_t \in \{-1, 1\}$ be the correct label.
If $y_t(\mathbf{w_t} \cdot \mathbf{x_t}) \leq 0$ then
$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = \alpha^{y_t x_{i,t}} w_{i,t}$.
Else
$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = w_{i,t}$.
$t \leftarrow t + 1$.

Figure 2.6: Pseudo-code for the Normalized Winnow algorithm.

$\theta \neq 1/2$. In Appendix B, we prove Theorem 2.7 along with some slightly improved but more complicated bounds. These proofs are based on joint work done with Nick Littlestone. Proofs for normalized Winnow with a threshold have since appeared in other publications. In [HPW99], Normalized Winnow is presented, and mistake bounds are given for learning disjunctions. Our bound is more general since it allows arbitrary linear-threshold functions.

As before, Normalized Winnow handles only positive weights. The solution is the same as with Unnormalized Winnow. We use complemented attributes, $1 - x_i$ to handle negative weights; this will at most double the number of attributes.

Another problem with Normalized Winnow is that θ may not be the correct value for the concept. Assume the concept is described by a particular hyperplane. The hyperplane can be represented with a vector of weights for the attributes and the appropriate threshold. We can multiply all these values by the same constant and still have the same hyperplane. If we divide all the values by the sum of the weights, the new weights sum to 1 and the threshold is a positive constant c. This matches the target function required by Normalized Winnow when $\theta = c$. When $\theta \neq c$, we need to use

extra attributes to correct the θ value. If θ is too small, we need to use an attribute that is always 0. Any weight on this attribute will effectively lower the one-norm of the other target weights. This will effectively increase the value of θ. If θ is too large, we need to use an attribute that is always 1. The weight of this attribute can be subtracted from θ to lower its value. Therefore, we need to add two attributes to learn the threshold.

Just as with Unnormalized Winnow, we need to set parameters correctly for the bounds to apply. In this case, both parameters, α and θ, can be difficult to set. The same problems we had with α in Unnormalized Winnow occur here. With θ, we show in Section 2.4 that the bound is optimal when $\theta = c$ and gets much worse the further θ is from c. We talk about how to solve these problems in Section 2.5.

2.3.4 Balanced Winnow Algorithm

The Balanced Winnow algorithm first appeared in [Lit89]. It gives another variation on the Winnow family of algorithms. Balanced Winnow keeps track of two vectors of weights, one for label 1 and the other for label -1. Whichever vector of weights has the larger dot product with the attributes determines the prediction label. It is interesting to note that the Balanced Winnow algorithm is identical to the Normalized Winnow algorithm using $\theta = 1/2$, complemented attributes, and transforming all the $[0, 1]$ attributes to $[1/2, 1]$ using the linear transformation $(x_i + 1)/2$ [GLS01]. We give the code for Balanced Winnow in Figure 2.7.

Let $\mathbf{u}^+$ and $\mathbf{u}^-$ be the weights for the target function where each $u_i^+ \geq 0$, each $u_i^- \geq 0$ and $\sum_{i=1}^{n} \left(u_i^+ + u_i^-\right) = 1$. The noise on instance x_t is defined as $\nu_t = \max(0, \delta - y_t(\mathbf{u}^+ \cdot \mathbf{x_t} - \mathbf{u}^- \cdot \mathbf{x_t})$ where $\delta \in (0, 1/2]$ is the margin. This noise function is based on the $(\mathbf{u}^+ - \mathbf{u}^-) \cdot \mathbf{x}_t$ hyperplane.

Theorem 2.8 *If $\alpha = \sqrt{\frac{1+\delta}{1-\delta}}$ then the number of mistakes made by Balanced Winnow is at most*

$$\frac{2 \ln 2n + 2 \sum_{i=1}^{n} (u_i^+ \ln u_i^+ + u_i^- \ln u_i^-)}{\delta^2} + \frac{2\left(1 + \frac{2\delta^2}{5(1-\delta)^2}\right)}{\delta} N.$$

A proof of this result is found in Appendix C. The proof is almost identical to a proof presented in [Lit89] with the exception of a more modern type of noise analysis.

Balanced Winnow(α)
Parameters
$\alpha > 1$ is the update multiplier.
Initialization
$t \leftarrow 1$ is the current trial.
$\forall i \in \{1, \ldots, n\}$ $w^+_{i,1} = 1$ are the positive weights.
$\forall i \in \{1, \ldots, n\}$ $w^-_{i,1} = 1$ are the negative weights.
Trials
Instance: $\mathbf{x}_t \in [0,1]^n$.
Prediction: If $\mathbf{w}^+_\mathbf{t} \cdot \mathbf{x_t} \geq \mathbf{w}^-_\mathbf{t} \cdot \mathbf{x_t}$
predict $\hat{y}_t = 1$ else predict $\hat{y}_t = -1$.
Update: Let $y_t \in \{-1, 1\}$ be the correct label.
If $y_t(\mathbf{w}^+_\mathbf{t} \cdot \mathbf{x_t} - \mathbf{w}^-_\mathbf{t} \cdot \mathbf{x_t}) \leq 0$
$\forall i \in \{1, \ldots, n\}$ $w^+_{i,t+1} = \alpha^{y_t x_{i,t}} w^+_{i,t}$ and $w^-_{i,t+1} = \alpha^{-y_t x_{i,t}} w^-_{i,t}$.
Else
$\forall i \in \{1, \ldots, n\}$ $w^+_{i,t+1} = w^+_{i,t}$ and $w^-_{i,t+1} = w^-_{i,t}$.
$t \leftarrow t + 1$.

Figure 2.7: Pseudo-code for the Balanced Winnow algorithm.

One advantage of Balanced Winnow is that complemented attributes are not needed. Negative weights are created with the $\mathbf{w}^-_t$ weights. However a constant 1 attribute is still needed to create a threshold. Another much more subtle advantage involves the proof technique. We show in Appendix C that the proof technique is not tight. Therefore, we could get better than expected performance for certain types of problems.

The main difficulty of Balanced Winnow, just like the other Winnow algorithms, is the setting of the α parameter. In Section 2.5, we address the issue of parameter selection.

2.3.5 ALMA Algorithm

The ALMA algorithm was originally published in [Gen01]. It is based on a generalization of Perceptron and Normalized Winnow called a p-norm algorithm [GLS01]. The algorithm includes a parameter p that allows it to perform like the Perceptron ($p = 2$), a Winnow ($p = O(\log n)$) algorithm, or something in between. While it has several parameters, part of the algorithm's appeal is that these parameters do not have to depend on the target concept to achieve a good mistake bound. We give the code for

ALMA(p, B, C)

Parameters

$B \geq 0$ controls the algorithm margin.

$C > 0$.

$p \geq 2$ and $q = \frac{p}{p-1}$ control the norms.

Initialization

$t \leftarrow 1$ is the current trial.

$\forall i \in \{1, \ldots, n\}$ $w_{i,1} = 0$ are the algorithm weights.

$k = 0$ is the number of updates.

Trials

Instance: $\mathbf{x}_t \in \mathbf{R}^n$.

Prediction: If $\mathbf{w_t} \cdot \mathbf{x_t} \geq 0$

predict $\hat{y}_t = 1$ else predict $\hat{y}_t = -1$.

Update: Let $y_t \in \{-1, 1\}$ be the correct label and $\hat{\delta} = B\|\mathbf{x_t}\|_p \sqrt{\frac{p-1}{k}}$.

If $y_t(\mathbf{w_t} \cdot \mathbf{x_t}) \leq \hat{\delta}$.

Let $\eta = \dfrac{C}{\sqrt{k(p-1)}\|\mathbf{x_t}\|_p}$

Let $f(w_i, \mathbf{w}) = sign(w_i)|w_i|^{q-1}/\|\mathbf{w}\|_q^{q-2}$.

Let $f^{-1}(z_i, \mathbf{z}) = sign(z_i)|z_i|^{p-1}/\|\mathbf{z}\|_p^{p-2}$.

$\forall i \in \{1, \ldots, n\}$ $z_{i,t} = f(w_{i,t}, \mathbf{w_t}) + \eta y_t x_{i,t}$.

$\forall i \in \{1, \ldots, n\}$ $w'_{i,t} = f^{-1}(z_{i,t}, \mathbf{z_t})$.

$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = w'_{i,t} / \max(1, \|\mathbf{w'_t}\|_q)$.

$k \leftarrow k + 1$.

Else

$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = w_{i,t}$.

$t \leftarrow t + 1$.

Figure 2.8: Pseudo-code for the ALMA algorithm.

ALMA in Figure 2.8.

The ALMA algorithm works by computing a vector $\mathbf{z}_t$. This vector is roughly equivalent to the weights used by the Perceptron algorithm. ALMA changes the influence of the individual weights by computing weights proportional to $z_{i,t}^{p-1}$. When $p = 2$ this gives Perceptron like weights. As p gets larger the influence of larger z values increases in a way that is similar to the multiplicative updates used by Winnow algorithms. These updates are further refined by allowing η to control the size of each $\mathbf{z}$ change and by a normalization step that is based on the size of the weights. These additions allow the algorithm to adapt the size of the updates during learning.

Let $\mathbf{u}$ be the vector of target weights where each $u_i \in R$, let $\omega = \max_{i \in T} \|\mathbf{x_i}\|_p$, and

let $\delta > 0$ be the margin. The noise for instance $\mathbf{x_t}$ is defined as $\nu_t = \max(0, \delta - \omega y_t \mathbf{u} \cdot \mathbf{x_t}/\|\mathbf{x_t}\|_p)$. This unusual definition of noise is required since the algorithm effectively normalizes every instance vector so that its p-norm is 1. The hyperplane for this noise function is still $\mathbf{u} \cdot \mathbf{x_t}$. In Section 2.4.1, where we compare the various algorithms, we will see this definition of noise can be exploited by an adversary to increase the number of mistakes.

The main term in the mistake bound is $G = (p-1)\omega^2\|\mathbf{u}\|_q^2/\delta^2$. This term is similar to the bound for the Perceptron when $p = 2$ and similar to the bounds on the Winnow algorithms when $p = \ln n$. However, the full bound is more complicated.

Theorem 2.9 (Gentile, 2001) *The number of mistakes made by ALMA(p) when $C = \sqrt{B^2+1} - B$ is at most*

$$\frac{G}{2C^2} + \sqrt{\frac{G^2}{4C^4} + \frac{G}{C^2} + \frac{GN}{\delta C^2}} + \frac{N}{\delta}.$$

This bound is in a slightly different form, but it is equivalent to the bound first published in [Gen01]. First, we have simplified their bounds by removing a parameter. This parameter was only useful for improving the bound when there are no noisy instances. Second we have moved some factors around to make it easier to compare to the other algorithms.

In this thesis, we always set $B = 0$ and $C = 1$. This optimizes the bound in Theorem 2.9 and corresponds to making ALMA mistake-driven. When $B > 0$, ALMA performs updates on instances that are correctly predicted but close to the current algorithm hyperplane. This is a technique that has been used in the past to improve the performance of linear-threshold learning algorithms [Ros62, LL00]. Unfortunately, this can not improve the performance against adversaries.[7] However, the technique might be effective when the instances are generated by something weaker than an adversary. ALMA has a sophisticated way of using these extra updates to approximate a hyperplane that maximally separates the data in a way similar to Support Vector Machines [CV95]. While this technique seems promising, we do not explore these extra updates in this thesis.

[7] See Appendix F for details.

A large advantage of this algorithm is that the parameters are easy to set. To get a Perceptron like bound set $p = 2$; to get a Winnow like bound set $p = \ln n$. Notice that for the Winnow case, no multiplication parameter is necessary. The only variable the algorithm needs is n which is available for most problems. In chapter 6, we will use a slightly modified form of the ALMA algorithm that allows learning with shifting target concepts [KSW02]. This form is identical except that it removes parameters B and C and instead gives η and $\hat{\delta}$ fixed values. Unfortunately, its mistake bound will depend on setting these parameters correctly.

2.4 Mistake bound Comparison

One problem with the previously published mistake bounds of linear-threshold algorithms is that they use a similar notation, but the notation often has a slightly different meaning for each algorithm. The main difference is the target weights. The algorithms need different target weights because some algorithms can not directly represent negative weights, and some algorithms can not directly represent the correct threshold. These differences change the value of δ, ν_t, and the number of attributes.

In this section, we give a standard form of target function. We use this target function to give comparable mistake bounds for each algorithm. To simplify the analysis, we often remove some negative terms from the bounds. This means the upper-bounds are not as tight as possible. Refer to earlier in this chapter or the appendices to tighten the mistake bounds. After we give the mistake bounds, we will comment on some of the similarities and differences between the bounds of the algorithms.

One difficulty with a standard target function is that the target function that gives the best mistake-bound for one algorithm may not be the best choice for another algorithm. For example, a learning problem might have two sets of attributes that independently give enough information to determine the concept. One set of attributes may give a better mistake bound for Normalized Winnow, the other set may give a better mistake bound for Perceptron. To avoid this problem, we restrict our analysis to the comparison of algorithm mistake bounds when given a particular target function and

margin.

2.4.1 Standard Target Function

We use the following notation for the standard target function. Let $\mathbf{x} \in [0,1]^n$ be the instance vector.[8] Assume that all target weights are non-negative and that $\sum_{i=1}^{n}(u_i + u_i^c) = 1$ where at most one of u_i and u_i^c is positive. Let $c \in (0,1)$ be the threshold and $\delta \in [0, c]$ be the margin. We define the standard target function as

$$\text{Predict 1 if } \sum_{i=1}^{n} u_i x_{i,t} + \sum_{i=1}^{n} u_i^c (1 - x_{i,t}) \geq c + \delta;$$

$$\text{Predict -1 if } \sum_{i=1}^{n} u_i x_{i,t} + \sum_{i=1}^{n} u_i^c (1 - x_{i,t}) \leq c - \delta.$$

Including the complemented attributes, there are a total of $2n$ attributes. The noise function is derived directly from the target function. In the case of the standard target function, $\nu_t = \max(0, \delta - y_t(\sum_{i=1}^{n} u_i x_{i,t} + \sum_{i=1}^{n} u_i^c \bar{x}_{i,t} - c))$.

Our goal is to express the bounds of all the algorithms covered in this chapter in terms of this notation. Several simple operations on a target function preserve the target function's hyperplane and have easily calculated effects on the noise function and other parameters of the target.

We consider two operations. First, we multiply both sides of the prediction inequalities by a positive constant k_1. The purpose of this operation is to renormalize the weights.

$$\text{Predict 1 if } k_1 \left(\sum_{i=1}^{n} u_i x_{i,t} + \sum_{i=1}^{n} u_i^c (1 - x_{i,t}) \right) \geq k_1 c + k_1 \delta;$$

$$\text{Predict -1 if } k_1 \left(\sum_{i=1}^{n} u_i x_{i,t} + \sum_{i=1}^{n} u_i^c (1 - x_{i,t}) \right) \leq k_1 c - k_1 \delta.$$

This new target function has a margin of $k_1\delta$ and new target weights of $\mathbf{u}/k_1$ and $\mathbf{u}^c/k_1$. The new noise function is ν_t/k_1. The number of attributes stays the same.

---

[8] Other instance representations could be beneficial for the Perceptron and ALMA algorithms. While we do not explore this issue in our definition of a standard target function, more information can be found at the end of Section 2.2.

Second, we add a value k_2 to both sides of the target function. The purpose of this operation is to change the algorithm's threshold.

$$\text{Predict 1 if } \; k_2 + \sum_{i=1}^{n} u_i x_{i,t} + \sum_{i=1}^{n} u_i^c (1 - x_{i,t}) \geq k_2 + c + \delta;$$

$$\text{Predict -1 if } \; k_2 + \sum_{i=1}^{n} u_i x_{i,t} + \sum_{i=1}^{n} u_i^c (1 - x_{i,t}) \leq k_2 + c - \delta.$$

This new target function gives the same margin and the same noise function but changes the threshold. It can be created by using an attribute that is always 1, but the specific details depend on the algorithm.

Notice that these two operations behave the same on both prediction inequalities. Therefore, we only consider the inequality $\sum_{i=1}^{n} u_i x_{i,t} + \sum_{i=1}^{n} u_i^c \bar{x}_{i,t} \geq c + \delta$ for the target functions. The operations on the other inequality are symmetric.

Unnormalized Winnow

We start with the standard target function and show how to modify it into a form used by Unnormalized Winnow. The standard target function predicts 1 if

$$\sum_{i=1}^{n} u_i x_{i,t} + \sum_{i=1}^{n} u_i^c (1 - x_{i,t}) \geq c + \delta.$$

Dividing both sides by c gives

$$\frac{\sum_{i=1}^{n} u_i x_{i,t} + \sum_{i=1}^{n} u_i^c (1 - x_{i,t})}{c} \geq 1 + \frac{\delta}{c}.$$

This target function can be used by Unnormalized Winnow and Complemented Unnormalized Winnow.

Using Theorem 2.5, the number of mistakes made by Unnormalized Winnow is at most

$$\frac{(2c + \delta) \ln \left(\frac{2n}{c}\right)}{\delta^2} + \frac{2.2N}{\delta}.$$

Using Corollary 2.6, the number of mistakes made by Complemented Unnormalized Winnow is at most

$$\frac{(2(1 - c) + \delta) \ln \left(\frac{2n}{1-c}\right)}{\delta^2} + \frac{2.2N}{\delta}.$$

Notice that the noise term of Unnormalized Winnow is the same as with the original representation. This is common in the conversions because the operations on the target functions usually change N and δ by the same amount.

Normalized Winnow

We start with the standard target function and show how to modify it into a form used by Unnormalized Winnow. The standard target function predicts 1 if

$$\sum_{i=1}^{n} u_i x_{i,t} + \sum_{i=1}^{n} u_i^c (1 - x_{i,t}) \geq c + \delta.$$

When $\theta = c$, the target function is in a form that can be used by Normalized Winnow. If $\theta \neq c$, we can get a usable target function by putting target weight on either a constant 0 or constant 1 attribute.

If $\theta \leq c$ then multiply the inequality by θ/c to get

$$\frac{\theta}{c}\left(\sum_{i=1}^{n} u_i x_{i,t} + \sum_{i=1}^{n} u_i^c (1 - x_{i,t})\right) \geq \theta + \frac{\theta}{c}\delta.$$

This is close to an acceptable target function, but we need the weights to sum to 1. Therefore set the weight of an attribute that is always 0 to $(c - \theta)/c$. This creates a target function that is usable by Normalized Winnow. Applying Theorem 2.7, the number of mistakes made by Normalized Winnow is at most

$$\frac{2(\frac{c}{\theta})^2 \left(\theta(1-\theta) + \frac{\theta\delta|1-2\theta|}{c}\right) \ln(2n+2)}{\delta^2} + \frac{2.8N}{\delta}$$

If $\theta \geq c$ then multiply the inequality by $(1 - \theta)/(1 - c)$ to get

$$\frac{1-\theta}{1-c}\left(\sum_{i=1}^{n} u_i x_{i,t} + \sum_{i=1}^{n} u_i^c (1 - x_{i,t})\right) \geq \frac{c(1-\theta)}{1-c} + \frac{1-\theta}{1-c}\delta.$$

The target weights need to sum to 1 so add $(\theta - c)/(1 - c)$ to both sides. This gives us

$$\frac{1-\theta}{1-c}\left(\sum_{i=1}^{n} u_i x_{i,t} + \sum_{i=1}^{n} u_i^c (1 - x_{i,t})\right) + \frac{\theta - c}{1-c} \geq \theta + \frac{1-\theta}{1-c}\delta.$$

This creates a target function that is usable by Normalized Winnow. Applying Theorem 2.7, the number of mistakes made by Normalized Winnow is at most

$$\frac{2(\frac{1-c}{1-\theta})^2 \left(\theta(1-\theta) + \frac{(1-\theta)\delta|1-2\theta|}{1-c}\right) \ln(2n+2)}{\delta^2} + \frac{2.8N}{\delta}$$

We can combine the previous results by defining $H = \max\left(\frac{c}{\theta}, \frac{1-c}{1-\theta}\right)$. This gives an upper-bound of

$$\frac{2H^2\left(\theta(1-\theta) + \frac{\delta|1-2\theta|}{H}\right)\ln(2n+2)}{\delta^2} + \frac{2.8N}{\delta}$$

Balanced

The standard target function predicts 1 if

$$\sum_{i=1}^{n} u_i x_{i,t} + \sum_{i=1}^{n} u_i^c(1 - x_{i,t}) \geq c + \delta.$$

To convert the standard target function to the form used by Balanced, we need to uncomplement the attributes.

$$\sum_{i=1}^{n} u_i x_{i,t} - \sum_{i=1}^{n} u_i^c x_{i,t} + \sum_{i=1}^{n} u_i^c - c \geq \delta.$$

The term $\sum_{i=1}^{n} u_i^c - c$ can be represented by the weight on an extra attribute that is always 1. If this term is positive, it can represented by a target weight associated with $\mathbf{u}$ otherwise it can represented in $\mathbf{u}^c$. However, the target weights do not sum to 1 with this extra target weight. In order for the bounds to apply, we need to divide the target by $(1 + |c - \sum_{i=1}^{n} u_i^c|)$. This gives a target that predicts 1 when

$$\frac{\sum_{i=1}^{n} u_i x_{i,t} - \sum_{i=1}^{n} u_i^c x_{i,t}}{(1 + |c - \sum_{i=1}^{n} u_i^c|)} - \frac{c - \sum_{i=1}^{n} u_i^c}{1 + |c - \sum_{i=1}^{n} u_i^c|} \geq \frac{\delta}{1 + |c - \sum_{i=1}^{n} u_i^c|}.$$

This creates a target function that is usable by Balanced Winnow. Applying Theorem 2.8, the number of mistakes made by Balanced Winnow is at most

$$\frac{2(1 + |c - \sum_{i=1}^{n} u_i^c|)^2 \ln(2n+2)}{\delta^2} + \frac{2.8N}{\delta}.$$

Perceptron

The target function for Balanced Winnow also works for Perceptron. The only difference is that Perceptron can represent negative target weights, so it does not explicitly need $\mathbf{u}^c$. Instead Perceptron uses $\mathbf{u} - \mathbf{u}^c$ as the target weights.

Following the same manipulations as Balanced Winnow we convert our standard target function to

$$\sum_{i=1}^{n}(u_i - u_i^c)x_{i,t} + \sum_{i=1}^{n} u_i^c - c \geq \delta.$$

At this point, we could apply the mistake bound from Section 2.2, since there is no requirement to normalize the target weights. However, we find it easier to compare the algorithms if we normalize the target weights so that their absolute value sums to 1. This new target function predicts 1 when

$$\frac{\sum_{i=1}^{n}(u_i - u_i^c)x_{i,t}}{(1 + |c - \sum_{i=1}^{n} u_i^c|)} - \frac{c - \sum_{i=1}^{n} u_i^c}{1 + |c - \sum_{i=1}^{n} u_i^c|} \geq \frac{\delta}{1 + |c - \sum_{i=1}^{n} u_i^c|}.$$

The Perceptron algorithm mistake bound explicitly involves the target weights and the instances, so we need some new notation. Let $\hat{u_i} = (u_i - u_i^c)/\left(1 + |c - \sum_{i=1}^{n} u_i^c|\right)$, and for the threshold, let $\hat{u}_{n+1} = (\sum_{i=1}^{n} u_i^c - c)/\left(1 + |c - \sum_{i=1}^{n} u_i^c|\right)$. Let $\hat{x}_{i,t} = x_{i,t}$ and $\hat{x}_{n+1,t} = 1$. This adds the attribute that is always 1. Based on Theorem 2.3, and defining $\hat{\beta}^2 = \sum_{t \in M_T} \|\hat{\mathbf{x}}_\mathbf{t}\|_2^2 / |M_T|$, the number of mistakes made by Perceptron is at most

$$\frac{(1 + |c - \sum_{i=1}^{n} u_i^c|)^2 \hat{\beta}^2 \|\hat{\mathbf{u}}\|_2^2}{\delta^2} + \frac{2N}{\delta}.$$

ALMA

The target function for ALMA is the same as Perceptron; predict 1 when

$$\frac{\sum_{i=1}^{n}(u_i - u_i^c)x_{i,t}}{(1 + |c - \sum_{i=1}^{n} u_i^c|)} - \frac{c - \sum_{i=1}^{n} u_i^c}{1 + |c - \sum_{i=1}^{n} u_i^c|} \geq \frac{\delta}{1 + |c - \sum_{i=1}^{n} u_i^c|}.$$

We normalized the target weights to simplify comparison with other algorithms. Let $\hat{u_i} = (u_i - u_i^c)/\left(1 + |c - \sum_{i=1}^{n} u_i^c|\right)$, and for the threshold, let $\hat{u}_{n+1} = (\sum_{i=1}^{n} u_i^c - c)/\left(1 + |c - \sum_{i=1}^{n} u_i^c|\right)$. Let $\hat{x}_{i,t} = x_{i,t}$ and $\hat{x}_{n+1,t} = 1$. This adds an attribute that is always 1. Using this notation, the target function is predict 1 if

$$\sum_{i=1}^{n} \hat{u}_i \hat{x}_{i,t} \geq \frac{\delta}{1 + |c - \sum_{i=1}^{n} u_i^c|}.$$

The difficulty for ALMA comes from the unusual definition of noise.[9] Let $\omega = \max_{t \in T} \|\hat{\mathbf{x}}_\mathbf{t}\|_p$. The noise for ALMA is defined as

$$\nu_t(ALMA) = \max\left[0, \frac{\delta}{1 + |c - \sum_{i=1}^{n} u_i^c|} - y_t \frac{\omega(\hat{\mathbf{u}} \cdot \hat{\mathbf{x}}_\mathbf{t})}{\|\hat{\mathbf{x}}_\mathbf{t}\|_p}\right].$$

[9]The definition of noise for the other algorithms seems more natural because the amount of noise is proportional to the amount the relevant attributes are perturbed.

For any instance where $y_t \hat{\mathbf{u}} \cdot \hat{\mathbf{x}}_\mathbf{t} \geq 0$, this noise is maximized when $\|\hat{\mathbf{x}}_\mathbf{t}\|_p = \omega$. For these instances, $\nu_t(ALMA) \leq (1 + |c - \sum_{i=1}^n u_i^c|))\nu_t(Std)$, where $\nu_t(Std) = \max(0, \delta - y_t(\sum_{i=1}^n u_i x_{i,t} + \sum_{i=1}^n u_i^c(1 - x_{i,t}) - c))$ is the noise for the standard target function. The noise is worse when $y_t \hat{\mathbf{u}} \cdot \hat{\mathbf{x}}_\mathbf{t} \leq 0$. This corresponds to instances on the wrong side of the target hyperplane.

To help understand the behavior of the noise when $y_t(\hat{\mathbf{u}} \cdot \hat{\mathbf{x}}_\mathbf{t}) \leq 0$ define $A = (1 + |c - \sum_{i=1}^n u_i^c|)$ and $\kappa = \frac{\delta}{\delta + A|\hat{\mathbf{u}} \cdot \hat{\mathbf{x}}_\mathbf{t}|} \in (0, 1]$. Because there is an attribute that is always 1, $\min_{t \in T} \|\hat{\mathbf{x}}_\mathbf{t}\|_p \geq 1$. We can use these terms to bound the change in noise. Based on the above notation, when $y_t \hat{\mathbf{u}} \cdot \hat{\mathbf{x}}_\mathbf{t} \leq 0$,

$$\frac{\nu_t(ALMA)}{\nu_t(Std)} = \frac{1}{A}\left(\frac{\delta + A\frac{\omega}{\|\hat{\mathbf{x}}_\mathbf{t}\|_p}|\hat{\mathbf{u}} \cdot \hat{\mathbf{x}}_\mathbf{t}|}{\delta + A|\hat{\mathbf{u}} \cdot \hat{\mathbf{x}}_\mathbf{t}|}\right) = \frac{\kappa}{A} + \frac{\omega}{\|\hat{\mathbf{x}}_\mathbf{t}\|_p}\frac{(1-\kappa)}{A} < \frac{\omega}{A}.$$

Therefore, the noise in the ALMA algorithm changes by a factor that is an average of $1/A$ and $\omega/(\|\hat{\mathbf{x}}_\mathbf{t}\|_p A)$. The upper-bound of ω/A should be accurate when κ and $\|\hat{\mathbf{x}}\|_p$ are small. The value of κ is small for problems where the adversary creates noisy instances that are far away from the target hyperplane. Whether $\|\hat{\mathbf{x}}\|_p$ can be made much smaller than ω at the same time will depend on the target function.

In order to give the final bound let

$$G = \frac{(1 + |c - \sum_{i=1}^n u_i^c|)^2 (p-1)\omega^2 \|\hat{\mathbf{u}}\|_q^2}{\delta^2} \ .$$

Based on Theorem 2.9, with our target function, the number of mistakes made by ALMA is at most

$$\frac{G}{2} + \sqrt{\frac{G^2}{4} + G + \frac{G\omega N}{\delta}} + \frac{\omega N}{\delta} \ .$$

2.4.2 Algorithm Comparison

In Table 2.1, we give the mistake bounds for the linear-threshold algorithms in this dissertation. Since we have expressed the relevant notation of each algorithm in terms of our standard target function, we can express each bound in that same notation.

To make the bounds easier to interpret, we have approximated some of the bounds by removing lower order terms and/or simplifying the leading constant. As a reminder on notation, c is the threshold of the standard target function. For Normalized Winnow

Unnormalized Winnow	$\frac{(2c+\delta)\ln\left(\frac{2n}{c}\right)}{\delta^2}+\frac{2.2N}{\delta}$
Complemented Unnormalized Winnow	$\frac{(2(1-c)+\delta)\ln\left(\frac{2n}{1-c}\right)}{\delta^2}+\frac{2.2N}{\delta}$
Normalized Winnow	$\frac{2H^2\left(\theta(1-\theta)+\frac{\delta}{H}\right)\ln(2n+2)}{\delta^2}+\frac{2.8N}{\delta}$
Balanced Winnow	$\frac{2(1+\lvert c-\sum_{i=1}^{n}u_i^c\rvert)^2\ln(2n+2)}{\delta^2}+\frac{2.8N}{\delta}$
Perceptron	$\frac{(1+\lvert c-\sum_{i=1}^{n}u_i^c\rvert)^2\hat{\beta}^2\lVert\hat{\mathbf{u}}\rVert_2^2}{\delta^2}+\frac{2N}{\delta}$
ALMA	$\frac{(1+\lvert c-\sum_{i=1}^{n}u_i^c\rvert)^2(p-1)\omega^2\lVert\hat{\mathbf{u}}\rVert_q^2}{\delta^2}+\frac{\omega N}{\delta}$

Table 2.1: Comparison of mistake bounds for linear-threshold algorithms.

let $H=\max(\frac{c}{\theta},\frac{1-c}{1-\theta})$. For Perceptron let $\hat{\beta}^2=\sum_{t\in M_T}\|\hat{\mathbf{x}}_\mathbf{t}\|_2^2/|M_T|$. For ALMA recall that $p\geq 2$, $q=p/(p-1)$, and $\omega=\max_{t\in T}\|\hat{\mathbf{x}}_\mathbf{t}\|_p$. Also, for both Perceptron and ALMA the vector $\hat{\mathbf{u}}$ has a one-norm of 1.

The common element of all these bounds is the dependence of the mistake bounds on $1/\delta^2$ for the term that involves the target, and a dependence of $1/\delta$ for the term that involves the noise. Therefore as the δ term goes to 0, our upper-bounds will go to infinity. Unfortunately, the δ margin can be surprisingly small even for problems with binary attributes. There exist functions for which $1/\delta$ grows exponentially with n. For example,

$$f(x_1,\ldots,x_n)=x_1\vee(x_2\wedge(x_3\vee(x_4\wedge\cdots x_n)\cdots))$$

has a δ of approximately $(1/2)^{n/2}$ [Lit88]. This leads to a poor mistake bound. Fortunately, as we shall see in later chapters, the real-world problems on which we experiment do not exhibit such poor performance. Either they do not have such small δ values, or the effects of small δ is mitigated because the problems are based on a distribution generating the instances and only a small percentage of the instances have a small δ.

Attribute Dependence

Looking at the bounds, we can lump the algorithms roughly into two group. The algorithms that depend on $\ln(n)$ are the Winnow algorithms and ALMA($\ln n$). The algorithms that depend on $\|\mathbf{x_t}\|_2^2$ are Perceptron and ALMA(2). (The two-norm comes from the β and ω definitions.) ALMA has an algorithm in both groups. When $p = 2$ it has a bound that is similar to the Perceptron. When $p = O(\ln(n))$ it has a bound similar to Balanced Winnow. Taking p values in between will give algorithms that interpolate between the two groups of algorithms [Gen03].

For algorithms with $\ln(n)$ in the bound, attributes not in the target function have a small effect on the bound. These are called irrelevant attributes [Lit88]. If we take a learning problem and add n_1 attributes, the mistake bound only increases a small amount because $n + n_1$, the number of attributes, only appears in $\ln(n + n_1)$.[10] This is in comparison to the Perceptron like algorithms that depend on $\|\mathbf{x_t}\|_2^2$. Assuming that the instances are in $[0, 1]^n$, if the instance vectors have a large number of attributes that are 1 then $\|\mathbf{x_t}\|_2^2 \approx n$. Therefore, Winnow algorithms seem to be a better choice when there are a large number of attributes.

There are situations where the $\|\mathbf{x_t}\|_2^2$ algorithms give better mistake bounds. If the instance vectors are sparse on trials with updates then the $\|\mathbf{x_t}\|_2^2$ factor can be much smaller than n. Also the $\|\hat{\mathbf{u}}\|_2^2$ factor can decrease the bound. Remember that $\|\hat{\mathbf{u}}\|_1 = 1$. Therefore, if each of the $\hat{u}_i$ is $1/n$ then $\|\hat{\mathbf{u}}\|_2^2 = 1/n$. This is commonly interpreted to mean that Perceptron does better than Winnow when there are a small number of non-zero attributes and many of the attributes are relevant [Lit88, KW97].

However, a $\ln(n)$ bound seems more practical. For many problems, it is useful to use a large number of speculative attributes. Often, only a few of these attributes are needed to form a good target function, but the extra attributes are included since we do not know the target function ahead of time. This common situation is what makes the bounds for the Winnow algorithms attractive.

---

[10] The mistake bound may decrease if the new attributes can be used to create a target function with a lower mistake bound.

Threshold Dependence

The threshold of the target function has a disproportionate effect on the mistake bound for many of the learning algorithms. It is not just another target weight. At a minimum, many of the algorithms need to learn at least part of the threshold. Perceptron, Normalized Winnow, Balanced Winnow, and ALMA all need to use weight on a constant attribute to learn the threshold. For most of these algorithms, the threshold adds a $(1 + |c - \sum_{i=1}^{n} u_i^c|)^2$ factor to the concept part of the bound. Since $|c - \sum_{i=1}^{n} u_i^c|$ is at most 1, this can quadruple the bound.

The threshold value c has a particularly large effect on the mistake bounds of the Winnow algorithms. Unnormalized Winnow gets an improved bound when the threshold is small because of the $(2c + \delta)$ factor. Complemented Unnormalized Winnow gets an improved bound for c values close to 1 because of the $(2(1 - c) + \delta)$ factor. When $\theta = c$, Normalized Winnow is potentially the best algorithm because of the $2\theta(1 - \theta)$ factor. It performs well for large and small c values. Even when $\theta = c = 1/2$, the Normalized Winnow algorithm has a target function bound of roughly $\frac{\ln(2n)}{2\delta^2}$. The $1/2$ factor on this term gives it an advantage over all the other Winnow algorithms. Unfortunately, as θ gets further away from c, the value of H grows. For Normalized Winnow, the optimal value of θ is c and values away from c cause the mistake bound to grow rapidly.

Balanced Winnow does not seem to be affected by the value of c as much as the other Winnow algorithms. However, as shown in Appendix C, the Balanced Winnow bound is not tight and the algorithm performs better for certain types of problems. In particular, problems with small c values should perform better than indicated in the bounds.

Noise Dependence

Last we want to mention the large similarity between all the parts of the bounds dealing with noise. Most of the algorithms have a noise bound of kN/δ where k is a constant. Therefore, every time a value of δ is added to N, the number of mistakes increases by

k. Put another way, the distance of the noisy instance from the margin determines the amount the instance increases the mistake bound. A natural measure of the distance is in terms of δ. If an instance is δ away then the instance is right on the target function hyperplane and is close to being incorrectly predicted by the target function. As noisy instances get further away from the correct side of the margin, they have a stronger effect on the mistake bound.

Using our above notation and our definition of N, all deterministic linear-threshold algorithms must have $k \geq 1$. More precisely, for a learning problem with linear-threshold target functions, the number of mistakes is at least $M(\text{Opt}) + \lfloor N/\delta \rfloor$ where $M(\text{Opt})$ is the maximum number of mistakes made by the optimal algorithm for the learning problem with no noisy instances. This is because an adversary can always force a deterministic algorithm to make a mistake with an instance that has $\nu_t = \delta$ noise. The adversary just picks an instance on the target function hyperplane and sets the label to ensure that the algorithm makes the wrong prediction. This forces a mistake for every δ amount of noise. Therefore, for any algorithm, the adversary can create at least $M(\text{Opt})$ mistakes using non-noisy instances and then another $\lfloor N/\delta \rfloor$ mistakes using noisy instances.

Looking over the algorithms, ALMA(p) potentially gives the worst behavior because of the $k = \omega = \max_{i \in T} \|\mathbf{x}_i\|_p$ factor in the noise term. This is a result of the noise definition used by ALMA. The effect is large when p is small, for example, when $p = 2$ the value of ω could be as large as $\sqrt{n+1}$. When $p = \ln n$ the effect is less severe since $\omega \leq e$. As explained in Section 2.4.1, this poor behavior depends on the adversary being able to exploit this weakness.

As we saw in Section 2.2, the Perceptron algorithm has optimal performance as the total noise gets large. In fact, many of the algorithms can give optimal performance on noisy instances. We show in the various Winnow appendices that the noise bounds for the Winnow algorithms approach N/δ as the multiplier approaches 1. This is intuitive, since smaller multipliers cause the algorithm to make a smaller adjustment on a noisy instance. However, there is a trade-off with the Winnow algorithms. As the multiplier gets smaller, the target function term in the mistake-bound grows. Fortunately, the

multiplier that is optimal for learning the target function often gives reasonable performance for noise. However, if the noise is excessive, a smaller multiplier can lower the mistake-bound. In practice, one should try a range of multipliers to get the best performance.

Finally, we do not want to give the impression that a N/δ mistake-bound is ideal. In many ways, optimal is still far from good. If a noisy instance has $\nu_t = 10\delta$ then it will only cause 1 mistake on the target function, but according to the bounds, appears to cost at least 10 mistakes for the learning algorithms. The lower-bound relies on all the noisy instances having a $\nu_t = \delta$. Realistic problems may have $\nu_t > \delta$ for many of the noisy instances. Therefore, even if the noise term is close to N/δ, noisy instances can have a large effect on the number of mistakes.

2.5 Parameters

One difficulty with the algorithms we use in this thesis is that they often have various parameters to set. In fact, as we have seen, the mistake bound of an algorithm often depends on the parameters being set appropriately. Unfortunately, for most practical problems, we do not have enough information about the problem to set the parameter to the theoretically optimal value. In addition, we do not know that the theoretically optimal value, based on the mistake bound proof, is the best value for the problem at hand. The proofs assume an adversary is generating the instances. When something weaker than an adversary, such as a distribution, is generating the instances, a different set of parameters may give better results. Even in the case of an adversary, there are currently no lower bound arguments for learning arbitrary linear-threshold functions with these algorithms that show the above parameter choices are optimal.

One solution to this parameter problem is to combine the results of several versions of the algorithm where each version is using a different parameter value. Because the algorithm's mistake bound varies continuously with the input parameters in the neighborhood of the optimal value,[11] hopefully one choice of parameters will be close

[11] This can be seen in the proofs found in the appendices.

enough to give good results. Ideally, we would like to make a number of mistakes close to the algorithm with the optimal parameters. The naive way to do this is to run several versions with different parameters and predict with the algorithm that is currently making the fewest mistakes. In practice, this will often work quite well, but an adversary can exploit this strategy to greatly increase the number of mistakes.

A better alternative is to use the Weighted Majority Algorithm (WMA) [LW94]. WMA is an on-line algorithm that is designed to perform almost as well as the best expert. Typically, we use the predictions of various algorithms and input them as experts into WMA. By best expert, we mean the expert that makes the fewest mistakes. As long as one expert is based on an algorithm with a "good" set of parameters, WMA will perform well.

WMA is similar to the Normalized Winnow algorithm with $\theta = 1/2$ and the instances restricted to $\{0,1\}^n$.[12] The only difference is WMA updates its hypothesis on every trial, not just trials with mistakes. This change sacrifices the ability of WMA to learn arbitrary linear-threshold functions in exchange for fewer mistakes when learning the best expert.[13] More formally, let n be the number of experts, let $\alpha = 2$, and let q be the minimum number of mistakes by any expert. The number of mistakes made by WMA is at most $2.41q + 2.41 \ln n$ [LW94]. There are other expert algorithms that have similar bounds [CBFH$^+$97].

While WMA is a good solution, particularly when dealing with an adversary, we explore other solutions to the parameter selection problem in Chapter 3 and Chapter 7. These solutions are aimed at problems where instances are generated by distributions. Distributions are more typical of the problems one is likely to encounter in practical machine learning and is the focus of much of the remaining chapters in this thesis.

---

[12] To use the algorithms in this thesis as experts, one must change any -1 prediction of an algorithm to 0 before inputting the prediction into WMA.

[13] In truth, making updates on every trial does not improve the performance against an adversary but can improve performance when dealing with something weaker than an adversary.

2.6 Summary

In this chapter, we give the linear-threshold algorithms used in this thesis and explain the adversarial on-line model that is commonly used to analyze these algorithms. For each algorithm, we give some information that may be useful in determining when to use the algorithm, such as a bound on the number of mistakes and conditions for when that bound is valid. Some of these bounds are improvements to previously published results. More details can be found in the appendices.

To facilitate comparison between the various algorithms, we convert all the mistake bounds into a single consistent notation. In previously publications, each algorithm used slightly different conventions and notation making any comparison difficult. With this uniform notation, we can more easily identify potential advantages between the various algorithms in the adversarial on-line setting.

These upper-bounds on the number of mistakes depend on certain details of the learning problem. For most problems, some of these details are unavailable to the learner. However, the bounds are still of value. First, a mistake-bound guarantees that the algorithm will work for a wide range of problems. An algorithm without a proven mistake bound may fail on untested problems. Second, a mistake bound allows us to easily compare algorithms. This comparison may show a particular algorithm is superior on a set of problems. Third, the bounds quantify the performance of the algorithms in terms of various problem parameters. For example, the Winnow algorithm performs well even with a large number of irrelevant attributes. This property is evident in the mistake bound.

While these linear-threshold algorithms are interesting in their own right, our main purpose, in this dissertation, is to use them as pieces of more complicated algorithms. The rest of the dissertation consists of techniques that modify on-line learning algorithms to make them more suitable for handling specific types of problems. Many of these techniques are designed explicitly for linear-threshold algorithms.

Chapter 3

Improving On-line Learning with Hypotheses Voting

In the previous chapter, we focused on the traditional analysis of on-line learning in which an adversary generates the instances. However, many practical problems do not involve a malevolent adversary trying to maximize the number of mistakes. Some problems are better modeled by sampling instances from a distribution. For example, when predicting phenomena in the natural world, such as sunspots or earthquakes, it is unlikely that nature is conspiring against our predictions.

All the algorithms we covered in Chapter 2 have performance guarantees against an adversary. We call these adversarial algorithms. Because the upper-bound on mistakes for an adversary covers all possible sequences of instances, these mistake bounds also apply when the instance are sampled from a distribution. Therefore adversarial algorithms are often applied to distribution based problems [CS96, LSCP96, DKR97, KMB03, BKV03, KR98, ZDJ01, TCS03, RZ98, RtY01, Sid02, GR99, BB01, MKCN98]. Unfortunately, most adversarial algorithms do not exploit the extra assumptions implicit when instances are generated by a distribution.

In this chapter, we give a technique that modifies an adversarial on-line algorithm and improves its performance on problems where instances are independently picked from an identical distribution. We give experiments that show these techniques lower the number of mistakes on real world data sets along with arguments to justify why this technique works.

Our technique modifies an existing on-line algorithm, B, to generate a new algorithm. We call B the basic algorithm and the new algorithm V-B. A basic on-line algorithm naturally generates new hypotheses as it changes its current hypothesis during an update. V-B periodically saves some of these hypotheses and uses a weighted

vote of the saved hypotheses for prediction. V-B selects these hypotheses in an attempt to maximize the accuracy of prediction.

Our main algorithm of the chapter is V-Combine. It is nearly identical to V-B except that it selects hypotheses from a set of basic algorithms. Any time algorithm V-B selects the current hypothesis from algorithm B, algorithm V-Combine selects the current hypothesis from whichever basic algorithm is currently making the fewest mistakes. This allows algorithm V-Combine to select hypotheses from the best of the basic algorithms.

This voting techniques is designed for inexpensive on-line algorithms. The algorithms we have tested all allow instances $x_t \in [0,1]^n$ and use linear-threshold functions to represent hypotheses. The time cost for these algorithms is $O(m_t)$ per trial where m_t is the number of non-zero attributes during trial t.[1] A major constraint of our technique is to keep the computational advantage of these inexpensive algorithms. This is necessary for problems with a large number of attributes. For example, we give results from text data experiments that have over 30,000 attributes.

The remainder of the chapter is organized as follows. In Section 3.1, we give information on previous research with on-line voting techniques. In Section 3.2, we give the motivation behind our on-line voting techniques. In Section 3.3, we give the voting technique. This includes a six part incremental breakdown of the modifications used in our voting procedure. In Section 3.4, we give experiments with the our voting technique on real world data sets. These experiments compare our technique with the basic on-line algorithms and some previously published on-line voting techniques.

3.1 Previous Research

There is a wide range of previous work on using voting techniques to improve the performance of prediction algorithms. However, only a small amount of this work has considered on-line learning. In this thesis, we only consider previous work that deals with on-line learning.

[1] More details can be found in Appendix D.

Our work builds off the on-line voting ideas in [Lit95]. The basic idea in [Lit95] is to take a random uniform sample of 30 hypotheses from the previous trials. A majority vote of these 30 hypotheses is used for predictions. To keep the computational costs low, the hypotheses used from trial to trial are not independent. During every trial, all the hypotheses currently used by the voting procedure have a small probability of being replaced by the current hypothesis. By choosing this probability appropriately, the hypotheses voting procedure predicts with a uniform sample of hypotheses from previous trials. A large motivation of this voting technique is to reduce the effects of noisy data on mistake-driven on-line learning algorithms [Lit95].

Another technique, presented in [OR01], uses bagging [Bre96] as a motivation to improve on-line voting. The main idea is to create multiple hypotheses from various samples of instances and then combine these hypotheses with a vote. The primary means of sampling is a Poisson distribution with a hypothesis often getting multiple copies of the current instance. Later we show that this technique is not as effective as Littlestone's voting for the algorithms considered in this thesis.

A related technique presented in [FG03] uses the Arc-x4 boosting algorithm [Bre98] to improve performance with voting. The algorithm assigns a value to each instance that is reflective of its importance in being classified correctly [Elk01]. All instances start with an importance of 1. When an instance arrives it is used to update the voting hypotheses sequentially. If the instance is predicted correctly by a voting hypothesis then its importance weight is decreased. If the instance is predicted incorrectly then its importance value is increased. This causes voting hypotheses later in the sequence to focus more of their effort correctly predicting instances that are difficult for the early hypotheses [FS96].

While the Arc-x4 algorithm is interesting, it allows the algorithm to explore a different representation space. We are primarily interested in linear-threshold functions in this thesis. Boosting algorithms, in general, use their voting hypotheses as a way to extend the representational ability of the algorithm. For example, an algorithm that can only learn hyperplanes can be used with boosting to create an algorithm that can learn a linear combination of hyperplanes that can perfectly separate the data [FS96].

In many ways, this amounts to learning new features where each feature is a hyperplane learned by focusing on a subset of the instances. While feature exploration is a desirable research avenue for on-line learning, it is not a problem we address in this dissertation.

In [FS98], voting uses all the hypotheses generated during the previous trials. If a hypothesis was used for l trials then it gets a weight of l when voting. This technique can become expensive as the number of trials grows. Saving and predicting with all the hypotheses multiplies the time and space costs of a mistake-driven algorithm by the number of mistakes. For some problems, this can become computationally impractical. The solution proposed in [FS98] for the Perceptron algorithm is to store only one hypothesis that keeps track of the average weight values of all the hypotheses. Later, we give results of this technique applied to all of our basic algorithms.

3.2 Voting Motivation

The main motivation for our voting technique is based on the observation that many algorithms have unstable accuracy in their hypotheses when instances are generated by a distribution. Instead of a smooth increase in accuracy over the course of the learning trials, these algorithms have an accuracy that can jump over a range of values. While this accuracy, on average, tends to increase, the trials with a low accuracy hypothesis can inflate the number of mistakes.

A good example of this instability can be seen in Figure 3.1. The figure contains a graph that shows the accuracy during a typical run of Balanced Winnow. The learning problem is to predict whether or not an area of land has forest cover that is of type spruce.[2] The accuracy is measured at every trial by sampling with a holdout set of 10,000 test instances. As can be seen, the accuracy is unstable. These inaccurate hypotheses cause extra mistakes.

The goal of an on-line algorithm is to minimize the number of mistakes. When the instances are generated by a distribution, this is achieved by predicting on each trial with a hypothesis that has high accuracy. All the algorithms in this thesis behave in a

[2]For more information on this concept see Section 3.4.1.

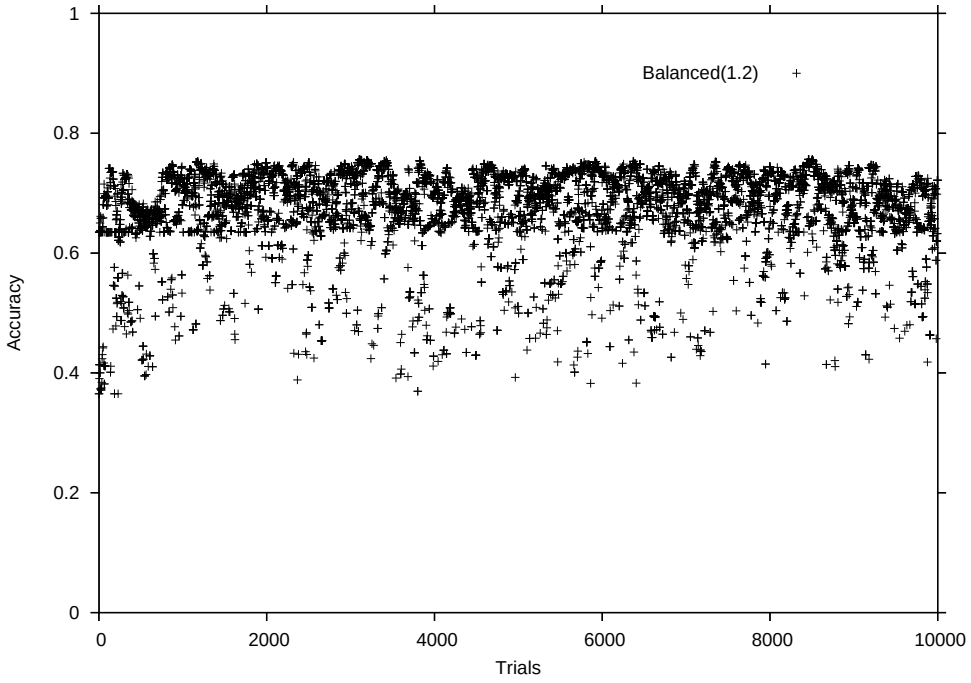

Figure 3.1: Accuracy of Balanced with $\alpha = 1.2$ on forest cover problem.

similar unstable way as Balanced Winnow on the data sets we have tested. This causes these algorithms to make extra mistakes.

The lack of stable hypothesis accuracy can partially be explained by the mistake-driven nature of these algorithms. As the current hypothesis improves, the algorithm is more likely to feel the effect of the noisy instances since the instances that correspond to the correct hypothesis are more likely to be skipped for updates. If the optimal hypothesis is reached, only the noisy instances can update the hypothesis. These noisy instances can cause a large change in the accuracy of the hypothesis.

Voting helps solve this problem in two ways. First, it uses a large number of hypotheses to remove the effects of poor hypotheses. If there are a few bad hypotheses, they do not outvote the majority of good hypotheses. Second voting improves accuracy by combining the influence of somewhat independent hypotheses. This is an effect that is similar to bagging [Bre96]. When a noisy instance occurs, the hypothesis is often perturbed to a poor hypothesis. The algorithm will continue to make updates on good, non-noisy trials in an attempt to correct the hypothesis. It may not get all the way to

the optimal hypothesis before another noisy instance perturbs the hypothesis in a potentially new direction. The basic algorithm may never learn the optimal hypothesis, it may only learn partially correct hypotheses. Since the trials are drawn from a distribution, the hypotheses are randomly perturbed and corrected based on the distribution. For many distributions, this will cause the partially correct hypotheses to perform well on different instances. Based on this intuition, these different hypotheses are likely to be spread out over the various trials.

3.3 Voting Algorithm

In this section, we give the algorithmic details of transforming a basic algorithm into our voting algorithm. We develop the voting technique gradually by starting with a simple form of the algorithm and continually adding details until we get the complete algorithm. In total, we make six refinements to our voting technique. We will give names to each of these refinements so that we can later test the effect of each addition with experiments. Our naming convention is to add a prefix to the name of the basic learning algorithm. For example, let B be a basic learning algorithm. L-B is the name we give to the Littlestone voting technique applied to algorithm B. The main algorithm of this chapter uses all six of our refinements.

To analyze the cost of a voting procedure, we break the time complexity into two pieces: the cost for the prediction step and cost of the update step. We assume that the number of attributes in an instance is always n but that the number of non-zero attributes for trial t is m_t. Furthermore, we assume these instances are presented to the learner in a sparse format that takes $O(m_t)$ storage.[3] We assume that the basic algorithms use $O(n)$ storage for their hypotheses and that they can predict and update these hypotheses in time $O(m_t)$. In many cases, our voting techniques needs to perform predictions during updates. These predictions are counted with the update cost since they occur during the update phase of on-line learning.

We need some additional notation to express the cost of the voting procedure. Let

[3]See Appendix D for more information on this sparse format.

$|M_T|$ be the number of mistakes made by the basic algorithm on a sequence of instances of length T. The particular sequence does not matter as the cost of the voting only depends on the number of mistakes. To simplify our results, let $m = \max_{t \in T} m_t$. The cost of the algorithm depends on the sparsity of the individual instances used, but it is simpler to express the bounds in terms of the maximum sparsity.

3.3.1 Littlestone's Voting

Littlestone's algorithm [Lit95] runs a basic on-line algorithm with no modification, however it does not use the basic algorithm's hypothesis to make predictions. Instead the algorithm stores h hypotheses, where h is a parameter set by the user, and predicts according to the majority prediction of these h hypotheses. At the start of each trial, the voting algorithm potentially replaces each of the the h voting hypotheses with the current hypothesis of the basic algorithm. Each hypothesis is independently replaced with probability $1/t$, where t is the current trial. This gives the voting a uniform distribution of hypotheses from the sequence of hypotheses used over the previous trials.

Over T trials, the total prediction cost of Littlestone's voting procedure is $O(hmT)$ because each stored hypotheses must make a prediction on every trial and each prediction takes $O(m)$. The update cost is more complicated since the voting algorithm is randomized. Each hypothesis is replaced with probability $1/t$ on trial t, therefore each hypothesis is expected to be replaced $\sum_{t=1}^{T} 1/t \leq \ln T + 1$ times. Since the cost of replacing a hypothesis is $O(n)$ and there are h hypotheses, the total expected cost is $O(hn \log T)$. Adding in the cost for updating the basic algorithm gives a cost of $O(m|M_T| + hn \log T)$. This gives a total expected time of $O(hmT + hn \log T)$ for predicting and updating. The space complexity is just the space complexity of the basic algorithm plus $O(hn)$ to store the h voting hypotheses. For all the algorithms we consider, this gives an $O(hn)$ space requirement.

3.3.2 Modification A

Our first modification is fairly simple. One problem with Littlestone's voting procedure is that at the start of voting the voting procedure is full of hypotheses from early trials. Since the algorithm has not had time to learn, these hypotheses are quite poor. Unfortunately, even after the basic algorithm starts to improve, these older, poor hypotheses are used for voting until they are randomly replaced. To solve this problem, instead of always predicting with the majority vote, we predict with the basic algorithm if it has better performance. Our measure of performance is just the current number of mistakes made by the voting hypotheses compared to the basic algorithm. In case of a tie, we randomly pick which prediction to use.

A related minor modification is to always use the most recent hypothesis in the voting procedure. We can do this essentially for free since the basic algorithm uses this hypothesis. Also, majority voting is most effective when there is an odd number of hypotheses.[4] Therefore we always round h up to an even number. Combining this with the most recent hypothesis gives an odd number of hypotheses.

The cost of this modification is essentially zero. The prefix given the voting procedure that uses these modifications is Va. For example, applying these modifications to the Perceptron algorithm forms the algorithm Va-Perceptron.

3.3.3 Modification B

In Littlestone's algorithm, h hypotheses are selected uniformly over all the trials. This has the effect of spreading out the selected hypotheses over the previous trials. This is an inexpensive way to randomly select hypotheses that are likely to make mistakes on different instances. As we explained in Section 3.2, the further apart the hypotheses the more likely that they have been perturbed in different ways.

Our second modification removes the randomization and picks h hypotheses from

[4]Let V_1 be a voting procedure with an even number of hypothesis. Let V_2 be identical to V_1 except that one hypothesis has been removed. Algorithm V_1 can have a tie when predicting a concept. The tie is generally decided by an arbitrary choice. Algorithm V_2 will give the exact same predictions as V_1 on all instances except instances that caused ties. Since V_1 made an arbitrary decision on these instances, the extra hypothesis of V_1 is not useful.

trial	hypotheses	spacing
1	(**1**)	1
2	(1,**2**)	1
3	(1,2,**3**)	1
4	(1,2,3,**4**)	1
5	(1,2,3,4)	1
6	(2,3,4,**6**)	2
7	(2,3,4,6)	2
8	(2,4,6,**8**)	2
9	(2,4,6,8)	2
10	(2,4,6,8)	2

trial	hypotheses	spacing
11	(2,4,6,8)	2
12	(4,6,8,**12**)	4
13	(4,6,8,12)	4
14	(4,6,8,12)	4
15	(4,6,8,12)	4
16	(4,8,12,**16**)	4
⋮	⋮	⋮
24	(8,12,16,**24**)	8
⋮	⋮	⋮
32	(8,16,24,**32**)	8

Figure 3.2: Example of the hypothesis replacement for voting modification B.

regularly spaced trials. For example, if $h = 4$ and the algorithm is at trial 128, we vote with hypotheses from trial 32, 64, 96, and 128. However to always keep a close to equal spacing between hypothesis, we would need to store many of the old hypotheses. For example, at trial 132 to keep a roughly equal spacing we would need the hypotheses from trial 33, 66, 99, and 132. To cut down on the storage requirements, we use a doubling technique to ensure that we reuse many of the saved hypotheses.

The doubling technique works as follows. We only replace voting hypotheses at certain trials in order to continually double the spacing between the voting hypothesis. Let $(t_1, t_2, \ldots)$ represent the trial numbers from which we have selected hypotheses for voting. In Figure 3.2, we give an example of how modification B works when $h = 4$. Every time a hypothesis is added it is marked with the current trial number. At the start, the algorithm just fills the voting with the first 4 hypotheses from the first 4 trials. At this point, the spacing between hypotheses is 1 trial. Next, the algorithm adds new hypotheses to give a spacing of 2. Therefore, the next hypotheses are added at trial 6 and 8. To add these hypotheses, we need to remove some existing hypotheses. The key to the technique is to always remove a hypothesis that creates a gap with the appropriate spacing. In this example, currently we want to create a spacing of 2 trials. We remove the hypothesis that corresponds to trial 1 when adding the hypothesis from trial 6, and we remove the hypothesis from trial 3 when adding the hypothesis from

Hypotheses replacement

Let t be the current trial and let

$\phi = \left\lceil \lg\left(\frac{t+1}{h}\right)\right\rceil \quad k = \left\lfloor \frac{t - h2^{\phi-1}}{2^{\phi}} \right\rfloor$

Add hypothesis

Next hypothesis added from trial $h2^{\phi-1} + (k+1)2^{\phi}$.

Remove hypothesis

Next hypothesis removed from trial $(2k+1)2^{\phi-1}$.

Figure 3.3: Pseudo-code to determine which hypotheses to add and remove for voting modification B.

trial 8. The new hypotheses and the new gaps create a spacing of two trials while only replacing half of the h hypotheses. Next the algorithm creates a spacing of 4 trials. This continues each time doubling the spacing.

Modification B replaces half of the existing hypotheses every time we double the spacing. It is not difficult to implement this system efficiently. When we reach the trial for the new hypothesis, we replace the oldest hypothesis that needs to be removed for the new spacing. The hypothesis to be removed follows a straightforward pattern. The first hypothesis to be placed with spacing 2^s removes the hypothesis with trial number 2^{s-1}. The next hypothesis with spacing 2^s removes the hypothesis with trial number $3(2^{s-1})$. This continues until all $h/2$ hypotheses from spacing 2^s are copied to the voting algorithm. In Figure 3.3, we give pseudo-code to determine the next trial to add a hypothesis to voting and which hypothesis to remove. The prefix given the voting procedure that uses this modification is Vb.

The time complexity of prediction is the same as Littlestone's voting algorithm, $O(mhT)$, because we are still performing each prediction with at most $h+1$ hypotheses. The updating is simplified since we do not have randomization. The cost can be derived from the number of times the spacing doubles. For $T > h$, the trial number is at least $(h/2+1)2^s$ where 2^s is the current spacing. This is based on when the first hypothesis with the new spacing arrives. This inequality can be rearranged to show that $s \leq \lfloor \lg(2T/(h+2)) \rfloor$ where T is the current trial. For the initial hypotheses, we replace h hypotheses; for each following doubling, we replace $h/2$ hypotheses. This

gives at most $h + h\lfloor \lg(2T/(h+2)) \rfloor/2 = O(h \log T)$ hypotheses copied. Including the updating of the basic hypothesis, the total cost is $O(mM + nh \log T)$ for updating. Therefore, the total cost is the same as Littlestone's voting technique.

3.3.4 Modification C

To further refine the voting, our third modification searches for high accuracy hypotheses. Assume we are using modification B and that we have hypotheses from trials $(128, 256, 384, 512)$ where $2^s = 128$ is the size of the current spacing. Instead of using hypotheses from these trials, modification C searches a window of trials centered around theses numbers. The size of the window is $1 + \min(w, f2^s)$ where $f \in [0, 1]$ is the fraction of the full window to search and w is a parameter to control the maximum window size. Letting $w = 100$ and $f = 0.5$, our example gives trial 256 a window from 224 to 288. As soon as we reach trial 224, we save the current hypothesis for voting. Call this hypothesis h_1. For each following trial, up to trial 288, if we get a new hypothesis, we estimate its accuracy. If that accuracy is higher than the estimate for h_1, we replace h_1 with the new hypothesis. This testing and replacement continues until we reach the end of the window.

Each time a new hypothesis is created, we estimate its accuracy by keeping track of the number of instances it predicts correctly during on-line learning. Let a be the number of instances predicted correctly by the hypothesis and let b be the number of total instances predicted. The accuracy estimate is $(a + 1)/(b + 2)$. We refine this estimate by saving r recent instances and testing the accuracy of the hypothesis on these r instances. Let c be the number of instances predicted correctly from the saved instances. Our refined estimate is $(a + c + 1)/(b + r + 2)$. The prefix given to the voting procedure that use this modification is Vbc since we must use modification B in order to use modification C.

The time complexity of the predictions is still $O(hmT)$. The time complexity of the updates is based on the number of hypotheses that we test using the r saved instances, and the number of times we copy a hypothesis because it appears better than our current hypothesis. When dealing with modification B, we derived that there is a new

window at most $h + h \lg (2T/(h+2))/2 = O(h \log T)$ times. For each of these windows, we need to make predictions on at most w hypotheses with r instances. In practice, the number of predictions is much smaller than w since we only need to test hypotheses that have changed. If a hypothesis appears more accurate then we need to copy that hypothesis for voting. At worst this means we have to copy a hypothesis for all w trials of the voting window. Again, this will typically be smaller since we only make a copy when a new hypothesis performs better on our statistical test. Therefore the worst-case cost of the updates is $O(mM + (rwm + wn)h \log T)$. We also need extra space to store the extra r instances. Storing them in a sparse format gives a total space complexity of $O(hn + rm)$.

3.3.5 Modification D

With modification C, we have already spent the effort to estimate the accuracy of the hypotheses we are using for voting. A further modification is to use those accuracy estimates to compute weights for voting. We use a weighting scheme related to the Naive Bayes algorithm [DH73]. While it is possible to use the full Naive Bayes algorithm to compute weights for voting, the lack of real independence between the hypotheses can cause it to make more mistakes than the following simplified version.

The voting procedure gives hypothesis h_i a weight of $\log(p_i/(1-p_i))$ where p_i is the estimate of the accuracy of hypothesis i used in voting. The voting algorithm predicts the weighted majority prediction of all the hypotheses. These weights have the effect of giving hypotheses with a lower accuracy a lower weight. This is helpful to reduce the influence of poor hypotheses that are learned during the beginning trials. These poor hypotheses can stay around a long time and cause extra mistakes. For example, if $h = 30$, the hypothesis from trial 16 will still be used in the voting procedure during trial 480. However, if the hypothesis has a relatively poor accuracy estimate then it will have a smaller influence on the voting when using modification D. Even hypotheses that are learned later in the learning process may benefit from these weights. The prefix given a voting procedure that uses this modification is Vd.

The additional cost of this modification is essentially zero if one has already performed modification B and C. To perform this modification independently, the cost is just based on modification C with a window size of $w = 1$. This will estimate the accuracy of the new hypothesis which can then be used to compute the weight.

3.3.6 Modification E

If the voting procedure is doing poorly, it may be caused by a large number of early, poor hypotheses. While modifications A and D help performance by reducing the influence of poor voting hypotheses, we can also improve performance by removing these inaccurate hypotheses and creating space for accurate hypotheses. Modification E accomplishes this by restarting the voting procedure. After a restart, all the internal variables of the voting procedure are reset to their initial values and the voting procedure restarts with the current trial set as the initial trial. This effectively removes all the hypotheses currently stored in the voting algorithm.

We only check for a restart after a certain number of trials in order to give the voting procedure a chance to learn. After this designated number of trials, we continually check to see if the basic algorithm has made fewer mistakes than the voting algorithm.[5] If the basic algorithm has made fewer mistakes then the voting algorithm is restarted. Let d be the number of trials we wait until we start checking for a restart. The initial value of d in our experiments is 50. After each restart, we double the value of d. Therefore, after k restarts, the algorithm will wait $50(2^k)$ trials until it begins checking for a restart. This allows the voting algorithm to handle cases where a large number of trials is needed before the voting procedure begins to pay off. It also keeps the extra cost associated with restarts small. The prefix given to this voting modification is Ve.

This is the last voting modification that involves a prefix. Therefore our full technique applied to basic algorithm B is Vabcde-B. To simplify our notation, when using all five previous voting modifications, we just use the name V-B. In other words, V-B

[5]Voting modification A uses the basic algorithm for prediction if it has fewer mistakes than the voting procedure. Therefore, to accurately gauge the performance of voting, we need to explicitly keep track of the number of mistakes made by the voting hypotheses.

is the same as Vabcde-B.

The prediction time complexity for this modification is still $O(hmT)$. The update cost is based on the number of restarts. Let q be the number of times the voting is restarted. Based on the doubling scheme, $T \geq d \sum_{i=0}^{q-1} 2^i$. Rearranging this inequality, we get $q \leq \lg(T/d)$. The total update cost will depend on the cost of an update from the voting algorithm we are modifying. Let c be the voting update cost; the new update cost is $O(c \log(T/d))$. Assume B is a basic algorithm. The total update cost of Vabcde-B is $O(mM + (rwm + wn)h(\log T)^2)$. As T grows this still compares favorably to the prediction cost. The space complexity of modification E is the same as the voting algorithm it is modifying.

3.3.7 Modification F

Our last refinement is a way to combine several different learning algorithms into a single voting technique. This is helpful if no single basic algorithm is the best on all problems. We want a voting algorithm that, in a sense, finds the best basic algorithm for voting on a particular problem. This is also the case when considering parameter choices for algorithms. A learning algorithm with parameters can be used to form a collection of learning methods where each algorithm in the collection uses different values for the parameters.

Modification F works on a set of basic algorithms. Let v be the number of basic algorithms. For each basic algorithm we keep track of the total number of mistakes. At the start of a trial, we determine which basic algorithm is making the fewest mistakes. For the rest of that trial, we use that algorithm as the basic algorithm for voting. Recall that the voting techniques all select a hypothesis for inclusion into voting based on the current hypothesis of a basic algorithm. Modification F makes the voting select a hypothesis from the basic algorithm that currently has the fewest mistakes. Notice that this often causes the voting to store hypotheses that were generated by different basic algorithms.

Modification F runs v basic algorithms but only runs one voting algorithm that potentially selects hypotheses from all the basic algorithms. We call this algorithm

Combine. The set of basic algorithms used in Combine are determined from context. When a voting technique is used with Combine we add the prefix notation. Vabcde-Combine is the main algorithm of this chapter since it uses all the voting modifications. Again, for brevity, we also call this algorithm V-Combine.

When using Combine, the time complexity of voting increases because we have to run v basic algorithms. The cost of finding which basic algorithm has the fewest mistakes is $O(vT)$. The time complexity of prediction increases to $O(vmT)$ since each basic algorithm needs to make a prediction on every trial. The time complexity of the updates is increased by $O(vmM')$ where M' is the maximum number of mistakes made by one of the v basic algorithms. Therefore the total time complexity of modification F is the time complexity of the original voting algorithm plus $O(vmT)$. For example, Vabcde-Combine has a total time complexity of $O(vmT+hmT+(rwm+wn)h(\log T)^2)$. The space complexity of modification F is the same as the modified voting algorithm.

While this cost may seem excessive, most of these parameters are fixed by the user, and can be used to control the cost. Furthermore, as the number of trials increases, the logarithmic nature of the update bound means the prediction cost dominates. When $v = O(h)$, the time complexities of our full algorithm and Littlestone's voting algorithm are similar.

3.4 Voting Experiments

In this section, we give experiments to verify the effectiveness of the voting technique. We perform the experiments using each of the linear-threshold algorithms of Chapter 2. As we will see, voting greatly improves performance.

We use the algorithms and parameter setting in Table 3.1 for our experiments. The Winnow-based algorithms use multiplier values 1.05, 1.2, 1.4, 1.7, and 2.0. For Normalized Winnow, we select the same multipliers along with thresholds of 0.3, 0.5, and 0.7. We choose these values based on preliminary experiments with artificial data. For both Normalized and Unnormalized Winnow, for each attribute x_i, we add a new attribute $1 - x_i$. This doubles the number of attributes but allows these algorithms

to learn negative weights. See Section 2.3.1 for more information on complemented attributes.

It is possible that all these Winnow algorithms will perform poorly because of our parameter choices. Fortunately the Perceptron, ALMA(2), and ALMA($\ln n$) algorithms have a finite mistake bound for learning linear-threshold functions regardless of parameter settings.[6] Furthermore, the two selections for the ALMA algorithm show its range of behavior from Perceptron like performance to Winnow like performance. In total, this gives us thirty-three algorithms to use with voting. Of course more algorithm and parameter choices are possible, but we limited the selection to reduce the computational complexity of the V-Combine algorithm.

3.4.1 Data Sets

Most of our data sets come from the UCI data set repository [DNMml]. We selected all problems from this data set that deal with supervised inductive learning and that have at least one thousand instances. This gives us a total of twenty-three UCI data sets. The names of these data sets along with a brief description is given in Table 3.2. For each data set we convert the problem into separate binary concepts. For any problem that has $L > 2$ labels, this creates L binary concepts. For each particular label/problem, all instances with that label are assigned a new label with value 1 while all other instances get a label of -1.

We performed a small amount of processing on the UCI data sets. The data sets have three types of attributes: binary, nominal, and continuous. The binary attributes are left unchanged. A nominal attribute with k values is converted into k binary attributes. Only one of these k attributes is set to 1 corresponding to the value of the original nominal attribute. The remaining binary attributes are set to 0.

Every continuous attribute is converted into eleven attributes. First, the continuous attribute is normalized into the $[0, 1]$ interval. This is our first attribute; call it c. Second, following the technique of [MHBD01], we create nine binary attributes that

[6] See Section 2.3.5.

Full Name	Abbreviated Name
Perceptron	Per
ALMA(2)	ALMA(2)
ALMA($\ln n$)	ALMA($\ln n$)
Balanced Winnow(1.05)	Bal(1.05)
Balanced Winnow(1.2)	Bal(1.2)
Balanced Winnow(1.4)	Bal(1.4)
Balanced Winnow(1.7)	Bal(1.7)
Balanced Winnow(2.0)	Bal(2.0)
Unnormalized Winnow(1.05)	UWin(1.05)
Unnormalized Winnow(1.2)	UWin(1.2)
Unnormalized Winnow(1.4)	UWin(1.4)
Unnormalized Winnow(1.7)	UWin(1.7)
Unnormalized Winnow(2.0)	UWin(2.0)
Complemented Unnormalized Winnow(1.05)	CUWin(1.05)
Complemented Unnormalized Winnow(1.2)	CUWin(1.2)
Complemented Unnormalized Winnow(1.4)	CUWin(1.4)
Complemented Unnormalized Winnow(1.7)	CUWin(1.7)
Complemented Unnormalized Winnow(2.0)	CUWin(2.0)
Normalized Winnow(1.05,.3)	NWin(1.05,.3)
Normalized Winnow(1.2,.3)	NWin(1.2,.3)
Normalized Winnow(1.4,.3)	NWin(1.4,.3)
Normalized Winnow(1.7,.3)	NWin(1.7,.3)
Normalized Winnow(2.0,.3)	NWin(2.0,.3)
Normalized Winnow(1.05,.5)	NWin(1.05,.5)
Normalized Winnow(1.2,.5)	NWin(1.2,.5)
Normalized Winnow(1.4,.5)	NWin(1.4,.5)
Normalized Winnow(1.7,.5)	NWin(1.7,.5)
Normalized Winnow(2.0,.5)	NWin(2.0,.5)
Normalized Winnow(1.05,.7)	NWin(1.05,.7)
Normalized Winnow(1.2,.7)	NWin(1.2,.7)
Normalized Winnow(1.4,.7)	NWin(1.4,.7)
Normalized Winnow(1.7,.7)	NWin(1.7,.7)
Normalized Winnow(2.0,.7)	NWin(2.0,.7)

Table 3.1: A list of the basic algorithms used in our experiments.

Name	concepts	attributes	instances	Description
ad	1	1589	3279	advertisements on Internet
adult	1	161	48842	salary census data
agaricus	1	112	8124	edible mushrooms
cmc	3	57	1473	contraceptive use
connect-4	3	126	67557	game openings
covtype	7	144	581012	forest cover type
flare	1	41	1066	solar flare activity
german	1	124	1000	bank loan
isolet	26	617	7797	speech recognition
kr-vs-kp	1	38	3196	chess end-game
letter	26	160	20000	optical character recognition
nursery	5	26	12960	nursery school applications
optdigits	10	640	5620	optical digit recognition
page-blocks	5	100	5473	document page layout
pendigits	10	160	10992	optical digit recognition
sat	6	360	6435	soil type from satellite image
segmentation	7	190	2310	image segmentation
shuttle	7	90	58000	shuttle data
spambase	1	570	4601	email spam
splice	3	287	3190	DNA splice junctions
thyroid	3	104	9172	thyroid disease diagnosis
mfeat	10	6490	2000	optical digit recognition
yeast	10	80	1484	protein localization sites
news	20	32889	18828	Usenet articles
reuters	11	18307	19813	news wire stories
web	7	22123	8282	web pages

Table 3.2: A description of the data sets used in our experiments.

correspond to the numbers 0 to 9. We set attribute j to 1 if $j/10 < c$. The remaining binary attributes are set to 0. For example, if $c = .25$ then the eleven attributes are $< .25, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0 >$.

This representation of continuous attributes allows a linear-threshold algorithm to represent a wider range of concepts. For example, assume x is a single continuous attribute and that x_0 through x_{10} correspond to the new eleven attributes based on x. Furthermore, assume attribute x only increases the chance of label 1 when $0.2 \geq x \leq 0.8$. Therefore the algorithm can give weight w to attribute x_3 and weight $-w$ to attribute x_9, where w corresponds to the influence of this attribute range on the label. This representation is not possible if the instance only uses attribute x.

We performed this processing on continuous attributes for all the UCI data sets except for isolet. Isolet is too expensive for our experiments since it is composed of a large number of trials, concepts, and continuous attributes. Instead, we just use one attribute with value c for every continuous attribute in isolet.

We also use three data sets that come from popular text sources. Reuters-21578 (reuters) is a data set of Reuters news wire stories [HBdu]. The stories have been categorized under various topics. We use the eleven labels that correspond to the most frequent topics. The twenty Newsgroups data set (news) is a collection of Usenet articles posted to twenty different newsgroups [HBdu]. The label of an article is based on the newsgroups in which it appears. The 4 Universities data set (web) is a collection of web pages that have been manually grouped into seven classes [CDF$^+$ta]. For each of these problems we create a binary concept to learn each label in the same way as the UCI data sets.

The attributes describing a text document are based on the words in the document. If the corresponding word is in the document the feature is set to 1 otherwise the feature is set to 0 [Lew92]. This gives a sparse representation that can be efficiently handled by the algorithms in this thesis. Before generating the features we remove a common set of stop words [Fox90] and stem the words [McCow]. Also to further reduce the number of features, we remove any word that appears less than three times in a data set.

To help design our techniques we used mostly artificial data sets. This is important

since we did not want to bias the results by using the testing data sets to help construct the algorithms. The only exception is the Reuters data set. We made some small improvements to the voting technique after noticing problems with the Reuters data set. This was before we did any experiments with the UCI data sets.

3.4.2 Statistics

For all the experiments in this chapter, we report results based on an average of 50 bootstrap samples [HMM+03]. Let T be the number of instances for a particular experiment. Each bootstrap sample is composed of $\min(T, 10000)$ instances sampled independently with replacement from all the instances in a problem. We limit the number of trials to 10,000 to reduce the cost of running the experiments. This does not have a large effect on our results since most algorithms seem to stop learning before 10,000 trials on these experiments. For statistical significance, we give a confidence interval for a bootstrap sample based on a t-test with a 95% confidence interval [DeG86].

Ideally, our result would include a graph that plots the total number of mistakes at the various trials. In Figure 3.4.2, we give one such graph for the satellite data set with label 1. We plot V-Combine along with the algorithms that make the fewest mistakes from the basic algorithms and Littlestone's voting procedure applied to the basic algorithms. The plots are fairly smooth since it is an average over the 50 bootstrap samples. Notice how the algorithms have a greater error rate at the beginning of the trials and then settle towards a fixed slope line. This is typical for our average plots when dealing with distributions generating the data. At the beginning there is a learning phase. This is followed by a phase where the algorithm does not improve its average error rate. This transition is gradual, and when learning is difficult, such as when there are many attributes, it can require more trials than available to get to the fixed error phase.

Unfortunately, we have too many algorithms and concepts to give mistake plots for every problem. Therefore, our primary technique for comparing algorithms is to give the total number of mistakes summed over all the data sets. However, as explained above, this could give a deceptive picture of the long term error rate of an algorithm.

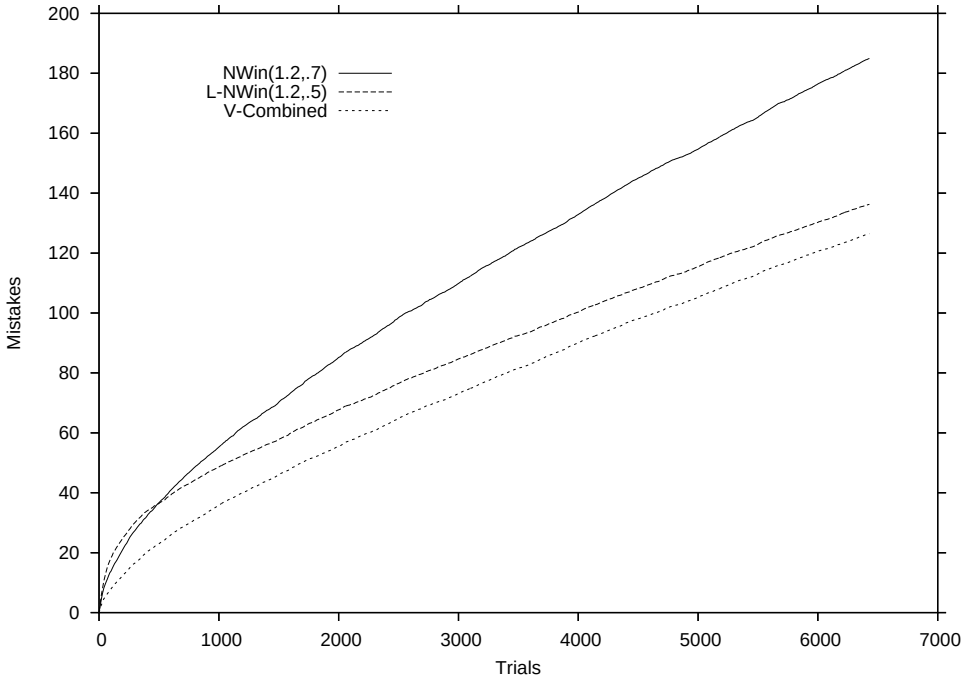

Figure 3.4: Mistake curve for Satellite label 1 concept.

Therefore, we also give the total sum of mistakes on the last 500 trials of each concept. In both cases, we use the notation $\hat{\mathrm{M}}(B)$ for the confidence interval on the expected number of mistakes made by algorithm B.

3.4.3 Main Voting Results

In this section, we report experimental results that show how voting improves the performance of the basic algorithms. All of our experiments set h, the number of voting hypotheses, to 30. This matches the value used in [Lit95]. For our voting algorithm, we use the following default values for the parameters explained in Section 3.3. We set $w = 100$, $f = 0.5$, and $r = 100$. These values are based on limited experiments with artificial data. In Section 5.2.2, we perform experiments where we test different values for these parameters.

Our first results compare a basic algorithms B to our voting algorithm V-B. In Figure 3.5, we give a scatter plot that contains a point for each algorithm/concept pair. This gives a total of 6138 points. The y coordinate of each points corresponds to the

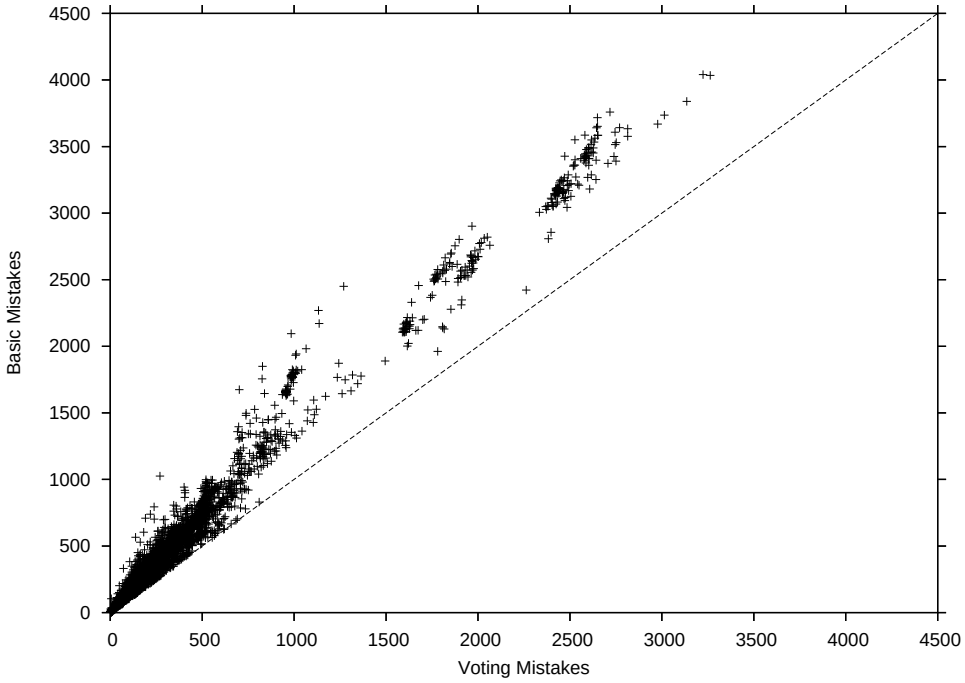

Figure 3.5: Scatter plot comparing basic algorithm to our voting algorithm.

final number of mistakes made by a basic algorithm on the particular concept; the x coordinate corresponds to the final number of mistakes made by the full voting version of the basic algorithm. Because all the points are above the $y = x$ line, the voting version increases performance for every algorithm/concept pair.

A large part of voting's advantage comes from the technique used by Littlestone's voting algorithm. In Figure 3.6, we repeat the previous scatter plot, this time using the final number of mistakes made by Littlestone's voting for the y coordinate and the final number of mistakes made by our voting procedure for the x coordinate. As can be seen, while the gain performance is not as dramatic, there is a consistent decrease in the number of mistakes. Only 149 of the algorithm/concept pairs show a small increase in mistakes when using our voting technique. By far, the majority of algorithms and concepts show a large statistically significant decrease in mistakes.

To further document the performance of voting, our remaining results focus on the performance of the individual algorithms and the V-Combine algorithm. In Table 3.3 we give the total number of mistakes for each basic algorithm, Littlestone's voting

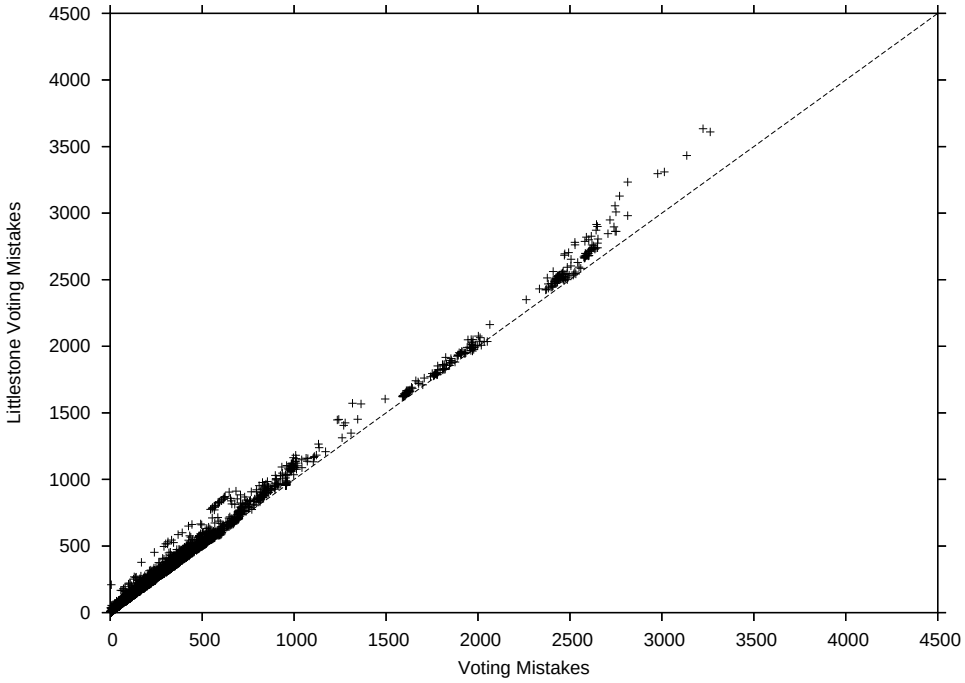

Figure 3.6: Scatter plot comparing Littlestone's voting algorithm to our voting algorithm.

algorithm, and our voting algorithm. For a given algorithm B, we refer to Littlestone's modified algorithm as L-B and our modified algorithm as V-B. We also included a new algorithm to help judge the difficulty of these concepts. The Majority Label algorithm keeps a running estimate of the probability that the label is 1. The algorithm predicts the label that has the highest probability based on this estimate.

In Table 3.4, we give the total number of mistakes of the algorithms on the last 500 trials of the concepts. This gives an approximation of the asymptotic behavior of these algorithms.

As can be seen in Table 3.3, Littlestone's voting algorithm always lowers the number of mistakes in these experiments. In some cases, the decrease is dramatic. In particular, the fewest mistakes, of the Littlestone voting algorithms, is made by L-Balanced(1.2) with a decrease of more than 27% over Balanced(1.2). Our voting algorithms continue this trend and always lowers the number of mistakes compared to Littlestone's algorithm. While the decrease is often only slightly better, the similar computational costs

Name	$\hat{M}$(Name)	$\hat{M}$(L-Name)	$\hat{M}$(V-Name)
Per	73342 ± 80	54352 ± 108	**51295** ± 74
ALMA(2)	66066 ± 95	52838 ± 117	**50046** ± 93
ALMA($\ln n$)	71023 ± 105	55217 ± 90	**53301** ± 82
Bal(1.05)	72989 ± 83	53336 ± 107	**51275** ± 96
Bal(1.2)	72398 ± 83	52542 ± 111	**50621** ± 84
Bal(1.4)	74887 ± 86	52904 ± 99	**50779** ± 79
Bal(1.7)	81075 ± 109	55954 ± 118	**52386** ± 87
Bal(2.0)	86641 ± 89	57199 ± 113	**54221** ± 79
UWin(1.05)	93375 ± 93	70226 ± 179	**65027** ± 103
UWin(1.2)	88163 ± 93	64315 ± 152	**59561** ± 87
UWin(1.4)	88900 ± 87	62678 ± 131	**57949** ± 74
UWin(1.7)	94672 ± 103	65376 ± 144	**58841** ± 73
UWin(2.0)	100624 ± 89	63452 ± 104	**60564** ± 81
CUWin(1.05)	99476 ± 98	77527 ± 154	**68613** ± 94
CUWin(1.2)	85792 ± 91	63654 ± 86	**59509** ± 84
CUWin(1.4)	81676 ± 96	59746 ± 114	**56068** ± 92
CUWin(1.7)	81567 ± 95	57845 ± 114	**55141** ± 73
CUWin(2.0)	82557 ± 78	57579 ± 94	**55290** ± 77
NWin(1.05,.3)	110746 ± 76	105828 ± 158	**87083** ± 110
NWin(1.2,.3)	88760 ± 89	73787 ± 120	**66453** ± 87
NWin(1.4,.3)	85753 ± 96	65946 ± 110	**60923** ± 78
NWin(1.7,.3)	88529 ± 85	63792 ± 95	**59332** ± 76
NWin(2.0,.3)	92840 ± 97	62376 ± 119	**59661** ± 92
NWin(1.05,.5)	86721 ± 91	61630 ± 136	**58946** ± 145
NWin(1.2,.5)	80307 ± 102	58627 ± 124	**55167** ± 94
NWin(1.4,.5)	79405 ± 91	56055 ± 102	**54021** ± 80
NWin(1.7,.5)	81935 ± 91	55709 ± 86	**54098** ± 71
NWin(2.0,.5)	85610 ± 111	56979 ± 99	**54927** ± 74
NWin(1.05,.7)	86297 ± 97	71848 ± 143	**65306** ± 96
NWin(1.2,.7)	77574 ± 82	59102 ± 99	**55917** ± 87
NWin(1.4,.7)	76053 ± 93	56666 ± 109	**53446** ± 77
NWin(1.7,.7)	77541 ± 90	55338 ± 93	**52705** ± 71
NWin(2.0,.7)	80087 ± 92	56065 ± 107	**53127** ± 72
Combine			**46432** ± 86
Majority Label	124266 ± 50		

Table 3.3: Total mistakes out of 1028600 trials from 186 concepts.

Name	$\hat{\mathrm{M}}$(Name)	$\hat{\mathrm{M}}$(L-Name)	$\hat{\mathrm{M}}$(V-Name)
Per	4734 ± 22	3443 ± 19	$\mathbf{3303} \pm 15$
ALMA(2)	4304 ± 18	3451 ± 19	$\mathbf{3259} \pm 20$
ALMA($\ln n$)	4590 ± 25	3547 ± 20	$\mathbf{3468} \pm 18$
Bal(1.05)	4708 ± 22	3445 ± 16	$\mathbf{3309} \pm 18$
Bal(1.2)	4700 ± 21	3423 ± 17	$\mathbf{3285} \pm 18$
Bal(1.4)	4979 ± 23	3433 ± 18	$\mathbf{3323} \pm 17$
Bal(1.7)	5522 ± 24	3685 ± 20	$\mathbf{3487} \pm 18$
Bal(2.0)	5960 ± 30	3838 ± 18	$\mathbf{3649} \pm 15$
UWin(1.05)	5739 ± 23	4319 ± 20	$\mathbf{4054} \pm 17$
UWin(1.2)	5494 ± 20	3983 ± 19	$\mathbf{3716} \pm 17$
UWin(1.4)	5745 ± 22	3938 ± 19	$\mathbf{3699} \pm 15$
UWin(1.7)	6317 ± 25	4256 ± 19	$\mathbf{3840} \pm 14$
UWin(2.0)	6830 ± 25	4208 ± 17	$\mathbf{4004} \pm 17$
CUWin(1.05)	5758 ± 23	4454 ± 21	$\mathbf{4173} \pm 16$
CUWin(1.2)	5215 ± 25	3863 ± 16	$\mathbf{3696} \pm 16$
CUWin(1.4)	5181 ± 23	3726 ± 18	$\mathbf{3567} \pm 17$
CUWin(1.7)	5359 ± 21	3736 ± 19	$\mathbf{3592} \pm 17$
CUWin(2.0)	5516 ± 21	3780 ± 17	$\mathbf{3645} \pm 17$
NWin(1.05,.3)	6022 ± 29	5368 ± 25	$\mathbf{5162} \pm 27$
NWin(1.2,.3)	5301 ± 21	4296 ± 20	$\mathbf{3975} \pm 18$
NWin(1.4,.3)	5392 ± 25	3970 ± 20	$\mathbf{3754} \pm 16$
NWin(1.7,.3)	5787 ± 21	3967 ± 18	$\mathbf{3764} \pm 16$
NWin(2.0,.3)	6224 ± 20	3998 ± 19	$\mathbf{3858} \pm 18$
NWin(1.05,.5)	5300 ± 20	3845 ± 21	$\mathbf{3689} \pm 17$
NWin(1.2,.5)	5077 ± 19	3718 ± 19	$\mathbf{3523} \pm 17$
NWin(1.4,.5)	5131 ± 22	3640 ± 16	$\mathbf{3499} \pm 19$
NWin(1.7,.5)	5460 ± 24	3682 ± 19	$\mathbf{3565} \pm 18$
NWin(2.0,.5)	5804 ± 24	3794 ± 20	$\mathbf{3658} \pm 16$
NWin(1.05,.7)	5327 ± 22	4515 ± 20	$\mathbf{4103} \pm 16$
NWin(1.2,.7)	4955 ± 21	3718 ± 19	$\mathbf{3563} \pm 19$
NWin(1.4,.7)	4966 ± 21	3658 ± 17	$\mathbf{3460} \pm 17$
NWin(1.7,.7)	5212 ± 23	3641 ± 16	$\mathbf{3486} \pm 16$
NWin(2.0,.7)	5451 ± 23	3752 ± 19	$\mathbf{3556} \pm 18$
Combine			$\mathbf{3103} \pm 18$
Majority Label	9404 ± 14		

Table 3.4: Sum of mistakes from the last 500 trials of 186 concepts.

of the two algorithms makes it worthwhile.

The results in Table 3.4 closely parallel the previous results. This is partially due to the fact that many of the concepts rapidly converge, on average, to a fixed error rate. Therefore this error rate has a large influence on the total number of mistakes.

The best algorithm in both tables is V-Combine. V-Combine uses modification F to combine the basic algorithms. It makes over 11% fewer mistakes than L-Balanced(1.2). A large part of its advantage comes from its ability to use different algorithms when voting. This can be seen in Table 3.5. This table gives the number of concepts where the corresponding algorithm has the fewest mistakes. The first column deals with the basic algorithms; the second column gives the results for our voting algorithm. For all concepts, a voting algorithm always produces the minimum number of mistakes.

One can make several interesting observations from Table 3.5. First, no single algorithm dominates the performance of the basic algorithms. This shows that running a wide range of algorithms is beneficial for V-Combine as it allows it to use hypotheses from whichever basic algorithm has the fewest mistakes. Second, the basic algorithm with the fewest mistakes is not always the voting algorithm with the fewest mistakes. This shows that our strategy of just picking the basic algorithm with the fewest mistakes for voting in V-Combine does not appear optimal since another basic algorithm may make fewer mistakes when used for voting.

Given these facts, it is surprising that V-Combine does so well. V-Combine might get an advantage because it uses several basic learning algorithms. At a particular trial, V-Combine selects a hypothesis from the algorithm that is currently doing the best. This may not be the basic algorithm that makes the fewest mistakes at the final trial. These hypotheses from different basic algorithms are most likely more diversified than hypotheses from the same algorithm. This extra diversity improves the performance of voting.

Notice that several of the basic algorithms in Table 3.5 do not have the minimum number of mistakes for any concept. However, based on the previous paragraph, it is possible that they still might help performance by increasing the diversity of the hypotheses used by V-Combine. Also, notice how the Normalized Winnow algorithm

Name	# basic best concepts	# V-Name best concepts
Per	0	0
ALMA(2)	56	15
ALMA($\ln n$)	17	2
Bal(1.05)	0	0
Bal(1.2)	1	1
Bal(1.4)	2	3
Bal(1.7)	0	2
Bal(2.0)	0	2
UWin(1.05)	2	0
UWin(1.2)	0	0
UWin(1.4)	0	2
UWin(1.7)	0	1
UWin(2.0)	0	0
CUWin(1.05)	0	0
CUWin(1.2)	0	0
CUWin(1.4)	0	0
CUWin(1.7)	0	0
CUWin(2.0)	0	3
NWin(1.05,.3)	2	0
NWin(1.2,.3)	0	0
NWin(1.4,.3)	0	1
NWin(1.7,.3)	1	1
NWin(2.0,.3)	0	1
NWin(1.05,.5)	0	3
NWin(1.2,.5)	0	3
NWin(1.4,.5)	1	2
NWin(1.7,.5)	0	0
NWin(2.0,.5)	1	4
NWin(1.05,.7)	39	2
NWin(1.2,.7)	21	4
NWin(1.4,.7)	6	12
NWin(1.7,.7)	8	13
NWin(2.0,.7)	29	12
Combine		97

Table 3.5: Number of concepts where the algorithm gives the minimum number of mistakes.

with threshold value of 0.7 seems to dominate in terms of performance. With the exception of the ALMA algorithm, it has the fewest mistakes on almost all the other concepts. This suggests that the concepts we use, on average, are somewhat conjunctive in nature where most of the relevant attributes need to have a value of 1 in order for the label to be 1. See Section 2.3.3 for more details.

To further elaborate on the statistics of Table 3.3, in Table 3.6 we give the number of concepts where Littlestone's voting algorithm makes fewer mistakes than our full voting algorithm. In addition, we give the sum of the mistake difference over those concepts where our voting technique make more mistakes. As can be seen, for most algorithms we improve on the performance of Littlestone's algorithms, and in those few cases where we make more mistakes, the number of extra mistakes is small.[7]

3.4.4 Voting Modifications

In this section, we break down our voting technique based on the individual modification described in Section 3.3. However, it requires too many experiments to test every legal combination of the voting modifications. Therefore, to give a partial analysis of the voting modifications, Table 3.7 gives results based on incrementally adding modification A through E to Littlestone's voting algorithm. To make the results easier to interpret, each column gives the change in mistakes based on adding the next modification. In other words, given on-line algorithm B, we report

$$\begin{aligned}
\Delta(Va) &= \hat{\mathrm{M}}(\text{L-}B) - \hat{\mathrm{M}}(\text{Va-}B). \\
\Delta(Vab) &= \hat{\mathrm{M}}(\text{Va-}B) - \hat{\mathrm{M}}(\text{Vab-}B). \\
\Delta(Vabc) &= \hat{\mathrm{M}}(\text{Vab-}B) - \hat{\mathrm{M}}(\text{Vabc-}B). \\
\Delta(Vabcd) &= \hat{\mathrm{M}}(\text{Vabc-}B) - \hat{\mathrm{M}}(\text{Vabcd-}B). \\
\Delta(Vabcde) &= \hat{\mathrm{M}}(\text{Vabcd-}B) - \hat{\mathrm{M}}(\text{Vabcde-}B).
\end{aligned}$$

Therefore if the additional voting modification lowers the mistakes from the previous modifications then the corresponding entry in the table is positive. Table 3.8 includes

[7] In Table 3.6, the number of extra mistakes is not an integer because our results are based on a bootstrap average.

Name	# of concepts	$\hat{M}$(V-Name)-$\hat{M}$(L-Name)
Per	4	1.5
ALMA(2)	2	0.2
ALMA($\ln n$)	4	0.2
Bal(1.05)	5	9.8
Bal(1.2)	6	2.0
Bal(1.4)	13	3.9
Bal(1.7)	3	0.2
Bal(2.0)	1	0.0
UWin(1.05)	10	3.7
UWin(1.2)	7	3.6
UWin(1.4)	3	1.5
UWin(1.7)	3	0.1
UWin(2.0)	4	16.0
CUWin(1.05)	0	0
CUWin(1.2)	1	3.9
CUWin(1.4)	0	0
CUWin(1.7)	1	3.5
CUWin(2.0)	6	10.9
NWin(1.05,.3)	0	0
NWin(1.2,.3)	0	0
NWin(1.4,.3)	1	17.6
NWin(1.7,.3)	1	9.0
NWin(2.0,.3)	11	29.1
NWin(1.05,.5)	10	7.2
NWin(1.2,.5)	4	8.7
NWin(1.4,.5)	4	17.9
NWin(1.7,.5)	14	11.9
NWin(2.0,.5)	13	28.1
NWin(1.05,.7)	6	8.6
NWin(1.2,.7)	8	16.1
NWin(1.4,.7)	1	3.8
NWin(1.7,.7)	1	10.2
NWin(2.0,.7)	2	0.2

Table 3.6: Number of concepts where Littlestone's voting algorithm makes fewer mistakes than our voting algorithm and the sum of the difference over those concepts.

the same algorithms but reports the difference in mistakes from the last 500 trials.

As can be seen in Table 3.7, in almost all cases the modifications yield fewer mistakes. The sole exceptions are for modification B. Note however that modification C only works when using modification B, modification C always significantly lowers the number of mistakes, and the few cases where modification B increases the number of mistakes, modification C more than makes up the deficit. The modification with the least average effect is modification E, restarting. While restarting is not, on average, very successful it does consistently lower the number of mistakes. Also it gives a large decrease in mistakes for V-Combine, which is the main algorithm of this chapter.

Table 3.8 shows a similar consistent gain in performance on the last 500 trials of the concepts. There are some discrepancies in the amount of improvement. Some of the techniques that have a large improvement over all the trials have a relatively smaller improvement during the final 500 trials. This can be explained by the initial learning phase of the algorithms as described in Section 3.4.2. Some of the modifications may help an algorithm more during the initial phase and some may the help the algorithm more with the final error-rate. More detailed analysis is obscured by the limited statistical significance of the the mistakes from the final 500 trials.

3.4.5 On-line Bagging

Bagging [Bre96] can also be used to create an on-line voting technique. On-line Bagging predicts with a majority vote of h hypotheses. The technique runs h versions of a basic algorithm where each algorithm is modified by updating with the current instance a random number of times. The number of updates is based on a Poisson distribution with a mean of 1. This simulates a bagged sample and hopefully causes the hypotheses to become independent enough to reduce the number of mistakes when predicting with a majority vote [OR01]. Given a basic algorithm B let B-B be the on-line bagged version of the algorithm. In Table 3.9, we give the results of applying this bagging technique to all the basic algorithms. As can be seen, while the algorithm does improve on the basic algorithm, it does not perform as well as our voting technique.

One explanation for this relatively poor performance is based on the mistake-driven

Name	Δ(Va)	Δ(Vab)	Δ(Vabc)	Δ(Vabcd)	Δ(Vabcde)
Per	938 ± 22	833 ± 98	411 ± 81	842 ± 31	33 ± 18
ALMA(2)	780 ± 26	873 ± 80	358 ± 67	716 ± 33	65 ± 20
ALMA($\ln n$)	693 ± 21	50 ± 60	585 ± 43	511 ± 24	78 ± 21
Bal(1.05)	717 ± 20	144 ± 81	386 ± 64	775 ± 30	38 ± 20
Bal(1.2)	695 ± 24	50 ± 82	417 ± 54	703 ± 29	54 ± 19
Bal(1.4)	720 ± 20	148 ± 69	552 ± 62	654 ± 25	51 ± 13
Bal(1.7)	920 ± 25	1297 ± 87	707 ± 61	612 ± 30	32 ± 15
Bal(2.0)	806 ± 27	774 ± 88	778 ± 64	611 ± 29	9 ± 20
UWin(1.05)	1616 ± 54	1283 ± 145	394 ± 101	1389 ± 45	519 ± 29
UWin(1.2)	1300 ± 53	1719 ± 121	674 ± 95	959 ± 32	102 ± 21
UWin(1.4)	1216 ± 27	1866 ± 111	861 ± 84	740 ± 31	47 ± 20
UWin(1.7)	1306 ± 43	3687 ± 115	937 ± 71	572 ± 26	33 ± 18
UWin(2.0)	862 ± 24	609 ± 100	900 ± 91	506 ± 30	12 ± 23
CUWin(1.05)	1475 ± 54	214 ± 157	557 ± 102	6608 ± 51	60 ± 22
CUWin(1.2)	1108 ± 26	175 ± 80	800 ± 64	2002 ± 33	60 ± 16
CUWin(1.4)	1073 ± 24	556 ± 85	844 ± 59	1155 ± 27	50 ± 17
CUWin(1.7)	940 ± 28	16 ± 81	883 ± 64	833 ± 32	32 ± 15
CUWin(2.0)	786 ± 22	*−93* ± 82	770 ± 64	777 ± 21	50 ± 21
NWin(1.05,.3)	10660 ± 113	*−580* ± 120	842 ± 95	7610 ± 87	213 ± 30
NWin(1.2,.3)	3046 ± 66	432 ± 87	682 ± 72	2976 ± 44	198 ± 24
NWin(1.4,.3)	1637 ± 41	467 ± 83	678 ± 57	2085 ± 36	155 ± 21
NWin(1.7,.3)	1090 ± 33	985 ± 87	746 ± 62	1523 ± 30	116 ± 22
NWin(2.0,.3)	729 ± 23	*−26* ± 85	728 ± 70	1187 ± 39	96 ± 27
NWin(1.05,.5)	950 ± 43	281 ± 116	434 ± 89	1019 ± 41	1 ± 27
NWin(1.2,.5)	1000 ± 29	1191 ± 90	654 ± 67	595 ± 32	20 ± 18
NWin(1.4,.5)	785 ± 28	120 ± 62	671 ± 61	432 ± 29	26 ± 16
NWin(1.7,.5)	669 ± 21	*−111* ± 74	660 ± 60	362 ± 33	30 ± 21
NWin(2.0,.5)	771 ± 24	299 ± 75	650 ± 49	293 ± 33	39 ± 23
NWin(1.05,.7)	3522 ± 43	834 ± 111	390 ± 83	1215 ± 39	581 ± 33
NWin(1.2,.7)	1361 ± 26	234 ± 83	602 ± 69	657 ± 34	331 ± 25
NWin(1.4,.7)	1096 ± 26	907 ± 75	528 ± 47	504 ± 26	183 ± 21
NWin(1.7,.7)	953 ± 26	626 ± 73	574 ± 63	355 ± 30	126 ± 23
NWin(2.0,.7)	991 ± 25	1053 ± 104	542 ± 62	292 ± 32	60 ± 26
Combine	1504 ± 24	158 ± 70	331 ± 63	498 ± 25	1164 ± 40

Table 3.7: Difference in number of mistakes as voting modification A, B, C, D, E, F are incrementally added.

Name	Δ(Va)	Δ(Vab)	Δ(Vabc)	Δ(Vabcd)	Δ(Vabcde)
Per	43 ± 6	17 ± 11	43 ± 14	34 ± 5	2 ± 3
ALMA(2)	36 ± 5	86 ± 12	39 ± 10	32 ± 5	$\mathit{-1} \pm 3$
ALMA($\ln n$)	19 ± 6	$\mathit{-23} \pm 11$	54 ± 10	26 ± 6	3 ± 4
Bal(1.05)	45 ± 5	7 ± 11	51 ± 11	28 ± 6	4 ± 3
Bal(1.2)	39 ± 4	31 ± 11	48 ± 10	17 ± 6	4 ± 4
Bal(1.4)	30 ± 5	$\mathit{-7} \pm 13$	59 ± 11	25 ± 6	3 ± 3
Bal(1.7)	46 ± 5	52 ± 12	79 ± 12	22 ± 6	$\mathit{-1} \pm 4$
Bal(2.0)	48 ± 5	42 ± 14	82 ± 12	19 ± 5	$\mathit{-2} \pm 4$
UWin(1.05)	54 ± 7	76 ± 20	48 ± 12	72 ± 6	14 ± 5
UWin(1.2)	68 ± 6	100 ± 14	66 ± 12	31 ± 6	3 ± 3
UWin(1.4)	54 ± 6	70 ± 17	85 ± 14	27 ± 6	3 ± 3
UWin(1.7)	65 ± 6	227 ± 17	97 ± 11	29 ± 5	$\mathit{-2} \pm 3$
UWin(2.0)	40 ± 5	42 ± 16	102 ± 13	23 ± 7	$\mathit{-1} \pm 4$
CUWin(1.05)	27 ± 10	79 ± 23	58 ± 16	116 ± 10	1 ± 3
CUWin(1.2)	43 ± 5	44 ± 12	50 ± 11	27 ± 5	3 ± 3
CUWin(1.4)	40 ± 4	34 ± 13	61 ± 11	22 ± 5	2 ± 3
CUWin(1.7)	44 ± 4	11 ± 14	70 ± 12	19 ± 5	0 ± 4
CUWin(2.0)	38 ± 4	8 ± 11	69 ± 9	18 ± 4	2 ± 4
NWin(1.05,.3)	201 ± 16	$\mathit{-86} \pm 12$	69 ± 13	2 ± 13	20 ± 5
NWin(1.2,.3)	129 ± 9	76 ± 14	56 ± 9	40 ± 7	20 ± 4
NWin(1.4,.3)	52 ± 8	45 ± 16	60 ± 12	49 ± 7	10 ± 5
NWin(1.7,.3)	22 ± 5	66 ± 15	72 ± 11	34 ± 6	8 ± 5
NWin(2.0,.3)	26 ± 5	14 ± 12	78 ± 11	19 ± 5	4 ± 5
NWin(1.05,.5)	39 ± 6	2 ± 16	64 ± 12	50 ± 8	1 ± 4
NWin(1.2,.5)	49 ± 5	75 ± 15	48 ± 13	20 ± 6	2 ± 4
NWin(1.4,.5)	37 ± 5	33 ± 13	58 ± 11	11 ± 6	2 ± 4
NWin(1.7,.5)	26 ± 4	19 ± 14	61 ± 11	9 ± 7	1 ± 4
NWin(2.0,.5)	30 ± 4	30 ± 14	60 ± 11	10 ± 5	6 ± 4
NWin(1.05,.7)	121 ± 9	101 ± 12	69 ± 10	86 ± 7	34 ± 7
NWin(1.2,.7)	43 ± 6	6 ± 14	61 ± 12	27 ± 7	17 ± 6
NWin(1.4,.7)	43 ± 5	72 ± 12	53 ± 11	22 ± 6	7 ± 4
NWin(1.7,.7)	32 ± 4	53 ± 12	50 ± 11	18 ± 5	2 ± 4
NWin(2.0,.7)	45 ± 5	87 ± 13	52 ± 9	13 ± 6	0 ± 5
Combine	16 ± 6	19 ± 13	50 ± 9	35 ± 5	28 ± 8

Table 3.8: Difference in mistakes on last 500 trials as voting modification A, B, C, D, E, F are incrementally added.

Name	$\hat{\mathrm{M}}$(Name)	$\hat{\mathrm{M}}$(B-Name)	$\hat{\mathrm{M}}$(L-Name)	$\hat{\mathrm{M}}$(V-Name)
Per	73342 ± 80	57214 ± 95	54352 ± 108	**51295** ± 74
ALMA(2)	66066 ± 95	53455 ± 94	52838 ± 117	**50046** ± 93
ALMA($\ln n$)	71023 ± 105	57230 ± 96	55217 ± 90	**53301** ± 82
Bal(1.05)	72989 ± 83	57163 ± 82	53336 ± 107	**51275** ± 96
Bal(1.2)	72398 ± 83	56656 ± 94	52542 ± 111	**50621** ± 84
Bal(1.4)	74887 ± 86	57410 ± 88	52904 ± 99	**50779** ± 79
Bal(1.7)	81075 ± 109	60312 ± 93	55954 ± 118	**52386** ± 87
Bal(2.0)	86641 ± 89	63309 ± 95	57199 ± 113	**54221** ± 79
UWin(1.05)	93375 ± 93	85178 ± 80	70226 ± 179	**65027** ± 103
UWin(1.2)	88163 ± 93	75234 ± 86	64315 ± 152	**59561** ± 87
UWin(1.4)	88900 ± 87	71205 ± 90	62678 ± 131	**57949** ± 74
UWin(1.7)	94672 ± 103	72348 ± 84	65376 ± 144	**58841** ± 73
UWin(2.0)	100624 ± 89	74801 ± 87	63452 ± 104	**60564** ± 81
UCWin(1.05)	99476 ± 98	91143 ± 96	77527 ± 154	**68613** ± 94
UCWin(1.2)	85792 ± 91	75016 ± 93	63654 ± 86	**59509** ± 84
UCWin(1.4)	81676 ± 96	68452 ± 89	59746 ± 114	**56068** ± 92
UCWin(1.7)	81567 ± 95	66396 ± 87	57845 ± 114	**55141** ± 73
UCWin(2.0)	82557 ± 78	65833 ± 80	57579 ± 94	**55290** ± 77
NWin(1.05,.3)	110746 ± 76	105786 ± 105	105828 ± 158	**87083** ± 110
NWin(1.2,.3)	88760 ± 89	79049 ± 107	73787 ± 120	**66453** ± 87
NWin(1.4,.3)	85753 ± 96	72220 ± 93	65946 ± 110	**60923** ± 78
NWin(1.7,.3)	88529 ± 85	70067 ± 92	63792 ± 95	**59332** ± 76
NWin(2.0,.3)	92840 ± 97	70643 ± 89	62376 ± 119	**59661** ± 92
NWin(1.05,.5)	86721 ± 91	73936 ± 102	61630 ± 136	**58946** ± 145
NWin(1.2,.5)	80307 ± 102	66440 ± 86	58627 ± 124	**55167** ± 94
NWin(1.4,.5)	79405 ± 91	64004 ± 83	56055 ± 102	**54021** ± 80
NWin(1.7,.5)	81935 ± 91	63668 ± 93	55709 ± 86	**54098** ± 71
NWin(2.0,.5)	85610 ± 111	64638 ± 78	56979 ± 99	**54927** ± 74
NWin(1.05,.7)	86297 ± 97	81413 ± 89	71848 ± 143	**65306** ± 96
NWin(1.2,.7)	77574 ± 82	67591 ± 84	59102 ± 99	**55917** ± 87
NWin(1.4,.7)	76053 ± 93	63502 ± 85	56666 ± 109	**53446** ± 77
NWin(1.7,.7)	77541 ± 90	62003 ± 86	55338 ± 93	**52705** ± 71
NWin(2.0,.7)	80087 ± 92	62352 ± 86	56065 ± 107	**53127** ± 72

Table 3.9: Number of mistakes made by algorithms including on-line bagging algorithms on all data sets.

nature of the basic algorithms we use. A mistake-driven algorithm will ignore an instance that is predicted correctly. Therefore, updating the algorithm with k copies of an instance may not result in k changes. As soon as the algorithm gets a hypothesis that predicts the instance correctly the remaining updates are ignored.

In fact, this can make the algorithm much more susceptible to noisy instances. A noisy instance is likely to require more updates before it is correctly predicted, and therefore will be given more influence. In addition, this noisy instance will affect a large fraction of the voting hypotheses. Each hypothesis has a $1/e$ probability of getting $k = 0$ based on the Poisson distribution. Therefore most hypotheses will attempt to update at least one copy of the noisy instance. If a hypothesis is accurate, it has a high probability of making a mistake on the noisy instance. This update on a noisy instance can cause a loss of accuracy.[8] Having a majority of the voting hypotheses with either poor accuracy or updating on a noisy instance will cause the bagging algorithm to make additional mistakes.

However, it is clear that Bagged voting does improve the performance over the basic algorithm. It also has the advantage that all of its hypotheses are based on the current trial and have seen a significant fraction of the instances. This is beneficial when dealing with a concept that is changing over the trials. Voting with hypotheses from previous trials can degrade performance in this case since these older hypotheses may be unrelated to the current concept. We will return to this topic in Chapter 6 when we deal with on-line learning of changing concepts.

3.4.6 On-line Averaging

In this section, we consider the technique of Freund and Schapire [FS98] that predicts with an average weight vector of all the previous hypotheses used by the basic algorithm. We call this the hypothesis averaging modification. The technique works by giving the hypothesis in each previous trial equal weight in the average. Therefore, any hypothesis that is used in multiple trials will have a larger effect on the average. This tends to

[8] See Section 3.2 for more details.

favor hypotheses that are accurate since they are changed less often with mistake-driven algorithms.

Unfortunately, doing a direct average of the weights does not make sense for all algorithms. Any hypothesis used by an algorithm can be multiplied by a constant without affecting the predictions of the algorithm. An algorithm might use a hypothesis during certain trials that effectively has a large normalization constant. During other trials the normalization constant might be much smaller. This causes the trials with the small normalization to have a much smaller effect on the average. This weighting can be undesirable if these trials have a hypothesis that is more accurate.

To help resolve this problem, we consider two averaging algorithms. The first algorithm, A1, does a direct average using the weight vectors provided by the basic algorithm. The second algorithm, A2, normalizes the weights before the average. The normalization consists of multiplying all the weights by a positive constant that ensures the infinity norm of the new weights is 1. In addition to applying these two modification to all 33 basic algorithms, we also created new Combine algorithms based on averaging. The A1-Combine algorithm averages using the unmodified weights of the current hypothesis; however, it always uses the hypothesis of the basic algorithm that is currently making the fewest mistakes. The A2-Combine algorithm is similar. It uses the hypothesis from the best basic algorithm, but A2-Combine normalizes the hypothesis before the hypothesis is added to the average.

The results of these techniques can be seen in Table 3.10. The table reports the total sum of mistakes over all 186 concept. As can be seen, for most algorithms the voting technique gives the best performance. However, for some basic algorithms the A1 modification is competitive. This is especially true for the Balanced Winnow algorithm.

The A2 modification is more disappointing. We had hoped this technique would resolve any normalization issues in the basic algorithm and give performance close to that of voting. There are some cases where A2 does improve on the A1 algorithm; however, A2 always does worse than voting. In particular, we had hoped the A2-Combine algorithm would see the same performance increase as V-Combine. Unfortunately, while A2-Combine greatly improves the A1-Combine algorithm, the different hypotheses used

Name	$\hat{\mathrm{M}}$(A1-Name)	$\hat{\mathrm{M}}$(A2-Name)	$\hat{\mathrm{M}}$(V-Name)
Per	52519 ± 91	58463 ± 105	**51295** ± 74
ALMA(2)	53515 ± 89	55620 ± 98	**50046** ± 93
ALMA($\ln n$)	56663 ± 102	61741 ± 151	**53301** ± 82
Bal(1.05)	52224 ± 93	58494 ± 109	**51275** ± 96
Bal(1.2)	50957 ± 89	59416 ± 117	**50621** ± 84
Bal(1.4)	**50582** ± 86	60878 ± 123	50779 ± 79
Bal(1.7)	**52217** ± 88	63309 ± 135	52386 ± 87
Bal(2.0)	54737 ± 93	65594 ± 148	**54221** ± 79
UWin(1.05)	84054 ± 95	73200 ± 88	**65027** ± 103
UWin(1.2)	64635 ± 80	62932 ± 138	**59561** ± 87
UWin(1.4)	60947 ± 131	59625 ± 126	**57949** ± 74
UWin(1.7)	59790 ± 77	59560 ± 97	**58841** ± 73
UWin(2.0)	61495 ± 82	60656 ± 93	**60564** ± 81
UCWin(1.05)	97969 ± 166	80647 ± 169	**68613** ± 94
UCWin(1.2)	66239 ± 101	65599 ± 128	**59509** ± 84
UCWin(1.4)	60357 ± 93	62459 ± 228	**56068** ± 92
UCWin(1.7)	59302 ± 85	59433 ± 188	**55141** ± 73
UCWin(2.0)	59160 ± 87	59779 ± 265	**55290** ± 77
NWin(1.05,.3)	207839 ± 711	134147 ± 306	**87083** ± 110
NWin(1.2,.3)	103805 ± 416	77754 ± 132	**66453** ± 87
NWin(1.4,.3)	81723 ± 238	66791 ± 116	**60923** ± 78
NWin(1.7,.3)	73253 ± 190	63247 ± 106	**59332** ± 76
NWin(2.0,.3)	70703 ± 171	62790 ± 103	**59661** ± 92
NWin(1.05,.5)	64485 ± 342	60751 ± 157	**58946** ± 145
NWin(1.2,.5)	60851 ± 320	57309 ± 89	**55167** ± 94
NWin(1.4,.5)	58393 ± 276	55992 ± 90	**54021** ± 80
NWin(1.7,.5)	57631 ± 234	55778 ± 78	**54098** ± 71
NWin(2.0,.5)	58493 ± 211	56318 ± 87	**54927** ± 74
NWin(1.05,.7)	77285 ± 79	82904 ± 124	**65306** ± 96
NWin(1.2,.7)	58850 ± 85	61405 ± 85	**55917** ± 87
NWin(1.4,.7)	53486 ± 68	56140 ± 69	**53446** ± 77
NWin(1.7,.7)	52295 ± 72	54378 ± 81	**52705** ± 71
NWin(2.0,.7)	53252 ± 67	54429 ± 79	**53127** ± 72
Combine	94238 ± 2755	60185 ± 1377	**46432** ± 86

Table 3.10: Number of mistakes made by algorithms including on-line averaging algorithms on all data sets.

in A2-Combine do not mix as well as V-Combine. Perhaps a different more complicated normalization scheme would give a better average.

While the performance of these averaging modifications is still below the performance of voting, there are places where the averaging techniques are desirable. The main advantage of hypotheses averaging is reduced computational cost. Since the averaging algorithm only needs to maintain a single hypothesis, the cost of prediction and updating for the linear-threshold algorithms covered in this dissertation is just $O(n)$ per trial. This can be significantly faster than the $O(hm_t)$ cost of the voting prediction.[9] Unfortunately, when dealing with sparse instances, a straightforward implementation of the averaging modification still needs to spend $O(n)$ per trial in order to compute the new average. We are unsure if this can be improved to take advantage of sparse instances.

3.4.7 On-line Best Hypothesis

The final technique we compare against is not a voting technique. In [Gal90], Gallant uses the most accurate hypothesis currently generated by the on-line algorithm to make a prediction. The accuracy is measure by using the number of instances the current hypothesis has predicted correctly. This strategy only makes sense for a mistake-driven algorithm since such algorithms repeatedly use a hypothesis until a mistake is made. In our experiments, we refine this strategy using a set of r recent instances to help estimate the accuracy. This is the same strategy as using in modification C of Section 3.3.4. We call this technique the best hypothesis modification.

Table 3.11 gives the total number of mistakes over all 186 concept for three version of the best hypothesis modification. The H1 algorithm sets $r = 0$ and only uses the number of correct predictions made by the current hypothesis to estimate accuracy. The H2 modification uses 100 recent instances to improve the accuracy estimate. H3 uses 3000 instances to give an estimate of the performance limit of this technique. Finally, we include our main voting algorithm with the default parameters for comparison

[9] Recall that n is the number of attributes and m_t is the number of nonzero attributes on trial t.

purposes. Also, we include a Combine version for all algorithms. For the best hypothesis algorithm, Combine works by predicting with the hypothesis of the best hypothesis algorithm that is currently making the fewest mistakes.

As can be seen in Table 3.11, the H3 modification with 3000 instances is always the best algorithm. Unfortunately, it can be expensive in both time and space since it needs store and predict with the 3000 saved instances. The more modest algorithms, H1 and H2, always do significantly worse than the full voting algorithm.

It is interesting to note that if one can find the most accurate hypotheses used by the basic algorithm that this hypothesis makes fewer mistakes, on average, than the voting technique. However, if one looks at the individual concepts, one finds that the best hypothesis technique is not always better than the voting technique. There are still several concepts where voting does improve on the best hypothesis.

3.5 Summary

In this chapter, we gave a technique that modifies adversarial on-line algorithms to improve their performance on problems where instances are sampled from a distribution. Our technique modifies an existing on-line algorithm B to generate a new algorithm we call V-B. V-B is a voting procedure that combines several hypotheses to make a more accurate prediction.

We explained V-B in terms of five separate modifications. The purpose these modifications is to increase the accuracy of prediction by helping to keep the voting procedure full of accurate and diverse hypotheses. These modifications are inexpensive and increase the cost of the algorithms used in this dissertation roughly by a factor equal to the number of hypothesis used for voting. We performed experiments that show our technique lowers the number of mistakes on real world data sets. The improvement is consistent over a wide range of problem types.

The main algorithm of this chapter is called V-Combine. V-Combine is similar to V-B but it selects hypotheses for its voting procedure from a set of basic algorithms. V-Combine selects its hypothesis from the basic algorithm that is currently making

Name	$\hat{M}$(H1-Name)	$\hat{M}$(H2-Name)	$\hat{M}$(H3-Name)	$\hat{M}$(V-Name)
Per	60368 ± 85	55238 ± 63	**49510** ± 60	51295 ± 74
ALMA(2)	57439 ± 107	53469 ± 95	**48739** ± 87	50046 ± 93
ALMA($\ln n$)	61785 ± 104	57418 ± 88	**51856** ± 98	53301 ± 82
Bal(1.05)	60191 ± 95	54971 ± 81	**49322** ± 81	51275 ± 96
Bal(1.2)	60080 ± 107	54958 ± 89	**49090** ± 81	50621 ± 84
Bal(1.4)	61874 ± 103	56510 ± 92	**50006** ± 79	50779 ± 79
Bal(1.7)	66175 ± 114	59919 ± 95	**52132** ± 91	52386 ± 87
Bal(2.0)	70048 ± 105	62924 ± 90	**54135** ± 93	54221 ± 79
UWin(1.05)	71655 ± 112	65781 ± 121	**59558** ± 90	65027 ± 103
UWin(1.2)	67454 ± 102	60775 ± 92	**54535** ± 83	59561 ± 87
UWin(1.4)	68265 ± 106	61041 ± 88	**54055** ± 81	57949 ± 74
UWin(1.7)	72365 ± 113	64439 ± 91	**56085** ± 66	58841 ± 73
UWin(2.0)	76611 ± 95	67710 ± 95	**58167** ± 99	60564 ± 81
UCWin(1.05)	77404 ± 100	71242 ± 107	**66048** ± 97	68613 ± 94
UCWin(1.2)	67718 ± 113	61873 ± 100	**56296** ± 85	59509 ± 84
UCWin(1.4)	65671 ± 106	59749 ± 92	**53591** ± 79	56068 ± 92
UCWin(1.7)	66752 ± 93	60666 ± 77	**53687** ± 75	55141 ± 73
UCWin(2.0)	68222 ± 100	61999 ± 82	**54373** ± 89	55290 ± 77
NWin(1.05,.3)	96776 ± 96	90664 ± 100	**84454** ± 92	87083 ± 110
NWin(1.2,.3)	75350 ± 112	69651 ± 81	**63317** ± 86	66453 ± 87
NWin(1.4,.3)	71870 ± 97	65912 ± 99	**58922** ± 92	60923 ± 78
NWin(1.7,.3)	73126 ± 115	66549 ± 79	**58444** ± 88	59332 ± 76
NWin(2.0,.3)	75929 ± 120	68354 ± 101	**59247** ± 97	59661 ± 92
NWin(1.05,.5)	67683 ± 116	61991 ± 116	**55397** ± 111	58946 ± 145
NWin(1.2,.5)	64734 ± 101	59277 ± 92	**52829** ± 74	55167 ± 94
NWin(1.4,.5)	64957 ± 100	59436 ± 92	**52532** ± 74	54021 ± 80
NWin(1.7,.5)	67487 ± 95	61499 ± 92	**53609** ± 77	54098 ± 71
NWin(2.0,.5)	70289 ± 106	63710 ± 100	**54916** ± 91	54927 ± 74
NWin(1.05,.7)	71353 ± 112	66645 ± 94	**61505** ± 94	65306 ± 96
NWin(1.2,.7)	64335 ± 106	59405 ± 90	**53468** ± 85	55917 ± 87
NWin(1.4,.7)	63456 ± 101	58381 ± 89	**51990** ± 79	53446 ± 77
NWin(1.7,.7)	65067 ± 100	59622 ± 85	**52247** ± 82	52705 ± 71
NWin(2.0,.7)	67272 ± 106	61286 ± 96	**53101** ± 89	53127 ± 72
Combine	55132 ± 97	51205 ± 83	**46277** ± 75	46432 ± 86

Table 3.11: Number of mistakes made by algorithms including on-line best hypothesis algorithms on all data sets.

the fewest mistakes. This allows V-Combine to efficiently select the best of the basic algorithms. In our experiments, the number of mistakes made by V-Combine is close to the minimum number of mistakes made by V-B where B is chosen from the set of basic algorithms used by V-Combine. Sometimes V-Combine even surpasses this minimum.

Chapter 4

Improving On-line Learning with Instance Recycling

In this chapter, we continue the theme started in the last chapter. Our goal is to modify algorithms that have performance guarantees against adversaries so that they perform well against something weaker than an omniscient adversary. While in the last chapter we assumed that the instances are generated by a fixed distribution, in this chapter, we are a little more flexible.

Assume B is an on-line algorithm. Our new algorithm is called R-B and works by saving and learning from old instances as if they were new trials. It is similar to algorithm B, but R-B keeps an array of recently seen instances to use for recycling. For each old instance, the algorithm keeps track of how many times the instance is used to change the state of the basic algorithm. When an instance changes the state, we say the instance has been used. A user specified parameter u sets a maximum on the number of times an instance can be used.

The R-B algorithm is useful for more than just distribution-based problems. For example, consider setting $u = 1$ so that each instance can only be used once. Anytime R-B does not use a new instance, the instance can be saved for a future update. If a subsequent hypothesis predicts this saved instance incorrectly, the learning algorithm can process the instance as if it is a new trial. The algorithm now makes a mistake on the saved instance, but it is not a real mistake since the trial did not come from the on-line environment. However, the adversarial mistake bound does not make this distinction; the number of mistakes that algorithm can make on real trials is lowered by one. Therefore, any problem where algorithm B does not use every instance could benefit from the R-B modification.

The R-B algorithm is designed for inexpensive on-line algorithms that have a

mistake-bound for an adversary. The algorithms we have tested all allow instances $x_t \in [0,1]^n$ and use linear-threshold functions to represent hypotheses. The time cost for these algorithms is $O(m_t)$ per trial where m_t is the number of non-zero attributes during trial t.[1] As before, a major constraint of our techniques is to keep the computational advantage of these inexpensive algorithms. This is necessary for problems with a large number of attributes.

The remainder of this chapter is organized as follows. In Section 4.1, we cover previous research on recycling instances and using extra instances for on-line learning. In Section 4.2, we give the formal statement of our instance recycling algorithm. Next, in Section 4.4, we justify why instance recycling works, focusing particularly on instances generated by a fixed distribution. Section 4.3 gives the results of experiments with instance recycling using the experimental framework of Chapter 3.

4.1 Previous Research

Instance recycling has typically been used when applying on-line algorithms in a batch setting. In this case, a fixed sized set of instances is repeatedly cycled over until an accurate hypothesis has been learned [DH73]. For on-line learning, Littlestone proposed the idea to recycle over all old instances after every update but to only allow each instance to be used once for an update [Lit96]. While he achieved good performance with this technique, he considered the computational cost too expensive. Here we modify that technique to keep a recent window of instances and allow more than one update with an instance. This window technique allows us to reduce the computational cost while the additional updates gives the algorithm some extra flexibility.

Another technique to improve the performance of mistake-driven linear-threshold algorithms is to modify the algorithm to update when an instance is close to being incorrect [Ros62, LL00]. More precisely, when an instance is close to the current hyperplane, update the algorithm even if the instance is correctly predicted. The closeness is usually a parameter, δ', specified by the user of the algorithm. This parameter is

[1] More details can be found in Appendix D.

often called the margin of the algorithm.[2] This technique does not improve the adversarial mistake bound of the algorithm, but may improve performance on problems where instances are generated by something weaker than an adversary.[3] However, on experiments with artificial data, we have found the value of δ' to be difficult to set. Therefore, we only use the mistake-driven version of the linear-threshold algorithms in our experiments.

4.2 Instance Recycling Algorithm

Instance recycling is a simple extension of an on-line algorithm. Assume B is a basic on-line algorithm. We create a new algorithm R-B that recycles old instances by running algorithm B with additional steps added to the update procedure. Instance recycling works by saving the most recent s instances. Anytime the state of the algorithm changes because of an update, we iterate over the s old instances as if they are new trials. This includes updating with the old instances. Every time we update with an instance, we record whether it changes the state of the basic algorithm. If it changes the state we say that the instance has been used. Each instance is only allowed to be used u times where u is a parameter set by the user. We only stop iterating over the old instances when updates that have been used less than u times no longer change the state of the basic algorithm. The process repeats on the next new instance that changes the algorithm's state. Since all the algorithms in this dissertation are mistake-driven, they only change their state on a mistake.

There are two types of mistakes made by algorithm R-B. Any mistake on a new instance is a real mistake made on the learning problem. Any mistake on an old instance is not counted as a real mistake since it is entirely internal to the algorithm. For this reason, we call mistakes on old instances internal mistakes.

This transformation is based on an idea proposed by Littlestone [Lit96]. The difference is Littlestone used all old instances instead of a shifting window of recent instances,

[2] See Section 2.3.5 for more details.

[3] See Appendix F.

and he only allowed each instance to be used for a single update. As we will see, we get good performance with a smaller increase in computational cost using our more general transformation.

The cost of this recycling technique is small enough to keep the on-line learning algorithms we use in this thesis efficient. We use an extra $O(sm)$ space to store the saved instances[4] in a sparse format. We break the time complexity up into the cost for prediction and updating; these costs include the cost of the basic algorithm. The cost of the prediction step is the same as the basic algorithm. Updates are more complicated. At most, the algorithm can perform a basic algorithm update on an instance u times. This means the algorithm updates at most uT times. For basic algorithms that can update in $O(m)$ time, the updates take $O(umT)$ time. This includes most of the algorithms in this thesis. In practice, we expect this cost to be only a fraction of this since many of the instance will not be used for the maximum number of updates. For a mistake-driven algorithm, when $u = 1$, the number of updates is at most $\min\{M, T\}$ where M is the mistake bound of the basic algorithm. When $u > 1$, we may increase the number of updates with noisy instances. This could increase the mistake bound because of the extra noise.

The update step also includes some predictions. After a basic algorithm update, we search the old instance list for an instance to use in an update. Since we treat these old instances as new trials, we need to make predictions on them. Based on the recycling algorithm, we can make at most s predictions until we either stop recycling or update on an old instance. Therefore, the maximum number of old instance predictions depends on s and the number of updates. Since there are at most uT updates, there are at most usT predictions from the old instances. Since all of our basic algorithms take $O(m)$ for a prediction, this gives a total cost of $O(usmT)$. The cost of old instance predictions dominates the cost of the algorithm, so the time complexity of the algorithm is $O(usmT)$. In practice, the number of predictions is only a fraction of this number since we generally have fewer updates.

[4]These saved instance can also be used for hypothesis accuracy estimation in voting.

We may want to run multiple basic algorithms and apply instance recycling to each algorithm. Generally, the goal will be to make a number of mistakes close to the best from the set of algorithms. For example, we can run the WMA algorithm as described in Section 2.5 to get performance close to the best of the individual algorithms. However, if we use instance recycling on v algorithms then the cost will increase by a factor of v. Instead, in Chapter 5, we show how combining instance recycling with voting to perform algorithm/parameter selection.

4.3 Instance Recycling Experiments

In this section, we give experiments to show how instance recycling improves the performance of basic algorithms. Our experimental methodology is identical to that used in Chapter 3. Please see Section 3.4 for details on the basic algorithms, data sets, and experiments.

For a given on-line algorithm B, we refer to the algorithm that uses recycled instances as R-B. For all the algorithms in this chapter, we set s the number of recycling instances to 100 and u the number of times we can update an instance to 1. These values are chosen as a baseline to judge performance and to keep the algorithm inexpensive. In Section 5.2.2, we perform experiments where we test different values for these parameters.

Our first results compare a basic algorithms B to the instance recycling algorithm algorithm R-B. In Figure 4.1, we give a scatter plot that contains a point for each algorithm/concept pair. This gives a total of 6138 points. The y coordinate of each points corresponds to the final number of mistakes made by a basic algorithm on the particular concept; the x coordinate corresponds to the final number of mistakes made by the instance recycling version of the basic algorithm. All but one point is above the $y = x$ line. This single point corresponds to 0.1 extra mistakes made by the recycling version of Normalized Winnow(1.2,.3) on a single concept.

Our remaining results focus on the performance of the individual algorithms. In Table 4.1, we give the total number of mistakes over all 186 concepts when $s = 0$ and

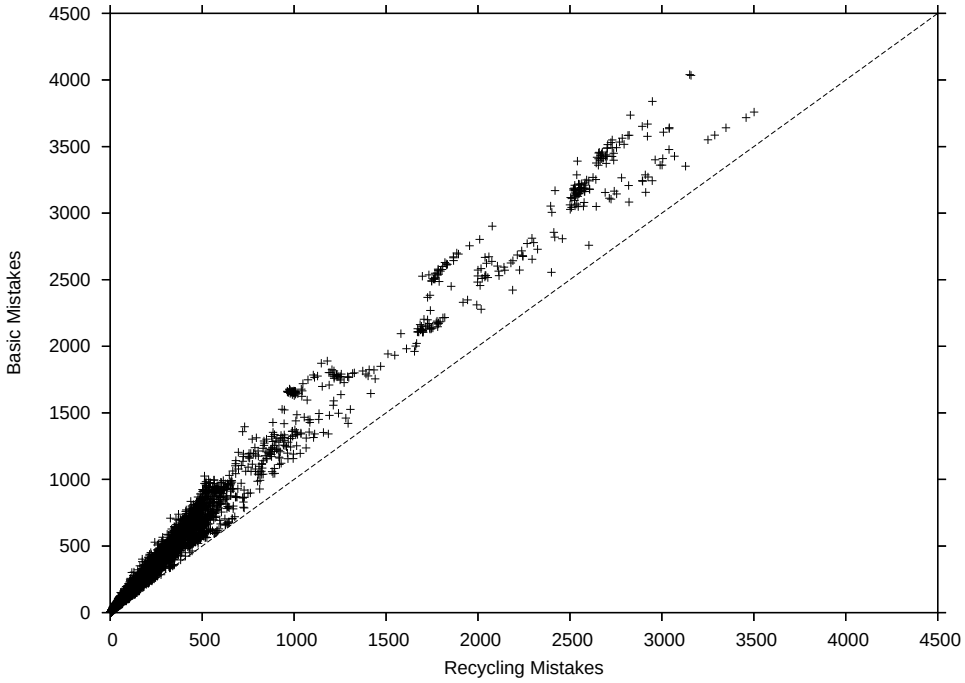

Figure 4.1: Scatter plot comparing basic algorithm to recycled algorithm.

$s = 100$. Table 4.2 uses the same experiments but only reports the sum of mistakes on the last 500 trials of each concept. As can be seen, the recycling consistently improves performance across all the algorithms.

The best algorithm in Table 4.1 is R-ALMA(2). Currently, the only algorithm that makes fewer mistakes is V-Combine from Section 3.3. The superiority of V-Combine is most likely a result of its ability to select hypotheses from any of the basic algorithms. A similar advantage should be possible with the instance recycling algorithms. The first column of Table 4.3 gives a count of the number of concepts where the corresponding algorithm makes the minimum number of mistakes. While it's not surprising that R-ALMA(2) performs best on the largest number of concepts, it does not have the best performance on all concepts. This suggests that a combination of these recycling algorithms should make fewer mistakes. While it is possible to create a straightforward algorithm to combine all of these recycling algorithms using a tool such as WMA,[5] we use the V-Combine algorithm as a way to get the benefits of instance recycling

[5]See Section 2.5 for more details.

Name	$\hat{\mathrm{M}}$(Name)	$\hat{\mathrm{M}}$(R-Name)
Per	73342 ± 80	**50926** ± 76
ALMA(2)	66066 ± 95	**48914** ± 72
ALMA($\ln n$)	71023 ± 105	**53091** ± 89
Bal(1.05)	72989 ± 83	**50782** ± 77
Bal(1.2)	72398 ± 83	**50888** ± 76
Bal(1.4)	74887 ± 86	**53703** ± 91
Bal(1.7)	81075 ± 109	**58924** ± 86
Bal(2.0)	86641 ± 89	**62837** ± 85
UWin(1.05)	93375 ± 93	**63041** ± 79
UWin(1.2)	88163 ± 93	**58735** ± 68
UWin(1.4)	88900 ± 87	**60417** ± 73
UWin(1.7)	94672 ± 103	**64862** ± 72
UWin(2.0)	100624 ± 89	**69016** ± 83
CUWin(1.05)	99476 ± 98	**66768** ± 71
CUWin(1.2)	85792 ± 91	**57946** ± 70
CUWin(1.4)	81676 ± 96	**56905** ± 81
CUWin(1.7)	81567 ± 95	**58406** ± 71
CUWin(2.0)	82557 ± 78	**60109** ± 79
NWin(1.05,.3)	110746 ± 76	**85230** ± 72
NWin(1.2,.3)	88760 ± 89	**64192** ± 74
NWin(1.4,.3)	85753 ± 96	**61872** ± 73
NWin(1.7,.3)	88529 ± 85	**63935** ± 75
NWin(2.0,.3)	92840 ± 97	**66664** ± 73
NWin(1.05,.5)	86721 ± 91	**57966** ± 82
NWin(1.2,.5)	80307 ± 102	**55295** ± 70
NWin(1.4,.5)	79405 ± 91	**56125** ± 63
NWin(1.7,.5)	81935 ± 91	**59080** ± 71
NWin(2.0,.5)	85610 ± 111	**61877** ± 75
NWin(1.05,.7)	86297 ± 97	**62901** ± 81
NWin(1.2,.7)	77574 ± 82	**55290** ± 75
NWin(1.4,.7)	76053 ± 93	**54890** ± 73
NWin(1.7,.7)	77541 ± 90	**56953** ± 68
NWin(2.0,.7)	80087 ± 92	**59205** ± 83
Majority Label	124266 ± 50	

Table 4.1: Number of mistakes for all algorithms with 0 recycled instances and 100 recycled instances

Name	$\hat{\mathrm{M}}$(Name)	$\hat{\mathrm{M}}$(R-Name)
Per	4734 ± 22	$\mathbf{3408} \pm 20$
ALMA(2)	4304 ± 18	$\mathbf{3278} \pm 17$
ALMA($\ln n$)	4590 ± 25	$\mathbf{3539} \pm 19$
Bal(1.05)	4708 ± 22	$\mathbf{3396} \pm 18$
Bal(1.2)	4700 ± 21	$\mathbf{3439} \pm 17$
Bal(1.4)	4979 ± 23	$\mathbf{3719} \pm 21$
Bal(1.7)	5522 ± 24	$\mathbf{4145} \pm 19$
Bal(2.0)	5960 ± 30	$\mathbf{4443} \pm 19$
UWin(1.05)	5739 ± 23	$\mathbf{3991} \pm 17$
UWin(1.2)	5494 ± 20	$\mathbf{3849} \pm 18$
UWin(1.4)	5745 ± 22	$\mathbf{4093} \pm 18$
UWin(1.7)	6317 ± 25	$\mathbf{4493} \pm 18$
UWin(2.0)	6830 ± 25	$\mathbf{4823} \pm 20$
CUWin(1.05)	5758 ± 23	$\mathbf{3989} \pm 18$
CUWin(1.2)	5215 ± 25	$\mathbf{3717} \pm 18$
CUWin(1.4)	5181 ± 23	$\mathbf{3789} \pm 19$
CUWin(1.7)	5359 ± 21	$\mathbf{4010} \pm 16$
CUWin(2.0)	5516 ± 21	$\mathbf{4168} \pm 20$
NWin(1.05,.3)	6022 ± 29	$\mathbf{4633} \pm 22$
NWin(1.2,.3)	5301 ± 21	$\mathbf{3898} \pm 17$
NWin(1.4,.3)	5392 ± 25	$\mathbf{3995} \pm 20$
NWin(1.7,.3)	5787 ± 21	$\mathbf{4297} \pm 17$
NWin(2.0,.3)	6224 ± 20	$\mathbf{4560} \pm 18$
NWin(1.05,.5)	5300 ± 20	$\mathbf{3769} \pm 19$
NWin(1.2,.5)	5077 ± 19	$\mathbf{3684} \pm 16$
NWin(1.4,.5)	5131 ± 22	$\mathbf{3820} \pm 20$
NWin(1.7,.5)	5460 ± 24	$\mathbf{4101} \pm 18$
NWin(2.0,.5)	5804 ± 24	$\mathbf{4344} \pm 17$
NWin(1.05,.7)	5327 ± 22	$\mathbf{3989} \pm 19$
NWin(1.2,.7)	4955 ± 21	$\mathbf{3655} \pm 17$
NWin(1.4,.7)	4966 ± 21	$\mathbf{3740} \pm 21$
NWin(1.7,.7)	5212 ± 23	$\mathbf{3972} \pm 18$
NWin(2.0,.7)	5451 ± 23	$\mathbf{4171} \pm 18$
Majority Label	9404 ± 14	

Table 4.2: Number of mistakes on last 500 trials for all algorithms with 0 recycled instances and 100 recycled instances

algorithm, voting, and multiple algorithms in one algorithm. This topic is covered in depth in the next chapter.

4.4 Discussion

Instance recycling is effective for a variety of reasons. First, as discussed before, the algorithms we consider have a bound on the total number of mistakes for instances generated by an adversary. Based on the recycling update procedure, this bound is on the total number of mistakes including internal mistakes. Therefore internal mistakes can potentially reduce the number of mistakes on new instances. It is important that the mistake bound on the basic algorithm is true for any distribution since the recycling technique can change the distribution of the instances. This is not a problem for algorithms with mistake bounds against adversaries.

Recycling is particularly effective for mistake-driven algorithms. Mistake-driven algorithms skip many instances because they are predicted correctly. In a sense, these instances are wasted because they have no effect on a mistake-driven algorithm. However, given the instability of the accuracy of these algorithms as seen in Section 3.2, a past instance may no longer be predicted correctly after several updates. Therefore it is useful to recycle these old instances for more updates. An update from an old instance may even have a cascading effect. Another old instance may be predicted incorrectly causing further updates. These internal mistakes can help lower the number of real mistakes.

Our last reason recycling gives good performance is that it removes instability in the accuracy of the current hypothesis of the basic algorithm.[6] Recycling is finished when there are no more possible updates. This means that either the update count for each instance is at its maximum or all the old instances that are not at their maximum are predicted correctly. If we assume that most instances are at their maximum then the algorithm has made a large number of internal mistakes, which should help lower the number of real mistakes. If many of the instances are not at their maximum count

[6] See Section 3.2.

Name	# Name best concepts	# R-Name best concepts
Perceptron	0	0
ALMA(2)	56	60
ALMA($\ln n$)	17	8
Bal(1.05)	0	2
Bal(1.2)	1	0
Bal(1.4)	2	7
Bal(1.7)	0	0
Bal(2.0)	0	0
UWin(1.05)	2	3
UWin(1.2)	0	3
UWin(1.4)	0	1
UWin(1.7)	0	0
UWin(2.0)	0	1
UCWin(1.05)	0	0
UCWin(1.2)	0	1
UCWin(1.4)	0	1
UCWin(1.7)	0	1
UCWin(2.0)	0	1
NWin(1.05,.3)	2	0
NWin(1.2,.3)	0	0
NWin(1.4,.3)	0	0
NWin(1.7,.3)	1	0
NWin(2.0,.3)	0	0
NWin(1.05,.5)	0	9
NWin(1.2,.5)	0	4
NWin(1.4,.5)	1	2
NWin(1.7,.5)	0	2
NWin(2.0,.5)	1	7
NWin(1.05,.7)	39	2
NWin(1.2,.7)	21	21
NWin(1.4,.7)	6	17
NWin(1.7,.7)	8	12
NWin(2.0,.7)	29	21

Table 4.3: Number of concepts where the algorithm gives the minimum number of mistakes.

then these instances are predicted correctly by the current hypothesis. Therefore, we must have a hypothesis that is accurate for a sample of recent instances. In practice, we find that many of the instances do not reach the maximum count and therefore the recycling often returns more accurate hypotheses than the basic algorithm.

The parameter u, that controls the maximum number of times we can reuse an instance, is useful when dealing with the effects of noise. If we know there is no noise in the concept then setting $u = \infty$ should give the best results. The algorithm recycles over the old instances until the hypothesis is consistent with these instances, and every internal mistake can help lower the number of real mistakes. As noise is added to the problem, internal mistakes on a noisy instance can mislead the algorithm and increase the number of mistakes. Recycling with a large u value tends to focus on the noisy instances, attempting to predict them correctly. This has the potential to greatly increase the number of mistakes. If $u = \infty$ this can even cause the algorithm to enter an infinite loop because there may be not be a target function that correctly predicts all the old instances.[7] However when $u = 1$, for the algorithms in the thesis, instance recycling has the same mistake bound against an adversary as the basic algorithm.[8] For something weaker than an adversary, such as a distribution, instance recycling can decrease the mistake bound, even when $u > 1$, if the number of internal mistakes offsets the loss of progress from noisy instances.

4.5 Summary

In this chapter, we gave another technique to improve the performance of adversarial on-line algorithms. Assume B is an on-line algorithm. Our new algorithm is called R-B and it works by recycling over saved old instances as if they were new trials. For each old instance, the algorithm keeps track of how many times it has been used to change the state of the basic algorithm. A user specified parameter, u, sets a maximum on the

[7] Only when $s \leq n$ can we guarantee that a hyperplane must exist that separates the s old instances [DH73].

[8] This is based on the adversary being able to generate the instances in any order. For more information see Appendix F.

number of times an instance can change the state. The R-B algorithm is efficient with only a modest increase in cost over the basic algorithms used in this dissertation.

Algorithm R-B, with $u = 1$, has the same mistake bound as algorithm B when instances are generated by an adversary. However, it can improve performance for something weaker than an adversary. A popular type of problem that is weaker than an adversary is based on a fixed distribution generating the instances. In this chapter, we performed experiments on the same fixed distribution problems introduced in Chapter 3. The R-B showed significant and consistent reductions in the number of mistakes over the basic algorithms. Some of the R-B algorithms even surpassed the voting algorithms of Chapter 3.

Chapter 5

Combining Voting and Recycling

The purpose of this chapter is to combine the techniques from the previous two chapters. In Chapter 3 and Chapter 4, we give techniques to improve the performance of algorithms that have a mistake-bound against an adversary for problems where the instances are generated by something weaker than an adversary. In particular, we focus on a fixed distribution generating the instances. While the linear-threshold algorithms used in this thesis are guaranteed to perform well against an adversary, for many practical learning problems, a distribution is a more realistic model for instance generation.

Both techniques modify an existing basic on-line algorithm to generate a new algorithm. The first technique, from Chapter 3, uses multiple hypotheses to replace the prediction strategy of the basic on-line algorithm. An on-line algorithm naturally generates new hypotheses as it changes its current hypothesis during an update. We periodically save some of these hypotheses. Instead of predicting with the current hypothesis, a vote of the saved hypotheses is used for predictions.

The second technique, from Chapter 4, recycles over saved old instances as if they were new trials. It is similar to the basic algorithm, but it keeps an array of recent instances to use for the recycling. For each old instance, the algorithm keeps track of how many times it has been used by an update. A user specified parameter sets a maximum on this number.

These techniques can be used together. Given an on-line algorithm B, one can apply the recycling technique to form algorithm R-B. Algorithm R-B can then be used as a basic algorithm to supply hypotheses to the voting procedure. This generates algorithm VR-B. Our full combined algorithm allows many basic algorithms and is based on modification F from Section 3.3.7. In this case, a set of basic algorithms

with recycling are used with voting to create the VR-Combine algorithm. This is the algorithm we recommend for on-line learning when instances are generated by fixed distributions.

Our techniques are designed for inexpensive on-line algorithms. The algorithms we have tested all allow instances, $x_t \in [0,1]^n$ and use linear-threshold functions to represent hypotheses. The time cost for these algorithms is $O(m_t)$ per trial where m_t is the number of non-zero attributes during trial t.[1] A major constraint of our techniques is to keep the computational advantage of these inexpensive algorithms. This is necessary for problems with a large number of attributes.

In this chapter, we give the results of experiments using the same data sets from Section 3.4. The experiments combine the voting and recycling technique using the linear-threshold algorithms from Chapter 2 as basic algorithms. In addition, we experimentally explore the parameter space of these techniques.

This chapter is organized as follows. In Section 5.1, we describe the technique used to combine voting and instance recycling and analyze its cost. In Section 5.2, we give experiments with the combined technique. This includes experiments on the parameters used in the voting and instance recycling. Section 5.3 gives a limited comparison with linear Support Vector Machines [CV95]. The final section is a summary of our results.

5.1 Voting and Recycling Algorithm

The algorithm that combines voting and recycling is straightforward given the voting technique in Chapter 3 and the instance recycling technique from Chapter 4. Assume B is an on-line learning algorithm. Applying instance recycling to this algorithm generates algorithm R-B. Using R-B as a basic algorithm for a voting procedure generates algorithm VR-B. This is the default notation if we use all the voting modifications from Section 3.3. If we only use some of the voting modifications we can prefix these modifications. For example, if we only use voting modifications A, B, and C then the algorithm would be called VRabc-B.

[1] More details can be found in Appendix D.

The cost of the combined algorithm is based on cost of the instance recycling and voting techniques. This information can be found in their respective chapters. As an example, consider the VR-B algorithm. The cost of the instance recycling is $O(usmT)$ where u is the number of times an instance can be used in an update, s is the number of saved old instances, m is the maximum number of non-zero attributes in an instance, and T is the current trial number. The cost of voting is the cost of the basic algorithm plus $O(hmT)$ plus $O((rwm + wm)h(\log T)^2)$. Remember that h is the number of hypotheses used in the voting, r is the number of instance used to help estimate accuracy, and w is the maximum window size that is used to search for good hypotheses. This gives a total cost of $O(usmT + hmT + (rwm + wm)h(\log T)^2)$. As T gets large, this gives a bound of approximately $O(usmT + hmT)$. The space cost is $O(sm)$ to store the instances and $O(hn)$ to store the voting hypotheses.

Our main algorithm uses modification F of the voting procedure to allow a set of basic algorithms, $\{B_1, B_2, \ldots, B_v\}$. Instance recycling is added by using instance recycling on all the basic algorithms. These algorithms, $\{\text{R-}B_1, \text{R-}B_2, \ldots, \text{R-}B_v\}$, are used as the basic algorithms of the Combine voting algorithm. The new algorithm is called VR-Combine. It is the algorithm we recommend for on-line learning when instances are generated by a fixed distribution. The cost is similar to VR-B except that cost of instance recycling is $O(vusmT)$. Therefore the total cost is $O(vusmT + hmT + (rwm + wm)h(\log T)^2)$. As T gets large this gives a bound of approximately $O(vusmT + hmT)$. Typically, the instance recycling dominates this cost. The space used is $O(sm)$ to store the instances, $O(hn)$ to store the voting hypotheses, and $O(vn)$ to store the basic algorithms.

5.2 Voting and Recycling Experiments

In this section, we give experiments to verify the effectiveness of combining the voting and instance recycling techniques. We perform the experiments using the linear-threshold algorithms and experimental methodology described in Section 3.4. As we will see, the combined technique improves upon the individual techniques.

Our experiments are divided into four sections. The first section tests the default parameter values that are used for voting in Chapter 3 and instance recycling in Chapter 4. The second section varies the parameter settings for the voting component and the third section varies the parameter settings for the instance recycling. The last section gives the results of experiments based on the best parameter settings found in the previous two sections.

5.2.1 Default Parameters

All our experiments in this section are based on using the previous default parameter settings for the voting and recycling techniques. For voting, the default values are $h = 30$, $w = 100$, $f = 0.5$, and $r = 100$. For instance recycling, the default values are $s = 100$ and $u = 1$.

In Figure 5.1, we give a scatter plot that compares a recycled algorithm, R-B, to voting and recycling algorithm, VR-B. Each point of the scatter plot represents an algorithm/concept pair, and we plot every possible combination used in our experiments. This gives a total of 6138 points. The y coordinate of each points corresponds to the final number of mistakes made by the recycling version of an algorithm on the particular concept; the x coordinate corresponds to the final number of mistakes with voting applied to the recycled algorithm. As can be seen, for every algorithm R-B, the VR-B algorithm makes fewer mistakes.

In Figure 5.2, we give a similar scatter plot that compares the voting version of algorithm B to VR-B. Again, we plot all 6138 possible combinations. For basic algorithm B, the y coordinate corresponds to the final number of mistakes made by V-B and the x coordinate corresponds to VR-B. The results are not as clear cut as the previous figure. There are several cases where V-B performs better than VR-B. While it is difficult to tell based on the density of the plot, V-B makes fewer mistakes in 277 of the experiments. This is a relatively small number, and we will show that on average VR-B makes fewer mistakes.

Next, we focus on the performance of the individual algorithms. In Table 5.1 we give the results for all of our algorithms with the default parameters. The first column

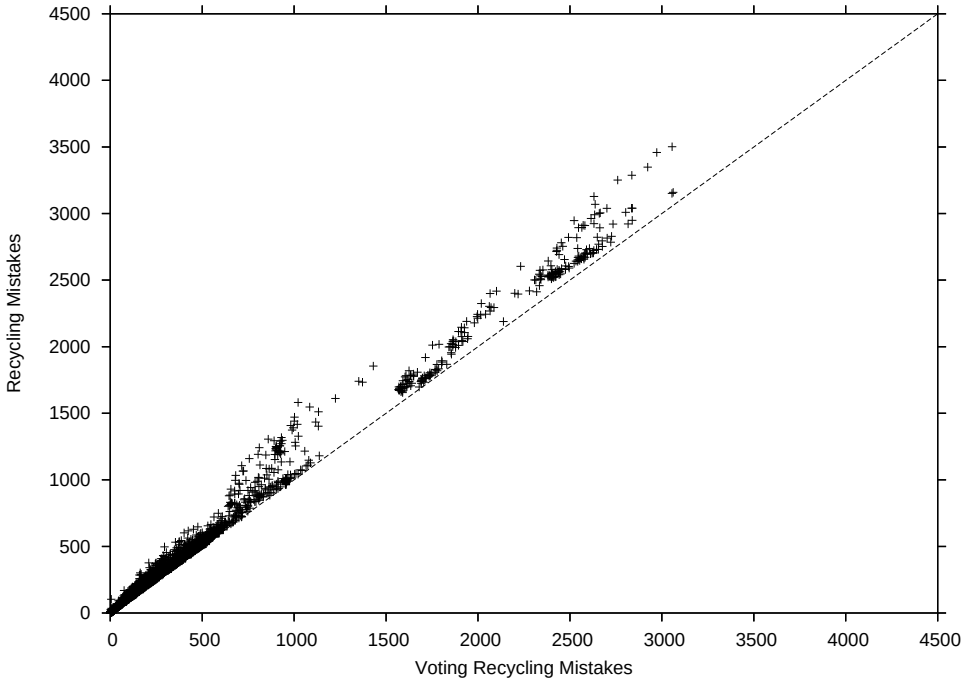

Figure 5.1: Scatter plot comparing recycled algorithm to algorithm with voting and recycling.

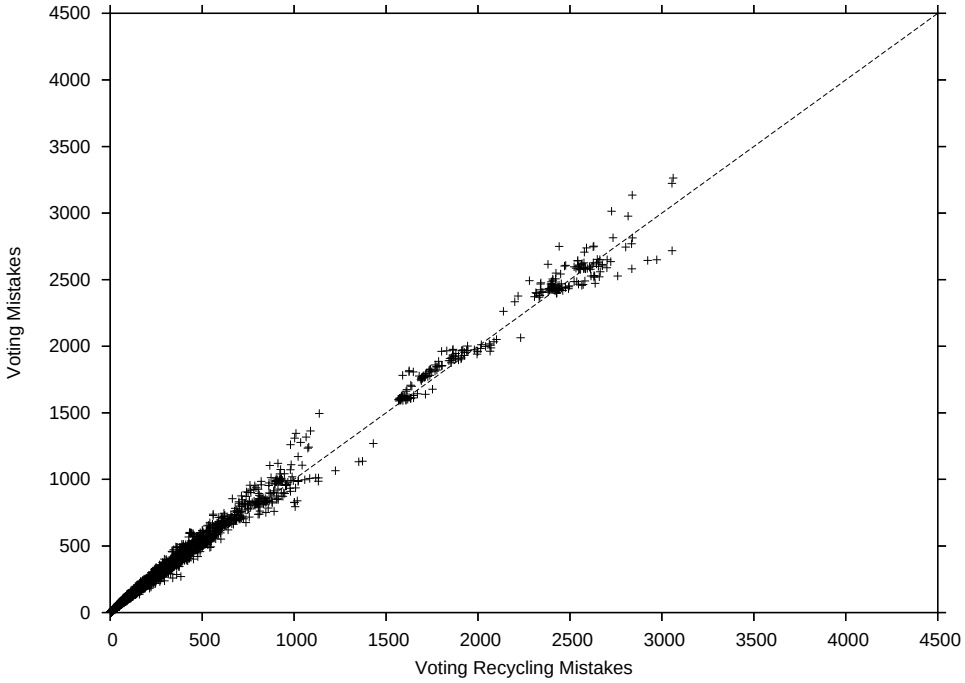

Figure 5.2: Scatter plot comparing recycled algorithm to algorithm with voting and recycling.

gives the total mistakes for the basic algorithm. The second column gives the results for voting with these basic algorithms. The next column gives the results with instance recycling, and the last column gives the full algorithm with both voting and instance recycling. Table 5.2 gives the mistake count for the same algorithms on the last 500 trials of all the concepts.

As can be seen in Table 5.1, combining voting and instance recycling continues to lower the number of mistakes. While the decrease is not additive, it is still significant. This is expected since there is less room for improvement with voting when the instance recycling is generating more accurate hypotheses. Still, for a small increase in cost, the VR algorithms give a significant decrease in mistakes.

The results in Table 5.2 show a somewhat more complicated but related picture. During the last 500 trials, for most cases, the voting algorithm makes less mistakes than the instance recycling version of the algorithm. However, over all the trials the recycling algorithm often makes less mistakes than the voting algorithm. This demonstrates that the recycling algorithm tends to make a greater contribution during the initial trials of the learning. This is not surprising since the voting procedure has difficulty during the initial stages of learning while instance recycling can exploit many of the early instances because of the initial inaccuracy of the prediction hypothesis.

Unfortunately, even though the VR algorithms make the smallest total number of mistakes, they do not always have the smallest number of mistakes on the final 500 trials. Eight of the VR algorithms make more mistakes on the last 500 trials. However, the VR algorithm is often close to the minimum; so close that the difference is often not statistically significant.

Overall the best performance comes from the VR-Combine algorithm. It makes more than 9% fewer mistakes than VR-ALMA(2). In fact, it makes the fewest mistakes on almost all the concepts. In the second column of Table 5.3, for each algorithm, we give the total number of concepts where that algorithm makes the fewest mistakes. The VR-Combine algorithm makes the fewest mistakes on all but one of the 186 concepts. This is the same effect we observed with the V-Combine algorithm in Section 3.4. The VR-Combine algorithm does so well because it selects hypotheses from several different

Name	$\hat{\mathrm{M}}$(Name)	$\hat{\mathrm{M}}$(V-Name)	$\hat{\mathrm{M}}$(R-Name)	$\hat{\mathrm{M}}$(VR-Name)
Per	73342 ± 80	51295 ± 74	50926 ± 76	**46179** ± 77
ALMA(2)	66066 ± 95	50046 ± 93	48914 ± 72	**45276** ± 67
ALMA($\ln n$)	71023 ± 105	53301 ± 82	53091 ± 89	**49341** ± 89
Bal(1.05)	72989 ± 83	51275 ± 96	50782 ± 77	**46153** ± 77
Bal(1.2)	72398 ± 83	50621 ± 84	50888 ± 76	**45987** ± 74
Bal(1.4)	74887 ± 86	50779 ± 79	53703 ± 91	**47338** ± 79
Bal(1.7)	81075 ± 109	52386 ± 87	58924 ± 86	**50672** ± 86
Bal(2.0)	86641 ± 89	54221 ± 79	62837 ± 85	**53580** ± 83
UWin(1.05)	93375 ± 93	65027 ± 103	63041 ± 79	**59204** ± 70
UWin(1.2)	88163 ± 93	59561 ± 87	58735 ± 68	**54182** ± 67
UWin(1.4)	88900 ± 87	57949 ± 74	60417 ± 73	**54334** ± 84
UWin(1.7)	94672 ± 103	58841 ± 73	64862 ± 72	**57082** ± 85
UWin(2.0)	100624 ± 89	60564 ± 81	69016 ± 83	**59707** ± 83
CUWin(1.05)	99476 ± 98	68613 ± 94	66768 ± 71	**62696** ± 77
CUWin(1.2)	85792 ± 91	59509 ± 84	57946 ± 70	**54068** ± 72
CUWin(1.4)	81676 ± 96	56068 ± 92	56905 ± 81	**51939** ± 79
CUWin(1.7)	81567 ± 95	55141 ± 73	58406 ± 71	**52371** ± 66
CUWin(2.0)	82557 ± 78	55290 ± 77	60109 ± 79	**53520** ± 79
NWin(1.05,.3)	110746 ± 76	87083 ± 110	85230 ± 72	**79418** ± 101
NWin(1.2,.3)	88760 ± 89	66453 ± 87	64192 ± 74	**59791** ± 63
NWin(1.4,.3)	85753 ± 96	60923 ± 78	61872 ± 73	**55926** ± 69
NWin(1.7,.3)	88529 ± 85	59332 ± 76	63935 ± 75	**56413** ± 83
NWin(2.0,.3)	92840 ± 97	59661 ± 92	66664 ± 73	**58163** ± 78
NWin(1.05,.5)	86721 ± 91	58946 ± 145	57966 ± 82	**53568** ± 87
NWin(1.2,.5)	80307 ± 102	55167 ± 94	55295 ± 70	**50493** ± 71
NWin(1.4,.5)	79405 ± 91	54021 ± 80	56125 ± 63	**50277** ± 62
NWin(1.7,.5)	81935 ± 91	54098 ± 71	59080 ± 71	**51778** ± 70
NWin(2.0,.5)	85610 ± 111	54927 ± 74	61877 ± 75	**53725** ± 76
NWin(1.05,.7)	86297 ± 97	65306 ± 96	62901 ± 81	**59755** ± 79
NWin(1.2,.7)	77574 ± 82	55917 ± 87	55290 ± 75	**51091** ± 70
NWin(1.4,.7)	76053 ± 93	53446 ± 77	54890 ± 73	**49587** ± 70
NWin(1.7,.7)	77541 ± 90	52705 ± 71	56953 ± 68	**50260** ± 71
NWin(2.0,.7)	80087 ± 92	53127 ± 72	59205 ± 83	**51647** ± 77
Combine		46432 ± 86		**41029** ± 68
Majority Label	124266 ± 50			

Table 5.1: Number of mistakes made by all the voting and recycling techniques with default parameter values.

Name	$\hat{\mathrm{M}}$(Name)	$\hat{\mathrm{M}}$(V-Name)	$\hat{\mathrm{M}}$(R-Name)	$\hat{\mathrm{M}}$(VR-Name)
Per	4734 ± 22	3303 ± 15	3408 ± 20	$\mathbf{3076} \pm 17$
ALMA(2)	4304 ± 18	3259 ± 20	3278 ± 17	$\mathbf{3035} \pm 18$
ALMA($\ln n$)	4590 ± 25	3468 ± 18	3539 ± 19	$\mathbf{3283} \pm 17$
Bal(1.05)	4708 ± 22	3309 ± 18	3396 ± 18	$\mathbf{3071} \pm 15$
Bal(1.2)	4700 ± 21	3285 ± 18	3439 ± 17	$\mathbf{3090} \pm 16$
Bal(1.4)	4979 ± 23	3323 ± 17	3719 ± 21	$\mathbf{3232} \pm 18$
Bal(1.7)	5522 ± 24	$\mathbf{3487} \pm 18$	4145 ± 19	3514 ± 18
Bal(2.0)	5960 ± 30	$\mathbf{3649} \pm 15$	4443 ± 19	3725 ± 17
UWin(1.05)	5739 ± 23	4054 ± 17	3991 ± 17	$\mathbf{3703} \pm 17$
UWin(1.2)	5494 ± 20	3716 ± 17	3849 ± 18	$\mathbf{3499} \pm 16$
UWin(1.4)	5745 ± 22	3699 ± 15	4093 ± 18	$\mathbf{3595} \pm 17$
UWin(1.7)	6317 ± 25	$\mathbf{3840} \pm 14$	4493 ± 18	3862 ± 15
UWin(2.0)	6830 ± 25	$\mathbf{4004} \pm 17$	4823 ± 20	4081 ± 17
CUWin(1.05)	5758 ± 23	4173 ± 16	3989 ± 18	$\mathbf{3814} \pm 14$
CUWin(1.2)	5215 ± 25	3696 ± 16	3717 ± 18	$\mathbf{3418} \pm 15$
CUWin(1.4)	5181 ± 23	3567 ± 17	3789 ± 19	$\mathbf{3408} \pm 17$
CUWin(1.7)	5359 ± 21	3592 ± 17	4010 ± 16	$\mathbf{3532} \pm 18$
CUWin(2.0)	5516 ± 21	3645 ± 17	4168 ± 20	$\mathbf{3640} \pm 19$
NWin(1.05,.3)	6022 ± 29	5162 ± 27	$\mathbf{4633} \pm 22$	4666 ± 27
NWin(1.2,.3)	5301 ± 21	3975 ± 18	3898 ± 17	$\mathbf{3601} \pm 17$
NWin(1.4,.3)	5392 ± 25	3754 ± 16	3995 ± 20	$\mathbf{3533} \pm 17$
NWin(1.7,.3)	5787 ± 21	3764 ± 16	4297 ± 17	$\mathbf{3697} \pm 16$
NWin(2.0,.3)	6224 ± 20	$\mathbf{3858} \pm 18$	4560 ± 18	3885 ± 17
NWin(1.05,.5)	5300 ± 20	3689 ± 17	3769 ± 19	$\mathbf{3440} \pm 16$
NWin(1.2,.5)	5077 ± 19	3523 ± 17	3684 ± 16	$\mathbf{3316} \pm 15$
NWin(1.4,.5)	5131 ± 22	3499 ± 19	3820 ± 20	$\mathbf{3355} \pm 16$
NWin(1.7,.5)	5460 ± 24	3565 ± 18	4101 ± 18	$\mathbf{3521} \pm 16$
NWin(2.0,.5)	5804 ± 24	$\mathbf{3658} \pm 16$	4344 ± 17	3690 ± 18
NWin(1.05,.7)	5327 ± 22	4103 ± 16	3989 ± 19	$\mathbf{3737} \pm 19$
NWin(1.2,.7)	4955 ± 21	3563 ± 19	3655 ± 17	$\mathbf{3318} \pm 16$
NWin(1.4,.7)	4966 ± 21	3460 ± 17	3740 ± 21	$\mathbf{3305} \pm 16$
NWin(1.7,.7)	5212 ± 23	3486 ± 16	3972 ± 18	$\mathbf{3426} \pm 15$
NWin(2.0,.7)	5451 ± 23	$\mathbf{3556} \pm 18$	4171 ± 18	3560 ± 18
Combine		3103 ± 18		$\mathbf{2861} \pm 18$
Majority Label	9404 ± 14			

Table 5.2: Number of mistakes on the last 500 trials made by all the voting and recycling techniques with default parameter values.

algorithms increasing the relative independence between its voting hypotheses.

5.2.2 Varying Parameters

Next, we explore our default choices for voting and instance recycling parameters. Unfortunately, it is too expensive to do an exhaustive test of the parameter choices, therefore we use the default parameters as a reference and perform experiments by varying one parameter from its default value. For voting, the default values are $h = 30$, $w = 100$, $f = 0.5$, and $r = 100$. For instance recycling, the default values are $s = 100$ and $u = 1$. In each table, the column header uses an abbreviated notation that includes the value of the changing parameter. At the end of the section, we run a final experiment using the optimal values from all the single parameter experiments.

Voting Parameters

Figure 5.3 contains a graph that shows the average number of mistakes as r, the number of instances used to estimate the accuracy of the hypotheses, is varied from 0 to 500. This average is taken over the 33 voting versions of the basic algorithms. The r parameter is used by voting modification C to select an accurate hypothesis from a window of candidates. In Table 5.4, we give the same results for each individual algorithm. As can be seen in the graph and table, all the algorithms show a monotonic decrease in the number of mistakes as we increase the number of instances. This suggests we use as many instances as possible given the complexity constraints on the algorithm. See Section 3.3.4 for more information on the complexity of this parameter.

The next parameter we vary is f, the percentage of the trial window that the voting algorithm is allowed to search for the most accurate hypothesis. These windows are chosen so that there is no overlap between successive windows; therefore when $f = 1$, as soon as the algorithm replaces one hypothesis and exits one window, it begins searching the next window for the next replacement. We show the average results over all algorithms in Figure 5.4, and the result for the individual algorithms in Table 5.5. The results in both the graph and table show a monotonic decrease in the number of mistakes as the window size grows. While many of these gains are modest and

Name	# Name best concepts	# VR-Name best concepts
Perceptron	0	0
ALMA(2)	56	0
ALMA($\ln n$)	17	1
Bal(1.05)	0	0
Bal(1.2)	1	0
Bal(1.4)	2	0
Bal(1.7)	0	0
Bal(2.0)	0	0
UWin(1.05)	2	0
UWin(1.2)	0	0
UWin(1.4)	0	0
UWin(1.7)	0	0
UWin(2.0)	0	0
UCWin(1.05)	0	0
UCWin(1.2)	0	0
UCWin(1.4)	0	0
UCWin(1.7)	0	0
UCWin(2.0)	0	0
NWin(1.05,.3)	2	0
NWin(1.2,.3)	0	0
NWin(1.4,.3)	0	0
NWin(1.7,.3)	1	0
NWin(2.0,.3)	0	0
NWin(1.05,.5)	0	0
NWin(1.2,.5)	0	0
NWin(1.4,.5)	1	0
NWin(1.7,.5)	0	0
NWin(2.0,.5)	1	0
NWin(1.05,.7)	39	0
NWin(1.2,.7)	21	0
NWin(1.4,.7)	6	0
NWin(1.7,.7)	8	0
NWin(2.0,.7)	29	0
Combine		185

Table 5.3: Number of concepts where the algorithm gives the minimum number of mistakes.

Algorithm	$\hat{\mathrm{M}}$(VR(0))	$\hat{\mathrm{M}}$(VR(10))	$\hat{\mathrm{M}}$(VR(100))	$\hat{\mathrm{M}}$(VR(200))	$\hat{\mathrm{M}}$(VR(500))
Per	46441 ± 76	46395 ± 75	46179 ± 77	46145 ± 77	**46108** ± 79
ALMA(2)	45362 ± 70	45340 ± 68	45276 ± 67	45242 ± 68	**45221** ± 70
ALMA($\ln n$)	49442 ± 87	49435 ± 88	49341 ± 89	49303 ± 88	**49279** ± 90
Bal(1.05)	46398 ± 80	46358 ± 75	46153 ± 77	46097 ± 77	**46079** ± 77
Bal(1.2)	46231 ± 74	46202 ± 77	45987 ± 74	45929 ± 75	**45881** ± 74
Bal(1.4)	47736 ± 85	47671 ± 83	47338 ± 79	47238 ± 80	**47174** ± 78
Bal(1.7)	51361 ± 87	51265 ± 87	50672 ± 86	50529 ± 80	**50471** ± 86
Bal(2.0)	54598 ± 89	54423 ± 91	53580 ± 83	53411 ± 82	**53327** ± 87
UWin(1.05)	59418 ± 72	59388 ± 72	59204 ± 70	**59196** ± 68	**59169** ± 66
UWin(1.2)	54434 ± 65	54407 ± 67	54182 ± 67	54099 ± 67	**54047** ± 63
UWin(1.4)	54889 ± 83	54814 ± 87	54334 ± 84	54198 ± 82	**54098** ± 80
UWin(1.7)	57995 ± 83	57852 ± 83	57082 ± 85	56891 ± 84	**56800** ± 87
UWin(2.0)	60807 ± 82	60624 ± 88	59707 ± 83	59528 ± 84	**59438** ± 80
UCWin(1.05)	62969 ± 77	62899 ± 77	62696 ± 77	62658 ± 81	**62626** ± 77
UCWin(1.2)	54263 ± 75	54248 ± 74	54068 ± 72	54008 ± 72	**53977** ± 75
UCWin(1.4)	52276 ± 77	52209 ± 81	51939 ± 79	51858 ± 75	**51801** ± 76
UCWin(1.7)	52881 ± 70	52796 ± 70	52371 ± 66	52269 ± 69	**52196** ± 75
UCWin(2.0)	54164 ± 73	54085 ± 71	53520 ± 79	53343 ± 75	**53286** ± 73
NWin(1.05,.3)	79717 ± 101	79606 ± 100	79418 ± 101	79307 ± 96	**79161** ± 98
NWin(1.2,.3)	60013 ± 71	59985 ± 68	59791 ± 63	59736 ± 64	**59700** ± 66
NWin(1.4,.3)	56342 ± 72	56297 ± 78	55926 ± 69	55854 ± 67	**55805** ± 70
NWin(1.7,.3)	57135 ± 82	57057 ± 82	56413 ± 83	56263 ± 77	**56202** ± 77
NWin(2.0,.3)	59150 ± 84	59008 ± 83	58163 ± 78	57988 ± 81	**57916** ± 77
NWin(1.05,.5)	53801 ± 88	53762 ± 89	53568 ± 87	53522 ± 86	**53507** ± 87
NWin(1.2,.5)	50773 ± 68	50708 ± 69	50493 ± 71	50443 ± 73	**50404** ± 70
NWin(1.4,.5)	50680 ± 68	50630 ± 68	50277 ± 62	50197 ± 66	**50146** ± 61
NWin(1.7,.5)	52436 ± 77	52342 ± 80	51778 ± 70	51624 ± 76	**51563** ± 75
NWin(2.0,.5)	54669 ± 85	54526 ± 78	53725 ± 76	53560 ± 72	**53489** ± 72
NWin(1.05,.7)	59849 ± 86	59808 ± 82	59755 ± 79	59719 ± 84	**59702** ± 84
NWin(1.2,.7)	51282 ± 66	51246 ± 67	51091 ± 70	51045 ± 67	**51020** ± 65
NWin(1.4,.7)	49914 ± 73	49863 ± 73	49587 ± 70	49516 ± 71	**49467** ± 72
NWin(1.7,.7)	50808 ± 71	50720 ± 73	50260 ± 71	50148 ± 69	**50078** ± 73
NWin(2.0,.7)	52393 ± 79	52281 ± 78	51647 ± 77	51512 ± 74	**51437** ± 76
Combine	41147 ± 69	41129 ± 66	41029 ± 68	41005 ± 66	**40976** ± 67

Table 5.4: Number of mistakes when using $r = \{0, 10, 100, 200, 500\}$ instances for hypotheses accuracy estimation. The remaining parameters are set at the default values.

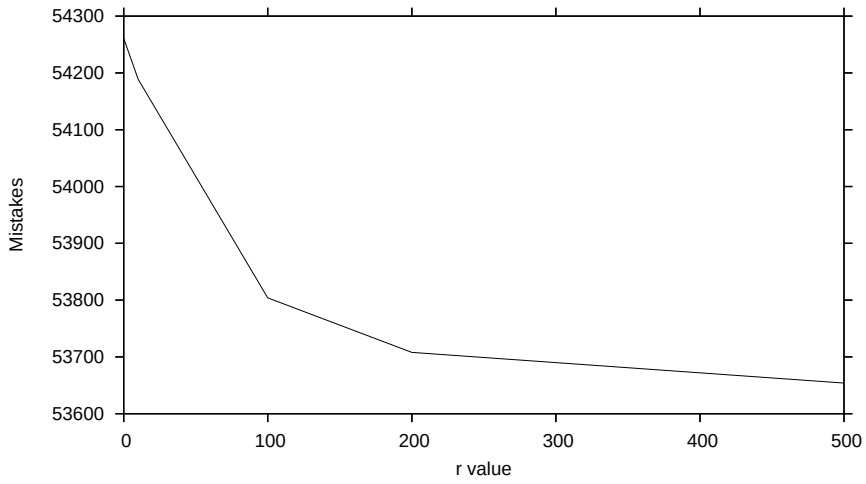

Figure 5.3: Average number of total mistakes of all versions of the VR algorithm when using $r = \{0, 10, 100, 200, 500\}$. All other parameters set to default value.

not statistically significant, the consistency of the gains suggests that choosing a large window is the best strategy.

Originally, we were concerned that allowing f to get too large would cause problems since the voting procedure could select a hypothesis from the end of one window and from the start of the next window. These hypotheses might be strongly correlated limiting the diversity of the voting hypotheses. Over these data sets, this does not appear to be a problem.

A parameter related to f is w, the maximum number of trials to search in a window. Searching the entire window for an accurate hypothesis can become expensive. This is, in part, why we have a limit to the number of trials that the algorithm is allowed to search. The default value for w is set to 100. However, we do not perform any experiments varying w because with only 10,000 trials and default values $h = 30$ and $f = 0.5$, the 100 trial limit is rarely reached. While we could test values of w smaller than 100, the experiments with f already gives a good idea of how voting is affected by window size.

Last, we consider the total number of mistakes as the number of hypotheses is varied from 10 to 50. The results for the average number of mistakes for all algorithms is given in Figure 5.5; the results for the individual algorithms is given in Table 5.6.

Name	$\hat{\mathrm{M}}$(VR(.1))	$\hat{\mathrm{M}}$(VR(.25))	$\hat{\mathrm{M}}$(VR(.5))	$\hat{\mathrm{M}}$(VR(.75))	$\hat{\mathrm{M}}$(VR(.9))
Per	46395 ± 79	46285 ± 78	46179 ± 77	46097 ± 76	**46072** ± 76
ALMA(2)	45358 ± 68	45317 ± 68	45276 ± 67	45214 ± 67	**45206** ± 66
ALMA($\ln n$)	49436 ± 90	49400 ± 90	49341 ± 89	49306 ± 89	**49288** ± 89
Bal(1.05)	46353 ± 79	46271 ± 76	46153 ± 77	46065 ± 79	**46034** ± 76
Bal(1.2)	46183 ± 80	46078 ± 74	45987 ± 74	45920 ± 74	**45895** ± 72
Bal(1.4)	47644 ± 83	47485 ± 77	47338 ± 79	47242 ± 79	**47205** ± 77
Bal(1.7)	51241 ± 88	50932 ± 86	50672 ± 86	50520 ± 86	**50474** ± 82
Bal(2.0)	54354 ± 85	53951 ± 83	53580 ± 83	53380 ± 84	**53295** ± 86
UWin(1.05)	59393 ± 73	59311 ± 71	59204 ± 70	59162 ± 70	**59139** ± 72
UWin(1.2)	54380 ± 67	54265 ± 65	54182 ± 67	54086 ± 68	**54060** ± 67
UWin(1.4)	54751 ± 85	54539 ± 80	54334 ± 84	54202 ± 81	**54156** ± 81
UWin(1.7)	57766 ± 82	57407 ± 83	57082 ± 85	56878 ± 82	**56799** ± 83
UWin(2.0)	60555 ± 80	60101 ± 79	59707 ± 83	59486 ± 79	**59407** ± 77
CUWin(1.05)	62897 ± 76	62816 ± 77	62696 ± 77	62611 ± 80	**62582** ± 78
CUWin(1.2)	54231 ± 73	54154 ± 74	54068 ± 72	54006 ± 70	**53984** ± 73
CUWin(1.4)	52190 ± 79	52049 ± 79	51939 ± 79	51856 ± 78	**51834** ± 75
CUWin(1.7)	52757 ± 78	52540 ± 71	52371 ± 66	52263 ± 65	**52234** ± 66
CUWin(2.0)	53989 ± 74	53718 ± 81	53520 ± 79	53377 ± 75	**53331** ± 77
NWin(1.05,.3)	79703 ± 103	79568 ± 98	79418 ± 101	79280 ± 97	**79221** ± 97
NWin(1.2,.3)	59956 ± 66	59880 ± 70	59791 ± 63	59755 ± 64	**59719** ± 66
NWin(1.4,.3)	56226 ± 77	56062 ± 74	55926 ± 69	55872 ± 71	**55843** ± 69
NWin(1.7,.3)	56950 ± 89	56642 ± 86	56413 ± 83	56273 ± 78	**56230** ± 76
NWin(2.0,.3)	58872 ± 81	58485 ± 78	58163 ± 78	57996 ± 80	**57939** ± 80
NWin(1.05,.5)	53762 ± 88	53674 ± 88	53568 ± 87	53492 ± 87	**53456** ± 88
NWin(1.2,.5)	50697 ± 73	50594 ± 72	50493 ± 71	50426 ± 73	**50403** ± 70
NWin(1.4,.5)	50570 ± 69	50411 ± 66	50277 ± 62	50200 ± 65	**50170** ± 62
NWin(1.7,.5)	52237 ± 74	51962 ± 70	51778 ± 70	51648 ± 72	**51624** ± 75
NWin(2.0,.5)	54394 ± 78	54011 ± 79	53725 ± 76	53572 ± 71	**53516** ± 74
NWin(1.05,.7)	59843 ± 81	59798 ± 83	59755 ± 79	59718 ± 82	**59705** ± 81
NWin(1.2,.7)	51232 ± 68	51153 ± 70	51091 ± 70	51037 ± 69	**51022** ± 66
NWin(1.4,.7)	49813 ± 70	49693 ± 72	49587 ± 70	49518 ± 68	**49504** ± 69
NWin(1.7,.7)	50629 ± 72	50430 ± 73	50260 ± 71	50165 ± 68	**50146** ± 67
NWin(2.0,.7)	52160 ± 77	51865 ± 78	51647 ± 77	51532 ± 74	**51488** ± 75
Combine	41129 ± 68	41092 ± 70	41029 ± 68	40972 ± 67	**40961** ± 65

Table 5.5: Number of mistakes when using $f = \{0.1, 0.25, 0.5, 0.75, 0.9\}$ of the hypothesis window for the highest accuracy hypothesis. The remaining parameters are set at the default value.

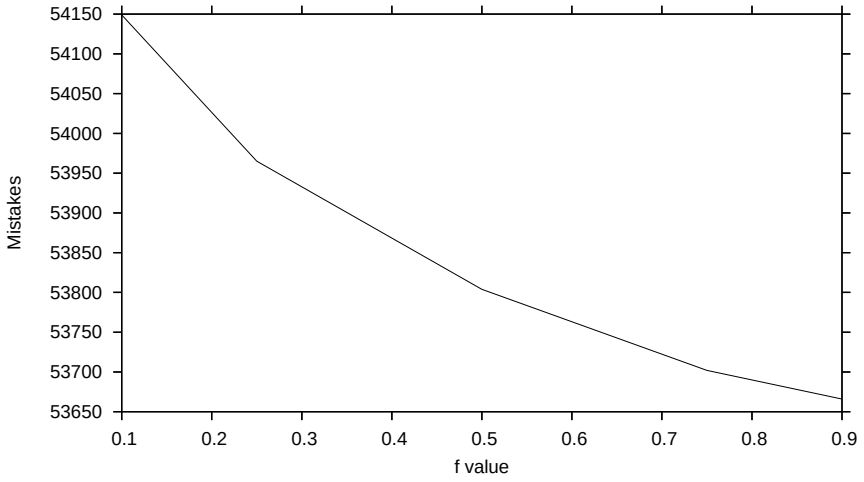

Figure 5.4: Average number of total mistakes of all versions of the VR algorithm when using $f = \{0.1, 0.25, 0.5, 0.75, 0.9\}$. All other parameters set to default value.

These results are somewhat surprising. The number of mistakes consistently increases for all algorithms when we increase the number of voting hypotheses beyond 30. In fact, for most algorithms $h = 20$ gives the fewest mistakes. While the difference in the number of mistakes is small, it is consistent.

More extensive testing on individual concepts suggests that the number of mistakes made by the voting algorithm is at a minimum for smaller h only after adding voting modification C. This seems plausible since voting modification C allows a window of hypothesis to be searched at a given trial. With a larger h value more hypotheses will be added but each will have a smaller window to select an accurate hypothesis. This may cause the algorithm to add both the accurate hypothesis and several lesser hypotheses. The net effect of these extra lower accuracy hypotheses is an increase in the number of mistakes. While this effect is not present on all concepts, it appears enough to make a smaller number of hypotheses better on average. These results suggest that $h = 20$ is a good parameter choice to maximize performance with the added benefit of keeping costs small.

Looking over all the parameter experiments for voting, while it is possible to lower the number of mistakes by choosing different values than the defaults, it does not have a large effect. In particular, for the VR-Combine algorithm $r = 500$, $f = 0.9$, and $h = 20$

Name	$\hat{\mathrm{M}}$(VR(10))	$\hat{\mathrm{M}}$(VR(20))	$\hat{\mathrm{M}}$(VR(30))	$\hat{\mathrm{M}}$(VR(40))	$\hat{\mathrm{M}}$(VR(50))
Per	46281 ± 82	46186 ± 79	**46179** ± 77	46228 ± 75	46245 ± 76
ALMA(2)	45334 ± 67	45274 ± 71	**45276** ± 67	45280 ± 68	45290 ± 69
ALMA($\ln n$)	49463 ± 84	**49343** ± 87	49341 ± 89	49352 ± 87	49369 ± 90
Bal(1.05)	46208 ± 74	**46132** ± 74	46153 ± 77	46192 ± 75	46223 ± 78
Bal(1.2)	46107 ± 75	**45984** ± 69	45987 ± 74	46015 ± 75	46037 ± 74
Bal(1.4)	47494 ± 76	**47330** ± 80	47338 ± 79	47375 ± 77	47402 ± 77
Bal(1.7)	50843 ± 88	**50654** ± 82	50672 ± 86	50758 ± 84	50812 ± 81
Bal(2.0)	53667 ± 93	**53548** ± 87	53580 ± 83	53701 ± 87	53776 ± 83
UWin(1.05)	59307 ± 80	**59200** ± 70	59204 ± 70	59237 ± 66	59265 ± 68
UWin(1.2)	54242 ± 66	**54164** ± 65	54182 ± 67	54186 ± 65	54200 ± 67
UWin(1.4)	54418 ± 74	**54303** ± 80	54334 ± 84	54374 ± 81	54427 ± 81
UWin(1.7)	57123 ± 88	**57016** ± 86	57082 ± 85	57167 ± 80	57247 ± 80
UWin(2.0)	59752 ± 85	**59648** ± 80	59707 ± 83	59837 ± 82	59927 ± 78
CUWin(1.05)	**62540** ± 75	62618 ± 81	62696 ± 77	62761 ± 79	62792 ± 75
CUWin(1.2)	54097 ± 77	**54049** ± 76	54068 ± 72	54082 ± 75	54098 ± 74
CUWin(1.4)	52036 ± 72	**51914** ± 74	51939 ± 79	51952 ± 75	51976 ± 77
CUWin(1.7)	52466 ± 75	**52355** ± 70	52371 ± 66	52431 ± 71	52459 ± 69
CUWin(2.0)	53594 ± 87	**53464** ± 80	53520 ± 79	53573 ± 72	53617 ± 74
NWin(1.05,.3)	**79214** ± 99	79356 ± 96	79418 ± 101	79547 ± 98	79547 ± 99
NWin(1.2,.3)	59876 ± 64	59800 ± 63	**59791** ± 63	59814 ± 64	59823 ± 66
NWin(1.4,.3)	56135 ± 67	55955 ± 66	**55926** ± 69	55956 ± 73	55981 ± 71
NWin(1.7,.3)	56566 ± 82	**56392** ± 79	56413 ± 83	56468 ± 85	56529 ± 76
NWin(2.0,.3)	58289 ± 80	**58094** ± 78	58163 ± 78	58269 ± 79	58326 ± 76
NWin(1.05,.5)	53638 ± 88	**53558** ± 88	53568 ± 87	53595 ± 89	53636 ± 91
NWin(1.2,.5)	50628 ± 79	50501 ± 72	**50493** ± 71	50521 ± 74	50547 ± 73
NWin(1.4,.5)	50443 ± 64	50299 ± 60	**50277** ± 62	50301 ± 61	50329 ± 63
NWin(1.7,.5)	51989 ± 80	**51763** ± 68	51778 ± 70	51809 ± 68	51851 ± 70
NWin(2.0,.5)	53888 ± 79	**53689** ± 76	53725 ± 76	53790 ± 76	53856 ± 78
NWin(1.05,.7)	59881 ± 82	59783 ± 79	**59755** ± 79	59761 ± 83	59763 ± 80
NWin(1.2,.7)	51214 ± 70	51102 ± 69	**51091** ± 70	51101 ± 67	51108 ± 64
NWin(1.4,.7)	49780 ± 71	49610 ± 67	**49587** ± 70	49618 ± 70	49607 ± 68
NWin(1.7,.7)	50443 ± 65	**50260** ± 69	**50260** ± 71	50294 ± 69	50312 ± 69
NWin(2.0,.7)	51847 ± 72	51657 ± 72	**51647** ± 77	51715 ± 76	51745 ± 75
Combine	41101 ± 61	**41009** ± 65	41029 ± 68	41048 ± 66	41057 ± 67

Table 5.6: Number of mistakes made with $h = \{10, 20, 30, 40, 50\}$ voting hypotheses. The remaining parameters are set at the default values.

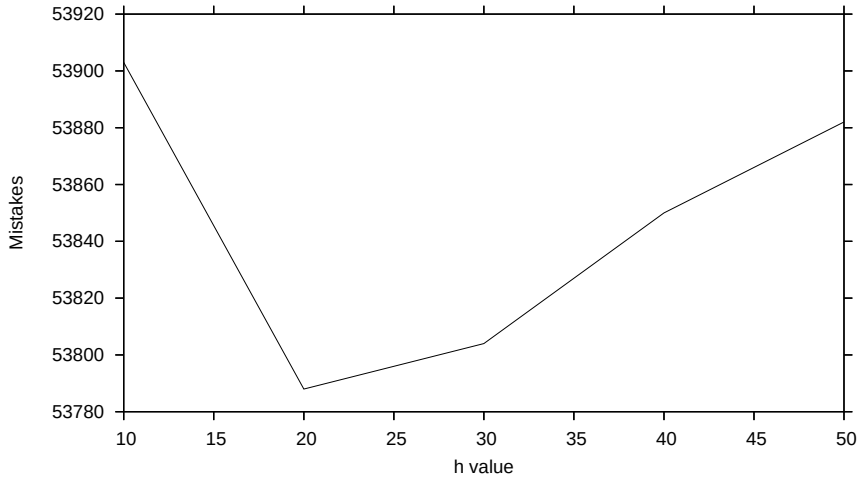

Figure 5.5: Average number of total mistakes of all versions of the VR algorithm when using $h = \{10, 20, 30, 40, 50\}$. All other parameters set to default value.

give the fewest mistakes, however, the differences are not statistically significant over the default values. Later we run experiments that use all the best parameter values for VR-Combine to see if in aggregate these parameter changes have a larger influence.

Instance Recycling Parameters

Next we look at the parameters for the instance recycling algorithm. In this case, we give results for recycling on the basic version of the algorithm and the basic version with voting. While the voting results are the most important, we include the results for the basic algorithms since they are affected by the parameter choices, and their behavior gives us insight into the VR algorithms. Again, we keep all the parameters at their default value and only modify a single instance recycling parameter.

The graph labeled basic in Figure 5.6 gives the average number of mistakes for recycling on the basic algorithms as we vary the number of recycling instances using $s = \{10, 50, 100, 200, 500\}$. The results for the individual algorithms are given in Table 5.7. As can be seen, the number of mistakes monotonically decreases as we increase the value of s. If fact, there is a dramatic statistically significant decrease in the number of mistakes for all of the algorithms. In some cases, the number of mistakes using $s = 500$ approaches the the performance of the voting and recycling algorithm with the default

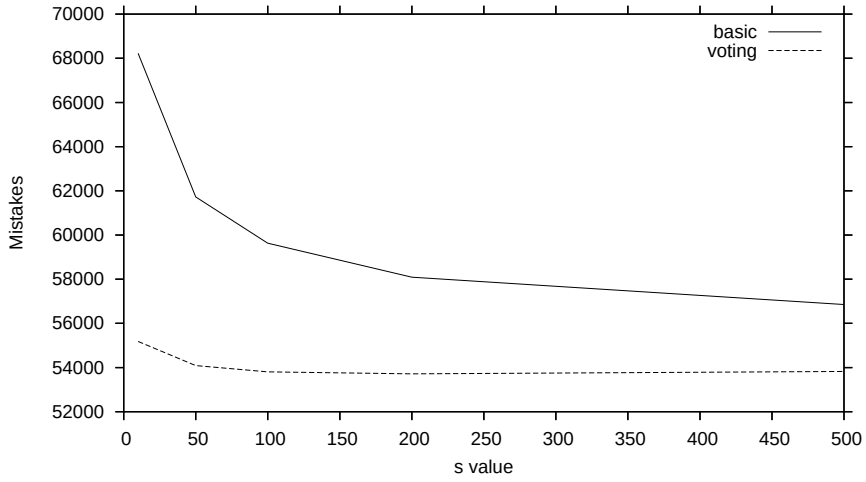

Figure 5.6: Average number of total mistakes for all basic and voting algorithms when using $s = \{10, 50, 100, 200, 500\}$. All other parameters set to default value.

parameters. Still instance recycling gets expensive as s grows, and we still expect a large benefit using the VR-Combine algorithm to effectively use different algorithms for different problems.

The graph labeled voting in Figure 5.6 gives the average result of using using $s = \{10, 50, 100, 200, 500\}$ on all the VR voting algorithms. Table 5.8 gives the results on the individual algorithms. Here the results are not so clear cut. While the average graph is almost monotonically decreasing, the individual algorithms show a wider range of behavior. The number of mistakes does monotonically decrease for the ALMA algorithms and Perceptron, but the Winnow algorithms often make more mistakes as the number of recycled instances increases. In particular, the larger the Winnow multiplier the more mistakes with a large s value.

This could be caused by a lack of diversity in the hypotheses generated by the instance recycling. The instances that are used for recycling are not from the same distribution that generated the instances. With the default $u = 1$ parameter, each instance can only be used for a single update, therefore the recycled instances only contain instances that are correctly predicted by the algorithm. While this will most likely lower the amount of noise in the recycled instances, this change in distribution can also effect the diversity in the generated hypotheses.

Name	$\hat{M}$(R(10))	$\hat{M}$(R(50))	$\hat{M}$(R(100))	$\hat{M}$(R(200))	$\hat{M}$(R(500))
Per	59635 ± 88	53052 ± 80	50926 ± 76	49288 ± 77	**47846** ± 68
ALMA(2)	56616 ± 94	50832 ± 82	48914 ± 72	47407 ± 83	**46003** ± 77
ALMA($\ln n$)	61407 ± 91	55191 ± 90	53091 ± 89	51501 ± 87	**50055** ± 90
Bal(1.05)	59501 ± 79	52880 ± 74	50782 ± 77	49090 ± 62	**47644** ± 74
Bal(1.2)	59565 ± 98	53034 ± 81	50888 ± 76	49260 ± 75	**47803** ± 77
Bal(1.4)	62338 ± 94	55818 ± 80	53703 ± 91	52108 ± 90	**50783** ± 88
Bal(1.7)	67826 ± 107	61047 ± 98	58924 ± 86	57415 ± 92	**56245** ± 92
Bal(2.0)	72356 ± 103	65033 ± 93	62837 ± 85	61319 ± 82	**60309** ± 94
UWin(1.05)	69931 ± 86	64575 ± 75	63041 ± 79	61836 ± 74	**60831** ± 70
UWin(1.2)	67134 ± 85	60685 ± 65	58735 ± 68	57306 ± 67	**56227** ± 64
UWin(1.4)	69471 ± 90	62538 ± 71	60417 ± 73	59015 ± 68	**58080** ± 71
UWin(1.7)	75041 ± 94	67190 ± 81	64862 ± 72	63345 ± 77	**62365** ± 83
UWin(2.0)	80166 ± 100	71549 ± 87	69016 ± 83	67399 ± 86	**66314** ± 87
UCWin(1.05)	74855 ± 86	68486 ± 73	66768 ± 71	65538 ± 75	**64510** ± 75
UCWin(1.2)	65856 ± 78	59723 ± 72	57946 ± 70	56686 ± 74	**55613** ± 72
UCWin(1.4)	64716 ± 84	58756 ± 67	56905 ± 81	55521 ± 73	**54376** ± 75
UCWin(1.7)	66324 ± 87	60396 ± 81	58406 ± 71	56935 ± 76	**55736** ± 75
UCWin(2.0)	68191 ± 88	62132 ± 76	60109 ± 79	58617 ± 73	**57358** ± 74
NWin(1.05,.3)	93265 ± 81	87341 ± 75	85230 ± 72	83490 ± 69	**81909** ± 69
NWin(1.2,.3)	73344 ± 78	66505 ± 76	64192 ± 74	62387 ± 66	**60916** ± 63
NWin(1.4,.3)	71195 ± 89	64230 ± 75	61872 ± 73	60116 ± 69	**58711** ± 77
NWin(1.7,.3)	73964 ± 90	66419 ± 82	63935 ± 75	62188 ± 79	**60951** ± 75
NWin(2.0,.3)	77509 ± 86	69282 ± 72	66664 ± 73	64904 ± 82	**63761** ± 76
NWin(1.05,.5)	66295 ± 92	59977 ± 88	57966 ± 82	56501 ± 79	**55316** ± 80
NWin(1.2,.5)	63775 ± 80	57348 ± 77	55295 ± 70	53725 ± 73	**52486** ± 70
NWin(1.4,.5)	64665 ± 75	58255 ± 77	56125 ± 63	54532 ± 71	**53219** ± 65
NWin(1.7,.5)	67948 ± 85	61318 ± 79	59080 ± 71	57463 ± 72	**56224** ± 67
NWin(2.0,.5)	71338 ± 93	64203 ± 83	61877 ± 75	60262 ± 86	**59124** ± 79
NWin(1.05,.7)	69112 ± 87	64501 ± 77	62901 ± 81	61646 ± 80	**60457** ± 84
NWin(1.2,.7)	62913 ± 82	57231 ± 64	55290 ± 75	53822 ± 72	**52570** ± 68
NWin(1.4,.7)	62695 ± 88	56905 ± 68	54890 ± 73	53364 ± 74	**52051** ± 71
NWin(1.7,.7)	64920 ± 82	59035 ± 71	56953 ± 68	55374 ± 67	**54095** ± 70
NWin(2.0,.7)	67498 ± 93	61350 ± 78	59205 ± 83	57609 ± 81	**56350** ± 74

Table 5.7: Number of mistakes made by instance recycling on basic algorithms with $s = \{10, 50, 100, 200, 500\}$ saved instances. The remaining parameters are set at the default values.

Name	$\hat{M}$(VR(10))	$\hat{M}$(VR(50))	$\hat{M}$(VR(100))	$\hat{M}$(VR(200))	$\hat{M}$(VR(500))
Per	48405 ± 75	46747 ± 76	46179 ± 77	45813 ± 70	**45484** ± 66
ALMA(2)	47810 ± 78	45971 ± 75	45276 ± 67	44651 ± 79	**44060** ± 76
ALMA($\ln n$)	51564 ± 83	49981 ± 84	49341 ± 89	48781 ± 83	**48279** ± 87
Bal(1.05)	48418 ± 82	46706 ± 69	46153 ± 77	45706 ± 71	**45394** ± 66
Bal(1.2)	48070 ± 77	46549 ± 69	45987 ± 74	45645 ± 70	**45387** ± 74
Bal(1.4)	48650 ± 70	47606 ± 67	47338 ± 79	**47265** ± 84	47405 ± 83
Bal(1.7)	50917 ± 81	**50582** ± 79	50672 ± 86	51066 ± 86	51684 ± 84
Bal(2.0)	**53168** ± 84	53268 ± 97	53580 ± 83	54256 ± 90	55265 ± 96
UWin(1.05)	60745 ± 84	59517 ± 82	59204 ± 70	58973 ± 68	**58862** ± 73
UWin(1.2)	55689 ± 71	54477 ± 63	54182 ± 67	**54010** ± 68	54083 ± 60
UWin(1.4)	55117 ± 78	54362 ± 75	**54334** ± 84	54545 ± 72	55031 ± 73
UWin(1.7)	56888 ± 75	**56758** ± 74	57082 ± 85	57596 ± 79	58431 ± 92
UWin(2.0)	**59112** ± 75	59299 ± 88	59707 ± 83	60411 ± 88	61416 ± 88
UCWin(1.05)	64758 ± 86	63168 ± 80	62696 ± 77	62358 ± 87	**62111** ± 76
UCWin(1.2)	56047 ± 77	54531 ± 72	54068 ± 72	53734 ± 75	**53564** ± 71
UCWin(1.4)	53371 ± 83	52241 ± 65	51939 ± 79	**51779** ± 71	51783 ± 75
UCWin(1.7)	53131 ± 79	52492 ± 77	**52371** ± 66	52413 ± 71	52635 ± 75
UCWin(2.0)	53776 ± 75	**53480** ± 73	53520 ± 79	53710 ± 71	54088 ± 69
NWin(1.05,.3)	83303 ± 107	80498 ± 106	79418 ± 101	78514 ± 103	**77722** ± 90
NWin(1.2,.3)	63149 ± 80	60715 ± 79	59791 ± 63	59096 ± 67	**58602** ± 63
NWin(1.4,.3)	58312 ± 80	56542 ± 67	55926 ± 69	55632 ± 73	**55582** ± 79
NWin(1.7,.3)	57450 ± 83	56552 ± 84	**56413** ± 83	56584 ± 78	57089 ± 95
NWin(2.0,.3)	58331 ± 90	**58009** ± 77	58163 ± 78	58650 ± 73	59486 ± 89
NWin(1.05,.5)	55332 ± 87	53957 ± 96	53568 ± 87	53290 ± 87	**53248** ± 89
NWin(1.2,.5)	52267 ± 77	50897 ± 75	50493 ± 71	50231 ± 70	**50216** ± 75
NWin(1.4,.5)	51771 ± 71	50610 ± 71	50277 ± 62	**50186** ± 67	50237 ± 73
NWin(1.7,.5)	52435 ± 68	51837 ± 70	**51778** ± 70	51955 ± 79	52398 ± 82
NWin(2.0,.5)	53766 ± 78	**53553** ± 82	53725 ± 76	54155 ± 92	54937 ± 95
NWin(1.05,.7)	61963 ± 101	60400 ± 94	59755 ± 79	59273 ± 91	**58792** ± 88
NWin(1.2,.7)	53139 ± 67	51619 ± 69	51091 ± 70	50712 ± 70	**50492** ± 70
NWin(1.4,.7)	51129 ± 72	49948 ± 67	49587 ± 70	49374 ± 73	**49313** ± 68
NWin(1.7,.7)	51091 ± 80	50421 ± 67	**50260** ± 71	50311 ± 67	50593 ± 79
NWin(2.0,.7)	51908 ± 82	**51596** ± 79	51647 ± 77	51941 ± 78	52397 ± 82
Combine	43558 ± 85	41670 ± 73	41029 ± 68	40465 ± 69	**39955** ± 67

Table 5.8: Number of mistakes made by instance recycling on the voting algorithms with $s = \{10, 50, 100, 200, 500\}$ saved instances. The remaining parameters are set at the default values.

In the majority of our experiments, the label of the data is skewed towards -1. However, a mistake-driven algorithm tends to make an equal number of mistakes on label -1 and label 1. Therefore, a higher proportion of instances with label 1 will cause a mistake, and the instance recycling will contain more label -1 instances. When the recycling takes place, the algorithms update on these label -1 instances and create a bias towards this label. This becomes more of a problem as we increase the number of recycling instances. The Winnow algorithms with large multipliers are more susceptible to this problem since they make the biggest changes to their current hypothesis.

Still, even with this problem, our primary algorithm, VR-Combine, improves as we increase s. This is most likely because it gains some additional diversity since it selects hypotheses from different algorithms. It is currently the best algorithm with a decrease of over a 1000 mistakes from VR-Combine with default parameters.

Last, we experiment with the number of times an instance can be used in an update. Our experiments test $u = \{1, 2, 3, 4, 5\}$. The average total number of mistakes for all the basic algorithms is given in Figure 5.7 and is labeled basic. Table 5.9 gives the same results for the individual basic algorithms. Remember from Section 4.4 that larger u values can increase the number of times an algorithm updates on noisy instances. This can cause the algorithm to make more mistakes. In addition, based on Appendix A, B, and C, the noise component of a Winnow algorithm's mistake bound depends on the value of the multiplier. A smaller multiplier is more resistant to noise. This matches the results we see in Table 5.9. When the multiplier is small the Winnow algorithms benefit from a large u value. However, as the multiplier grows, even $u = 2$ causes an increase in the number of mistakes.

The results of varying u are more positive with voting. The graph labeled voting in Figure 5.7 gives the total number of mistakes averaged over all the voting algorithms. Table 5.10 gives the same results for the individual algorithms. In general, a larger u value improves the performance for voting. There are a few exceptions exceptions, but on average the larger u value lowers the number of mistakes. This is probably a result of the larger hypothesis diversity that results from the large u values. A larger u value will cause more updates with instances which tends tends to increase the diversity as

Name	$\hat{\mathrm{M}}$(R(1))	$\hat{\mathrm{M}}$(R(2))	$\hat{\mathrm{M}}$(R(3))	$\hat{\mathrm{M}}$(R(4))	$\hat{\mathrm{M}}$(R(5))
Perceptron	50926 ± 76	50180 ± 52	**49807** ± 75	50029 ± 78	50350 ± 78
ALMA(2)	48914 ± 72	**48363** ± 82	48583 ± 85	48934 ± 85	49317 ± 85
ALMA($\ln n$)	53091 ± 89	**52209** ± 83	52414 ± 80	52793 ± 77	53163 ± 77
Bal(1.05)	50782 ± 77	49675 ± 69	**49673** ± 72	49910 ± 68	50162 ± 70
Bal(1.2)	50888 ± 76	**50288** ± 75	50482 ± 76	50810 ± 77	51177 ± 82
Bal(1.4)	**53703** ± 91	53878 ± 80	54577 ± 85	55216 ± 91	55766 ± 84
Bal(1.7)	**58924** ± 86	59517 ± 79	60271 ± 89	60989 ± 80	61470 ± 86
Bal(2.0)	**62837** ± 85	63520 ± 79	64281 ± 91	64820 ± 96	65291 ± 84
UWin(1.05)	63041 ± 79	58030 ± 68	56036 ± 78	55143 ± 75	**54537** ± 79
UWin(1.2)	58735 ± 68	55633 ± 76	**55043** ± 66	55092 ± 73	55350 ± 75
UWin(1.4)	60417 ± 73	**58979** ± 80	59211 ± 82	59749 ± 78	60314 ± 76
UWin(1.7)	64862 ± 72	**64280** ± 76	64774 ± 85	65372 ± 76	65942 ± 85
UWin(2.0)	69016 ± 83	**68626** ± 83	69201 ± 84	69748 ± 100	70385 ± 163
UCWin(1.05)	66768 ± 71	60388 ± 72	58151 ± 73	57031 ± 69	**56491** ± 82
UCWin(1.2)	57946 ± 70	54775 ± 70	**54319** ± 80	54486 ± 75	54853 ± 79
UCWin(1.4)	56905 ± 81	**55602** ± 70	56024 ± 74	56662 ± 81	57330 ± 77
UCWin(1.7)	58406 ± 71	**58271** ± 78	59012 ± 90	60025 ± 156	61171 ± 262
UCWin(2.0)	**60109** ± 79	60400 ± 79	61522 ± 147	62936 ± 286	64378 ± 291
NWin(1.05,.3)	85230 ± 72	74034 ± 73	69792 ± 66	67611 ± 76	**66338** ± 80
NWin(1.2,.3)	64192 ± 74	60204 ± 72	**59469** ± 71	59489 ± 77	59771 ± 79
NWin(1.4,.3)	61872 ± 73	**60471** ± 74	60955 ± 70	61658 ± 78	62302 ± 89
NWin(1.7,.3)	63935 ± 75	**63887** ± 76	64755 ± 83	65622 ± 82	66296 ± 86
NWin(2.0,.3)	**66664** ± 73	67078 ± 70	67918 ± 91	68683 ± 80	69311 ± 80
NWin(1.05,.5)	57966 ± 82	55309 ± 70	54714 ± 86	**54597** ± 85	54703 ± 78
NWin(1.2,.5)	55295 ± 70	**53949** ± 60	54193 ± 70	54710 ± 85	55242 ± 77
NWin(1.4,.5)	56125 ± 63	**55974** ± 74	56911 ± 70	57784 ± 67	58537 ± 74
NWin(1.7,.5)	**59080** ± 71	59845 ± 75	60954 ± 73	61926 ± 80	62679 ± 90
NWin(2.0,.5)	**61877** ± 75	62795 ± 75	63882 ± 78	64713 ± 87	65360 ± 88
NWin(1.05,.7)	62901 ± 81	58347 ± 82	56827 ± 84	56210 ± 71	**55919** ± 80
NWin(1.2,.7)	55290 ± 75	**53534** ± 70	53566 ± 70	54005 ± 75	54514 ± 75
NWin(1.4,.7)	54890 ± 73	**54686** ± 69	55565 ± 76	56402 ± 82	57151 ± 84
NWin(1.7,.7)	**56953** ± 68	57723 ± 74	58852 ± 76	59742 ± 79	60474 ± 84
NWin(2.0,.7)	**59205** ± 83	60225 ± 81	61310 ± 72	62161 ± 79	62818 ± 79

Table 5.9: Number of mistakes made by instance recycling on the basic algorithms with the number of times an instance can be used set to $u = \{1, 2, 3, 4, 5\}$. The remaining parameters are set at the default values.

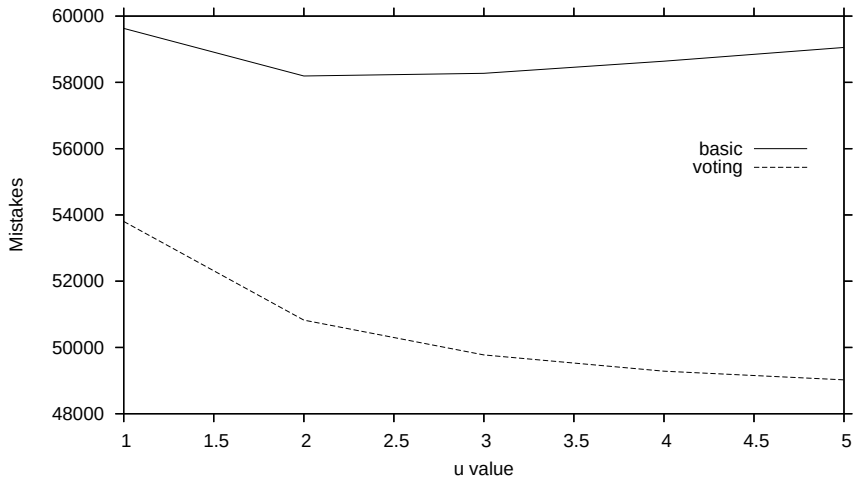

Figure 5.7: Average number of total mistakes for all basic and voting algorithms when using $u = \{1, 2, 3, 4, 5\}$. All other parameters set to default value.

explained in Section 3.2. In addition, the recycled instances are less likely to contain a skewed distribution of labels since an instance can be used multiple times. As can be seen, this gives us the best algorithm so far with VR-Combine making only 39578 mistakes when $u = 5$. The extra diversity of the hypotheses used most likely explains the large u value for VR-Combine.

Best Parameters

The final experiment for this chapter is to take the best results on VR-Combine from the previous parameter experiments and see how much these values improve the algorithms. We give the total number of mistakes in Table 5.11 and the mistakes in the last 500 trials in Table 5.12. The tables use VR1 to represent the voting and instance recycling algorithm with default parameters and VR2 to represent the algorithms with $r = 500$, $f = 0.9$, $h = 20$, $s = 500$, and $u = 5$. As can be seen, all the algorithms have a large statistically significant decrease in total mistakes over the default parameters. In the final 500 trials, the algorithms also have a decrease in mistake count from the default parameters. The result is not always statistically significant, but this could be a result of the smaller number of trials we are testing.

These parameters give us the best algorithm we have seen on these sets of data.

Name	$\hat{M}$(VR(1))	$\hat{M}$(VR(2))	$\hat{M}$(VR(3))	$\hat{M}$(VR(4))	$\hat{M}$(VR(5))
Perceptron	46179 ± 77	44100 ± 65	43371 ± 63	43057 ± 71	**42913** ± 66
ALMA(2)	45276 ± 67	43783 ± 77	43321 ± 77	43121 ± 78	**43026** ± 77
ALMA($\ln n$)	49341 ± 89	47294 ± 88	46585 ± 83	46264 ± 77	**46120** ± 75
Bal(1.05)	46153 ± 77	44010 ± 70	43279 ± 70	42984 ± 63	**42831** ± 66
Bal(1.2)	45987 ± 74	44111 ± 69	43458 ± 71	43139 ± 73	**42952** ± 69
Bal(1.4)	47338 ± 79	45913 ± 70	45503 ± 74	45351 ± 69	**45285** ± 79
Bal(1.7)	50672 ± 86	49494 ± 77	49185 ± 80	49023 ± 77	**48919** ± 79
Bal(2.0)	53580 ± 83	52452 ± 82	51988 ± 92	51714 ± 86	**51513** ± 88
UWin(1.05)	59204 ± 70	53575 ± 68	50968 ± 70	49524 ± 66	**48560** ± 66
UWin(1.2)	54182 ± 67	49872 ± 65	48210 ± 63	47340 ± 68	**46864** ± 64
UWin(1.4)	54334 ± 84	51125 ± 75	50108 ± 76	49643 ± 80	**49448** ± 72
UWin(1.7)	57082 ± 85	54717 ± 75	53930 ± 77	53611 ± 78	**53438** ± 81
UWin(2.0)	59707 ± 83	57810 ± 76	57223 ± 89	56994 ± 88	**56974** ± 116
UCWin(1.05)	62696 ± 77	55948 ± 80	52932 ± 77	51098 ± 67	**49946** ± 66
UCWin(1.2)	54068 ± 72	49278 ± 63	47582 ± 71	46743 ± 71	**46310** ± 63
UCWin(1.4)	51939 ± 79	48658 ± 67	47683 ± 75	47278 ± 67	**47142** ± 71
UCWin(1.7)	52371 ± 66	50247 ± 71	49676 ± 73	49506 ± 83	**49476** ± 89
UCWin(2.0)	53520 ± 79	51862 ± 73	**51480** ± 81	51510 ± 115	51715 ± 156
NWin(1.05,.3)	79418 ± 101	69317 ± 83	64533 ± 79	61739 ± 76	**59939** ± 84
NWin(1.2,.3)	59791 ± 63	54254 ± 71	52224 ± 70	51311 ± 76	**50828** ± 80
NWin(1.4,.3)	55926 ± 69	52514 ± 83	51561 ± 82	51243 ± 85	**51097** ± 86
NWin(1.7,.3)	56413 ± 83	54233 ± 85	53737 ± 79	53589 ± 76	**53582** ± 84
NWin(2.0,.3)	58163 ± 78	56523 ± 76	56113 ± 87	56051 ± 83	**55981** ± 85
NWin(1.05,.5)	53568 ± 87	49460 ± 74	47858 ± 84	47077 ± 73	**46551** ± 74
NWin(1.2,.5)	50493 ± 71	47467 ± 61	46482 ± 70	46025 ± 68	**45837** ± 69
NWin(1.4,.5)	50277 ± 62	48005 ± 63	47501 ± 65	**47381** ± 66	47384 ± 73
NWin(1.7,.5)	51778 ± 70	50400 ± 78	**50152** ± 75	50201 ± 79	50260 ± 82
NWin(2.0,.5)	53725 ± 76	52641 ± 74	52481 ± 77	**52455** ± 78	52465 ± 71
NWin(1.05,.7)	59755 ± 79	53897 ± 78	51390 ± 81	49970 ± 76	**49085** ± 70
NWin(1.2,.7)	51091 ± 70	47607 ± 64	46394 ± 62	45866 ± 69	**45646** ± 65
NWin(1.4,.7)	49587 ± 70	47227 ± 71	46679 ± 64	**46582** ± 68	46587 ± 69
NWin(1.7,.7)	50260 ± 71	48847 ± 68	48591 ± 70	**48552** ± 66	48644 ± 65
NWin(2.0,.7)	51647 ± 77	50594 ± 79	**50416** ± 73	50426 ± 71	50464 ± 75
Combine	41029 ± 68	39960 ± 69	39699 ± 67	39606 ± 63	**39581** ± 68

Table 5.10: Number of mistakes made by instance recycling on the voting algorithms with the number of times an instance can be used set to $u = \{1, 2, 3, 4, 5\}$. The remaining parameters are set at the default values.

Name	$\hat{\mathrm{M}}$(Name)	$\hat{\mathrm{M}}$(VR1-Name)	$\hat{\mathrm{M}}$(VR2-Name)
Per	73342 ± 80	46179 ± 77	**40924** ± 57
ALMA(2)	66066 ± 95	45276 ± 67	**40921** ± 58
ALMA($\ln n$)	71023 ± 105	49341 ± 89	**43564** ± 77
Bal(1.05)	72989 ± 83	46153 ± 77	**40791** ± 59
Bal(1.2)	72398 ± 83	45987 ± 74	**41168** ± 65
Bal(1.4)	74887 ± 86	47338 ± 79	**44270** ± 69
Bal(1.7)	81075 ± 109	50672 ± 86	**48798** ± 84
Bal(2.0)	86641 ± 89	53580 ± 83	**52166** ± 92
UWin(1.05)	93375 ± 93	59204 ± 70	**46609** ± 63
UWin(1.2)	88163 ± 93	54182 ± 67	**45493** ± 65
UWin(1.4)	88900 ± 87	54334 ± 84	**48730** ± 78
UWin(1.7)	94672 ± 103	57082 ± 85	**53480** ± 87
UWin(2.0)	100624 ± 89	59707 ± 83	**56975** ± 80
CUWin(1.05)	99476 ± 98	62696 ± 77	**47865** ± 67
CUWin(1.2)	85792 ± 91	54068 ± 72	**44462** ± 66
CUWin(1.4)	81676 ± 96	51939 ± 79	**45770** ± 68
CUWin(1.7)	81567 ± 95	52371 ± 66	**48498** ± 155
CUWin(2.0)	82557 ± 78	53520 ± 79	**51000** ± 228
NWin(1.05,.3)	110746 ± 76	79418 ± 101	**56081** ± 81
NWin(1.2,.3)	88760 ± 89	59791 ± 63	**48058** ± 69
NWin(1.4,.3)	85753 ± 96	55926 ± 69	**49476** ± 88
NWin(1.7,.3)	88529 ± 85	56413 ± 83	**53002** ± 89
NWin(2.0,.3)	92840 ± 97	58163 ± 78	**56051** ± 81
NWin(1.05,.5)	86721 ± 91	53568 ± 87	**44580** ± 63
NWin(1.2,.5)	80307 ± 102	50493 ± 71	**44004** ± 69
NWin(1.4,.5)	79405 ± 91	50277 ± 62	**46096** ± 62
NWin(1.7,.5)	81935 ± 91	51778 ± 70	**49576** ± 87
NWin(2.0,.5)	85610 ± 111	53725 ± 76	**52228** ± 80
NWin(1.05,.7)	86297 ± 97	59755 ± 79	**46520** ± 69
NWin(1.2,.7)	77574 ± 82	51091 ± 70	**43670** ± 62
NWin(1.4,.7)	76053 ± 93	49587 ± 70	**45183** ± 64
NWin(1.7,.7)	77541 ± 90	50260 ± 71	**47720** ± 69
NWin(2.0,.7)	80087 ± 92	51647 ± 77	**49959** ± 82
Combine		41029 ± 68	**37393** ± 65

Table 5.11: Number of mistakes over all concepts made with VR1 using the default parameters and VR2 using $r = 500$, $f = 0.9$, $h = 20$, $s = 500$, $u = 5$.

Name	$\hat{\mathrm{M}}$(Name)	$\hat{\mathrm{M}}$(VR1-Name)	$\hat{\mathrm{M}}$(VR2-Name)
Per	4734 ± 22	3076 ± 17	$\mathbf{2801} \pm 17$
ALMA(2)	4304 ± 18	3035 ± 18	$\mathbf{2785} \pm 15$
ALMA($\ln n$)	4590 ± 25	3283 ± 17	$\mathbf{2924} \pm 16$
Bal(1.05)	4708 ± 22	3071 ± 15	$\mathbf{2789} \pm 16$
Bal(1.2)	4700 ± 21	3090 ± 16	$\mathbf{2826} \pm 18$
Bal(1.4)	4979 ± 23	3232 ± 18	$\mathbf{3100} \pm 19$
Bal(1.7)	5522 ± 24	3514 ± 18	$\mathbf{3442} \pm 17$
Bal(2.0)	5960 ± 30	3725 ± 17	$\mathbf{3689} \pm 19$
UWin(1.05)	5739 ± 23	3703 ± 17	$\mathbf{3069} \pm 16$
UWin(1.2)	5494 ± 20	3499 ± 16	$\mathbf{3121} \pm 15$
UWin(1.4)	5745 ± 22	3595 ± 17	$\mathbf{3366} \pm 14$
UWin(1.7)	6317 ± 25	3862 ± 15	$\mathbf{3723} \pm 17$
UWin(2.0)	6830 ± 25	4081 ± 17	$\mathbf{3967} \pm 16$
CUWin(1.05)	5758 ± 23	3814 ± 14	$\mathbf{3064} \pm 17$
CUWin(1.2)	5215 ± 25	3418 ± 15	$\mathbf{3017} \pm 16$
CUWin(1.4)	5181 ± 23	3408 ± 17	$\mathbf{3161} \pm 17$
CUWin(1.7)	5359 ± 21	3532 ± 18	$\mathbf{3365} \pm 22$
CUWin(2.0)	5516 ± 21	3640 ± 19	$\mathbf{3543} \pm 23$
NWin(1.05,.3)	6022 ± 29	4666 ± 27	$\mathbf{3300} \pm 19$
NWin(1.2,.3)	5301 ± 21	3601 ± 17	$\mathbf{3104} \pm 17$
NWin(1.4,.3)	5392 ± 25	3533 ± 17	$\mathbf{3326} \pm 16$
NWin(1.7,.3)	5787 ± 21	3697 ± 16	$\mathbf{3617} \pm 17$
NWin(2.0,.3)	6224 ± 20	3885 ± 17	$\mathbf{3855} \pm 14$
NWin(1.05,.5)	5300 ± 20	3440 ± 16	$\mathbf{2990} \pm 16$
NWin(1.2,.5)	5077 ± 19	3316 ± 15	$\mathbf{3018} \pm 16$
NWin(1.4,.5)	5131 ± 22	3355 ± 16	$\mathbf{3214} \pm 15$
NWin(1.7,.5)	5460 ± 24	3521 ± 16	$\mathbf{3482} \pm 15$
NWin(2.0,.5)	5804 ± 24	3690 ± 18	$\mathbf{3665} \pm 16$
NWin(1.05,.7)	5327 ± 22	3737 ± 19	$\mathbf{3057} \pm 17$
NWin(1.2,.7)	4955 ± 21	3318 ± 16	$\mathbf{2998} \pm 18$
NWin(1.4,.7)	4966 ± 21	3305 ± 16	$\mathbf{3160} \pm 17$
NWin(1.7,.7)	5212 ± 23	3426 ± 15	$\mathbf{3357} \pm 17$
NWin(2.0,.7)	5451 ± 23	3560 ± 18	$\mathbf{3517} \pm 18$
Combine		2861 ± 18	$\mathbf{2655} \pm 16$

Table 5.12: Number of mistakes made on the final 500 trials with VR1 using the default parameters and VR2 using $r = 500$, $f = 0.9$, $h = 20$, $s = 500$, $u = 5$.

The VR2-Combine algorithm only makes 37,393 mistakes on all the trials and 2,655 mistakes on the final 500 trials. In fact, the improvement in VR2-Combine in Table 5.11 is greater than the sum of improvements from the single parameter experiments. This implies the parameters interact in positive ways to decrease the number of mistakes. It is possible that further modification, most likely by increasing s and u, could give even more improvement.

Unfortunately, our parameter experiments have biased results. These results are biased because we learned the parameters based on experiments over the same data sets used to test the algorithms. However, the bias is most likely small since we only tested a small number of parameters. In practice, one could remove this bias. For example, one could use the common batch learning technique of using one set of instances to learn the parameters and another set to evaluate the algorithm. Another option, keeping with the on-line nature of the evaluation, is to run the algorithms with all the possible parameter values and use either WMA (see Section 2.5) or our previous voting techniques to combine predictions from the various algorithms. To keep costs reasonable, one could delete some of the algorithms at certain trials based on performance.

Based on the results, we recommend use $s = 500$ and $u = 5$, for future problems that do not contain an excessive amount of noise. If noise or computational cost is a problem then a smaller s and u value might be necessary. If cost is not a large factor, multiple s and u values can be used with a single VR-Combine algorithm to help optimize performance.

5.3 SVM Comparison

Our final set of experiments with fixed concepts allow us to gauge how well our techniques perform compared to an expensive globally optimizing linear-threshold algorithm. In this case, we compare against SVM^{light} [Joa98] which is an implementation of the Support Vector Machine (SVM) learning algorithm of Cortes and Vapnick [CV95]. SVM classification algorithms learn a hyperplane that maximizes the δ margin between the two classes. See Section 2.2 for more information on the margin. While an SVM

algorithm can learn more complicated target functions than a hyperplane, we restricted the experiments to hyperplanes to facilitate the comparison.

SVM algorithms can allow noisy instances on the wrong side of the margin. However, any instance that occurs on the wrong side incurs a penalty in the optimization problem. The amount of this penalty depends on the perpendicular distance to the correct side of the margin and the value of a parameter C. A large C value minimizes the number of noisy instances while a small value allows a larger margin at the expense of more noisy instances.

For our on-line experiments, we used twelve different C values. These values included the default C value used by SVM^{light}. Unfortunately, the default value performed poorly for many of the 186 fixed concept problems. Therefore, based on the recommendation of Jankowski and Grabczewski [JG03], we included eleven other C values. We used $C \in \{2^{-5}, 2^{-4}, \ldots, 2^{5}\}$. This range of values from [JG03] was justified based on the performance of an SVM algorithm using many of the same data sets as this dissertation.

Because the SVM^{light} algorithm solves a quadratic programming problem, it can be prohibitively expensive. This is even more so when using the algorithm in an on-line fashion. Therefore, we used many techniques to speed up execution.

First, we only update the SVM algorithm when a new instance could change the optimal hyperplane. This means the instance must be on the wrong side of the current margin. Therefore, the algorithm needs to reprocess all the instances to determine if this new instance should be considered a noisy instance or the current hyperplane should be changed. Second, we only run all twelve C values for the first 1000 trials. After this point, we only continue with the C parameter that is currently making the fewest mistakes. To convert this technique into a single on-line algorithm, we create a new algorithm, Combine-SVM, that always predicts with the SVM algorithm that is currently making the fewest mistakes. After 1000 trials, Combine-SVM predicts based on the single remaining SVM algorithm. Third, we performed a bootstrap sample using an average of twenty samples instead of fifty samples.

However, even using all of the techniques, the SVM^{light} algorithm was still too

$\hat{\mathrm{M}}$(VR1-Combine)	$\hat{\mathrm{M}}$(VR2-Combine)	$\hat{\mathrm{M}}$(Combine-SVM)
35795 ± 59	32884 ± 60	34395 ± 140

Table 5.13: Total Number of mistakes made on all trials of the 178 concepts with VR1 using the default parameters and VR2 using $r = 500$, $f = 0.9$, $h = 20$, $s = 500$, $u = 5$.

$\hat{\mathrm{M}}$(VR1-Combine)	$\hat{\mathrm{M}}$(VR2-Combine)	$\hat{\mathrm{M}}$(Combine-SVM)
2598 ± 16	2428 ± 14	2451 ± 26

Table 5.14: Number of mistakes on the final 500 trials of the 168 concepts with VR1 using the default parameters and VR2 using $r = 500$, $f = 0.9$, $h = 20$, $s = 500$, $u = 5$.

expensive for our time constraints. Sometimes a single bootstrap sample on a single concept would take over three weeks to complete. Therefore, we removed some concepts that were taking too long to execute. We removed a total of eight of the the 186 concepts: the connect-4 with label win, optdigits with label 0, shuttle with labels 1, 4, and 5, thyroid0387 with label -, yeast with label ERL, and the spambase concept.

In Figure 5.8, we give a scatter plot showing the total number of mistakes made on the 178 concepts by VR2-Combine on the x-axis and Combine-SVM on the y-axis. As can be seen the performance of the algorithms is comparable. VR2-Combine makes fewer mistakes on 113 of the concepts; however, in most cases, the algorithms give similar performance. In Table 5.13, we give the sum of the mistakes over the 178 concepts for VR1-Combine, VR2-Combine and Combine-SVM. VR2-Combine has a definite statistical advantage when summed over all the concepts. In Table 5.14, we give the sum of the mistakes for the final 500 trials of the 178 concepts. VR2-Combine still has an advantage, but there is limited statistical significance. In general, the superior performance and the greatly reduced computational cost make VR2-Combine a compelling choice when learning learning threshold functions.

5.4 Summary

This chapter continues the theme of improving the performance of adversarial on-line algorithms for problems where instances are generated by a fixed distribution. We give

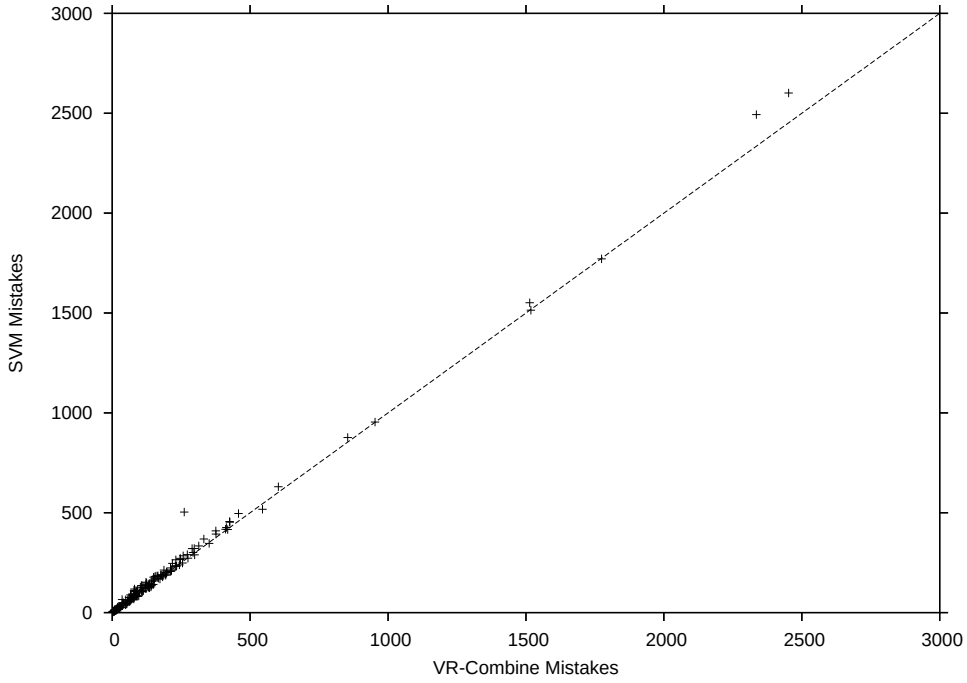

Figure 5.8: Scatter plot comparing VR2 using $r = 500$, $f = 0.9$, $h = 20$, $s = 500$, $u = 5$ with Combine-SVM.

a straightforward combination of the voting technique from Chapter 3 and the instance recycling technique of Chapter 4. Given a basic on-line algorithm B, the new algorithm that combines instance recycling and voting is called VR-B. We show VR-B improves the performance over either solitary technique using the fixed distribution, experimental framework of Chapter 3.

The main algorithm of this chapter is VR-Combine. It performs instance recycling on a set of basic algorithms, S, and uses the hypotheses generated from these algorithms to supply the voting procedure. VR-Combine is the algorithm we recommend for on-line learning with fixed distributions. It is a flexible algorithm with several useful parameters and includes the ability to use any number of basic algorithms. In our experiments, VR-Combine performs better than almost all of the VR-B algorithms. The only exception is a single concept out of the 186 concepts we tested.

We also explored other parameters that are used in the voting and instance recycling algorithms. We found that our default parameter settings are overly conservative

and that large performance gains are possible with different parameter settings. In particular, the VR-Combine has a large reduction in mistakes when the instance recycling parameter s and u are increased in value. This reduction in mistakes makes VR-Combine competitive with state of the art algorithms such as Support Vector Machines when learning linear-threshold functions. It is plausible that even fewer mistakes are possible if one uses multiple values of u and s for each basic algorithm, and if one increases the set of basic algorithms.

Chapter 6

Tracking Linear-threshold Concepts

In this chapter we give a mistake bound for tracking shifting concepts using a simple variation of the Unnormalized Winnow algorithm. We show that this version of Winnow can learn arbitrary linear-threshold functions that are allowed to change over the trials. This chapter extends the results found in [Mes02] and [Mes03].

Tracking is a natural advantage of the on-line model. Since the algorithm constantly receives new feedback on instances, it can use this feedback to refine the hypothesis. Even if the target function changes over the course of the trials, the algorithm can use the new information to, in principal, modify its hypothesis to the new target function. Of course, the mistake bound must depend on how much and how often the target function changes. In this chapter, we give such a bound for a version of the Winnow algorithm showing, in a limited sense, that the mistake bound depends on changes in the target function in close to an optimal way.

Our tracking version makes two modifications to the Unnormalized Winnow algorithm. The first is a standard modification to force the attribute weights to stay above a given constant, $\epsilon > 0$. Anytime a normal update attempts to lower a weight below ϵ, the weight is set to ϵ. Intuitively, this modification allows the algorithm to quickly learn a moving concept by not allowing the weights to get too small. When an attribute becomes relevant, its weight only needs to be increased from ϵ to the new value.

After proving a bound with the above algorithm, our second modification refines the result by preprocessing the instances to guarantee that the attribute weights can not get too large. While it is not obvious, very small positive attribute values can lead to large weights in the Winnow algorithm. These large weights increase the mistake bound, so we need some type of guarantee that the weights can not get too large. This

is accomplished by shifting the attributes so they are not allowed to get arbitrarily close to zero.

The minimum weight modification has been used with various Winnow type algorithms. In particular, it has been used to give mistake bounds for learning changing experts [LW94, HW98] and shifting disjunctions [AW98]. These algorithms have been shown to be useful in practice when learning shifting concepts such as predicting disk idle times for mobile applications [HLSS00] and solving load balancing problems on a network of computers [BB00]. This chapter builds on these results by giving a mistake bound for learning arbitrary linear-threshold functions that are allowed to change over the trials. The additional knowledge that some of these algorithms have good bounds when learning arbitrary shifting linear-threshold concepts may help justify applying these algorithms to a wider range of tracking problems.

There are other linear-threshold algorithms that can track concepts. In [Her01] a variation of the Perceptron algorithm is given that allows tracking concepts. In [KSW02], two algorithms are given that allow concept tracking. Of these three algorithms, the best bounds are obtained by a tracking version of ALMA found in [KSW02]. ALMA has a parameter p that allows the algorithm to behave like Perceptron when $p = 2$ and Winnow when $p = O(\ln n)$. Later in this chapter, we compare this tracking version of ALMA to the tracking version of Unnormalized Winnow.

Another advantage of our tracking version of Winnow is that it eliminates the dependence of the algorithm on the number of attributes. With an appropriate setting of parameters, instead of the algorithm depending on $\ln(n)$, as in the normal Winnow algorithm proof [Lit89, AW98], the algorithm depends on the maximum value of $\ln(\|X_t\|_1)$. In Appendix A, we show how these parameters can be used for the standard Unnormalized Winnow algorithm to achieve a similar mistake bound.

The remainder of the chapter is organized as follows. In Section 6.1, we give a formal statement of the concept tracking problem and explain the Tracking Unnormalized Winnow algorithm; we then present the main mistake bound. In Section 6.2, we give a proof of the main mistake bound along with a slight refinement. In Section 6.3, we give an alternative form of Tracking Unnormalized Winnow that works with complemented

Tracking Unnormalized Winnow$(\alpha, \sigma, \epsilon)$

Parameters

$\alpha > 1$ is the update multiplier.

$\epsilon > 0$ is the minimum value of the weights.

$\sigma \geq \epsilon$ is the starting value of the weights.

Initialization

$t \leftarrow 1$ is the current trial.

$\forall i \in \{1, \ldots, n\}$ $w_{i,1} = \sigma$ are the weights.

Trials

Instance: $\mathbf{x}_t \in [0,1]^n$.

Prediction: If $\mathbf{w_t} \cdot \mathbf{x_t} \geq 1$

Predict $\hat{y}_t = 1$ else predict $\hat{y}_t = -1$.

Update: Let $y_t \in \{-1, 1\}$ be the correct label.

If ($y_t = 1$ and $\hat{y}_t = -1$) then (promotion step)

$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = \alpha^{x_{i,t}} w_{i,t}$.

Else If ($y_t = -1$ and $\hat{y}_t = 1$) then (demotion step)

$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = \max(\epsilon, \alpha^{-x_{i,t}} w_{i,t})$.

Else

$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = w_{i,t}$.

$t \leftarrow t + 1$.

Figure 6.1: Pseudo-code for Tracking Unnormalized Winnow

attributes. In Section 6.4, we analyze the strengths and weaknesses of the bound and give an instance transformation that improves the bound when the attribute values become small. In Section 6.5, we compare the mistake bound to bounds on certain types of linear-threshold problems.

6.1 Tracking Problem Statement and Algorithm

In this section, we review the concept tracking Winnow algorithm and give the notation that is needed to understand an upper-bound on the number of mistakes.

In Figure 6.1, we give the modified Winnow algorithm for learning shifting concepts. This algorithm is based on the Unnormalized Winnow algorithm from Chapter 2. The tracking version's main difference is that the weights are not allowed to go below a minimum value ϵ. This minimum weight allows the algorithm to recover quickly when there is a change in the target concept [LW94]. The algorithm is identical to an algorithm given in [AW98].

The model of on-line tracking we present in this chapter is similar to models defined by Kuh, Petsche, and Rivest [KPR91] and Helmbold and Long [HL91]. The central element of the mistake bound is a sequence of target functions. Let $C = (C_1, \ldots, C_T)$ be a sequence of concepts with one concept per trial where T is the final trial. The concept is allowed to change each trial, and the bound will depend on how often and how much the concept changes. Intuitively, a good bound should be possible if the concept makes small changes or if the concept makes infrequent large changes. Allowing the concept to vary each trial allows us to represent changes that occur every trial; however, the concept might remain the same for large sequences of trials.

An adversary generates the instances, and the goal of the algorithm is to minimize the number of mistakes. The mistake bound will depend on the sequence of concepts and the amount of noise in the instances generated by the adversary.

We define concept C_t by specifying the weights, $\mathbf{u}_t$, of a hyperplane and a margin parameter, δ_t. Let $u_{1,t}, \ldots, u_{n,t} \geq 0$ where $u_{i,t} \in \mathbf{R}$, and $0 < \delta_t \leq 1$ where $\delta_t \in \mathbf{R}$. Each $\mathbf{u}_t$ and δ_t specifies a target function as explained in Section 2.2. In order to work with all these target functions, let $\delta = \min_{t \in \{1,\ldots,T\}} \delta_t$. The noise in an instance is defined as $\nu_t = \max[0, \delta - y_t(\mathbf{u_t} \cdot \mathbf{x_t} - 1)]$. An instance with $\nu_t = 0$ corresponds to an instance of the trial concept. An instance with $\nu_t > 0$ corresponds to a noisy instance, and ν_t is a measure of the noise in the linear-threshold function.

If the adversary is allowed to generate an arbitrary number of noisy instances, the concept would be impossible to learn. We can restrict the amount of noise by making assumptions on ν_t. For example, we can assume that $\sum_{t=1}^{T} \nu_t \leq N$ where $N \in \mathbf{R}$. This is similar to the noise model in [Lit89] where only a finite amount of noise is allowed over all trials. More information on bounding the amount of noise can be found in Section 2.2.

The mistake bound uses the following notation. This notation will be explained and motivated in the proof section. For notational convenience, we assume that T is the final trial, although our results can apply to a potentially infinite set of trials.

Terms used in the mistake bound

Let $\lambda \geq \max_{t\in\{1,\ldots,T\}} \|X_t\|_1$.

Let $\zeta = \min_{t\in\{1,\ldots,T\},\ i\in\{1,\cdots,n\}}\{x_{i,t} \mid x_{i,t} > 0\}$.

Let $H(C) = \sum_{i=1}^{n}(u_{i,T} + \sum_{t=1}^{T-1}\max(0, u_{i,t} - u_{i,t+1}))$.

Let $\nu_t = \max[0, \delta - y_t(\mathbf{u_t}\cdot\mathbf{x_t} - 1)]$.

Let $N = \sum_{t=1}^{T}\nu_t$

Theorem 6.1 *For instances generated by a concept sequence C, if $\alpha = 1+\delta$ and $\epsilon = \sigma = \frac{\delta}{50\lambda}$ then the number of mistakes is less than*

$$(2.05+\delta)\left(\frac{\zeta H(C)}{\delta(1+\delta)} + \frac{\ln\left(\frac{50\lambda}{\delta\zeta}\right)H(C)}{\delta^2} + \frac{N}{\delta(1+\delta)}\right) \quad .$$

We can use asymptotic notation to make this bound a little more digestible. Removing some lower order terms and using the fact that $0 < \zeta \leq 1$, $\zeta < \lambda$, and $0 < \delta \leq 1$, the number of mistakes is

$$O\left(H(C)\ln\left(\frac{50\lambda}{\delta\zeta}\right)/\delta^2 + N/\delta\right) \quad .$$

6.2 Proof of Mistake Bound

While most modern Winnow type proofs use a potential function to bound the number of mistakes [Lit89, Lit91, AW98, GLS01], we have found it useful to go back to the old style of proof used in [Lit88] to deal with tracking linear-threshold functions. While potential functions are useful for generalizing over a range of learning algorithm [GLS01, HW01, KSW02], it is a topic for future research to understand how this type of proof relates to the potential function technique.

The purpose of the next four lemmas is to give an upper-bound on the number of weight demotions as a function of the number of weight promotions. Remember that a promotion is an update that increases the weights on a mistake where the correct label is 1, and a demotion is an update that decreases the weights on a mistake where the correct label is 0. Intuitively, a bound must exist as long as every demotion removes at least a fixed constant of weight. This is because the algorithm always has positive weights and the only way the weights can increase is through a promotion.

Let $\triangle(f)$ represent the change in function f after a trial is completed. For example, $\triangle(\sum_{i=1}^{n} w_{i,t}) = \sum_{i=1}^{n} w_{i,t+1} - \sum_{i=1}^{n} w_{i,t}$.

Lemma 6.2 *On a promotion,* $\triangle(\sum_{i=1}^{n} w_{i,t}) < (\alpha - 1)$.

Proof After a promotion,

$$\sum_{i=1}^{n} w_{i,t+1} = \sum_{i=1}^{n} w_{i,t}\ \alpha^{x_{i,t}}\ .$$

Next, we use the fact that $\alpha^{x_{i,t}} \leq (\alpha-1)x_{i,t}+1$ for all $x_{i,t} \in [0,1]$. (This is true because $\alpha^{x_{i,t}}$ is a convex function.) This shows that,

$$\sum_{i=1}^{n} w_{i,t}\ \alpha^{x_{i,t}} \leq \sum_{i=1}^{n} w_{i,t}[(\alpha-1)x_{i,t}+1] = (\alpha-1)\sum_{i=1}^{n} w_{i,t}x_{i,t} + \sum_{i=1}^{n} w_{i,t}\ .$$

Since we have a promotion, $\sum_{i=1}^{n} w_{i,t}x_{i,t} < 1$. This combined with the above facts and remembering that $\alpha > 1$ gives

$$\sum_{i=1}^{n} w_{i,t+1} < (\alpha-1) + \sum_{i=1}^{n} w_{i,t}\ .$$

Therefore,

$$\triangle\left(\sum_{i=1}^{n} w_{i,t}\right) = \sum_{i=1}^{n} w_{i,t+1} - \sum_{i=1}^{n} w_{i,t} < (\alpha-1)\ .$$

■

Next we want to prove a similar result about demotions. However, because demotions have the added difficulty of not being able to lower the weight below ϵ, the proof is a little more complex. First we prove a small lemma that will help with the more difficult proof.

We are going to consider two types of attributes that occur during a demotion. Let $A \subseteq \{1,\ldots,n\}$ be all i such that $w_i\alpha^{-x_{i,t}} < \epsilon$. These are the indexes of weights that are forced to ϵ since $w_{i,t+1} = \max(\epsilon, \alpha^{-x_{i,t}}w_{i,t})$. Let $B \subseteq \{1,\ldots,n\}$ be the indexes of weights that have a normal demotion. These are the attributes such that $x_{i,t} > 0$ and $w_{i,t}\alpha^{-x_{i,t}} \geq \epsilon$. All the other weights do not change.

Lemma 6.3 *On a demotion, if* $\sum_{i\in B} w_{i,t}x_{i,t} \geq \theta$ *then* $\triangle\left(\sum_{i\in B} w_{i,t}\right) \leq \frac{1-\alpha}{\alpha}\theta$.

Proof After a demotion,

$$\sum_{i \in B} w_{i,t+1} = \sum_{i \in B} w_{i,t} \ \alpha^{-x_{i,t}}.$$

Next, we use the fact that $\alpha^{-x_{i,t}} \leq \left(\frac{1}{\alpha} - 1\right) x_{i,t} + 1$ for all $x_{i,t} \in [0,1]$. (This fact is true because $\alpha^{-x_{i,t}}$ is a convex function.) This shows that

$$\sum_{i \in B} w_{i,t} \ \alpha^{-x_{i,t}} \leq \sum_{i \in B} w_{i,t} \left[\left(\frac{1}{\alpha} - 1 \right) x_{i,t} + 1 \right] = \frac{1-\alpha}{\alpha} \sum_{i \in B} w_{i,t} x_{i,t} + \sum_{i \in B} w_{i,t} \ .$$

Based on the assumption of the lemma, $\sum_{i \in B} w_{i,t} x_{i,t} \geq \theta$. This combined with the above facts and remembering that $\alpha > 1$ gives

$$\sum_{i \in B} w_{i,t+1} \leq \frac{1-\alpha}{\alpha} \theta + \sum_{i \in B} w_{i,t} \ .$$

Therefore,

$$\triangle \left(\sum_{i \in B} w_{i,t} \right) = \sum_{i \in B} w_{i,t+1} - \sum_{i \in B} w_{i,t} \leq \frac{1-\alpha}{\alpha} \theta \ .$$

■

Here is the main lemma concerning demotions. This lemma gives an upper-bound on how much the weights decrease after a demotion. The amount of decrease will depend on various factors including $\lambda \geq \max_{t \in \{1,\ldots,T\}} \|X_t\|_1$. Also this lemma assumes that $\epsilon < 1/\lambda$. This is a reasonable assumption that ensures the algorithm can lower weights during demotions. Otherwise the algorithm could be forced to always make a mistake on certain instances for many linear-threshold concepts. For example, assume there are λ attributes that are not part of the target concept. An adversary can give these attributes value 1 on every instance and every other attribute value 0. This creates an instance with label 0. If $\epsilon \geq 1/\lambda$ then the algorithm will always incorrectly predict 1 since

$$\sum_{i=1}^{n} w_{i,t} x_{i,t} \geq \sum_{i=1}^{n} \epsilon x_{i,t} = \epsilon \sum_{i=1}^{n} x_{i,t} = \epsilon \lambda \geq 1.$$

Lemma 6.4 *On a demotion, if* $\epsilon < 1/\lambda$ *then* $\triangle \left(\sum_{i=1}^{n} w_{i,t} \right) < \frac{1-\alpha}{\alpha}(1 - \epsilon\lambda)$.

Proof We are going to use the set of attributes A and B defined earlier. We will break the proof into two cases. For the first case assume that there is at least one attribute

in B. Since a demotion has occurred we know,

$$\sum_{i\in A} w_{i,t}x_{i,t} + \sum_{i\in B} w_{i,t}x_{i,t} = \sum_{i=1}^{n} w_{i,t}x_{i,t} \geq 1 \ .$$

From this we can use Lemma 6.3 to derive that

$$\triangle\left(\sum_{i\in B} w_{i,t}\right) \leq \frac{1-\alpha}{\alpha}\left(1 - \sum_{i\in A} w_{i,t}x_{i,t}\right) \ .$$

Now we want to get $\sum_{i\in A} w_{i,t}x_{i,t}$ into a more useful form. Let $v_{i,t}$ represent the amount $w_{i,t}$ is above ϵ.

$$\sum_{i\in A} w_{i,t}x_{i,t} = \sum_{i\in A}(\epsilon + v_{i,t})x_{i,t} = \epsilon\sum_{i\in A} x_{i,t} + \sum_{i\in A} v_{i,t}x_{i,t} < \epsilon\lambda + \sum_{i\in A} v_{i,t} \ .$$

Being careful to keep track of what is negative, we can substitute this into the previous formula.

$$\triangle\left(\sum_{i\in B} w_{i,t}\right) < \frac{1-\alpha}{\alpha}\left(1 - \epsilon\lambda - \sum_{i\in A} v_{i,t}\right) \ .$$

Next, since all the weights with an index in A get demoted to ϵ, we know that for all $i \in A$, the weight $v_{i,t}$ will be removed. Therefore $\triangle\left(\sum_{i\in A} w_{i,t}\right) = -\sum_{i\in A} v_{i,t}$. Combining this information proves the first case where B has at least one element.

$$\triangle\left(\sum_{i=1}^{n} w_{i,t}\right) = \triangle\left(\sum_{i\in B} w_{i,t}\right) + \triangle\left(\sum_{i\in A} w_{i,t}\right)$$

$$< \frac{1-\alpha}{\alpha}\left(1 - \epsilon\lambda - \sum_{i\in A} v_{i,t}\right) - \sum_{i\in A} v_{i,t} \leq \frac{1-\alpha}{\alpha}\left(1 - \epsilon\lambda\right) \ .$$

The second case, where B is empty, is similar. Using some of the same notation,

$$\triangle\left(\sum_{i=1}^{n} w_{i,t}\right) = \triangle\left(\sum_{i\in A} w_{i,t}\right) = -\sum_{i\in A} v_{i,t} \ .$$

Since a demotion has occurred, we know that $\sum_{i=1}^{n} w_{i,t}x_{i,t} \geq 1$, but since the only active attributes are in A,

$$1 \leq \sum_{i=1}^{n} w_{i,t}x_{i,t} = \sum_{i\in A}(\epsilon + v_{i,t})x_{i,t} = \epsilon\sum_{i\in A} x_{i,t} + \sum_{i\in A} v_{i,t}x_{i,t} \leq \epsilon\lambda + \sum_{i\in A} v_{i,t} \ .$$

We can use this to bound $\triangle\left(\sum_{i=1}^{n} w_{i,t}\right)$.

$$\triangle\left(\sum_{i=1}^{n} w_{i,t}\right) = -\sum_{i\in A} v_{i,t} \leq \epsilon\lambda - 1 < \frac{1-\alpha}{\alpha}(1 - \epsilon\lambda) \ .$$

The last step of the inequality is true since we assumed $\epsilon < 1/\lambda$. ■

Now we can combine the lemmas to give an upper-bound on the number of demotions as a function of the number of promotions. Let $P = \{t \mid \text{promotion mistake on trial } t\}$, and let $D = \{t \mid \text{demotion mistake on trial } t\}$.

Lemma 6.5 *If $\epsilon < 1/\lambda$ then at any trial, $|D| < \frac{\alpha(\sigma-\epsilon)n}{(\alpha-1)(1-\epsilon\lambda)} + \frac{\alpha}{1-\epsilon\lambda}|P|$.*

Proof We know $\sum_{i=1}^{n} w_{i,t}$ can never go below ϵn, and we know $\sum_{i=1}^{n} w_{i,t}$ has a value of σn at the beginning of the algorithm. Since the weights can only change during demotions and promotions, we can relate the minimum weight to the current weight as follows.

$$\epsilon n \leq \sigma n + \sum_{t \in P} \triangle \left(\sum_{i=1}^{n} w_{i,t} \right) + \sum_{t \in D} \triangle \left(\sum_{i=1}^{n} w_{i,t} \right) \quad .$$

Using the upper-bounds from Lemma 6.2 and Lemma 6.4,

$$\epsilon n < \sigma n + (\alpha - 1)|P| + \frac{1-\alpha}{\alpha}(1 - \epsilon\lambda)|D| \quad .$$

Noting that $1 - \alpha$ is negative and $1 - \epsilon\lambda$ is positive, we can rearrange the inequality to prove the lemma. ■

Our goal at this stage is to show how the relevant weights[1] increase in weight during the running of the algorithm. If we can show that they must eventually increase, and that they have a maximum value, we can derive a mistake bound on the algorithm. First we want to give a bound on the maximum value of a weight. Let w_{max} be the maximum value of any weight, and $\zeta = \min_{t \in \{1,\ldots,T\},\ i \in \{1,\cdots,n\}} \{x_{i,t} \mid x_{i,t} > 0\}$.

Lemma 6.6 *If $\alpha < e$ then $w_{max} \leq \max\left(\sigma, \frac{\alpha^{\zeta}}{\zeta}\right)$.*

Proof If a promotion never occurs the maximum weight is the initial weight, σ. The only time a weight, $w_{a,t}$ can increase is after a promotion where $x_{a,t} > 0$. Promotions

[1]Relevant weights are defined as weights w_i where the corresponding target weights have $u_i > 0$

can only occur if $\sum_{i=1}^{n} w_{i,t}x_{i,t} < 1$. Therefore for any attribute $x_{a,t} > 0$,

$$w_{a,t}x_{a,t} \leq \sum_{i=1}^{n} w_{i,t}x_{i,t} < 1 \ .$$

Since the new weight is $\alpha^{x_{a,t}}w_{a,t}$, we want to find $\max(\alpha^x w)$ over all x and w where $0 < \zeta \leq x \leq 1$ and $wx < 1$. Since any feasible solution to the above problem can be changed to increase the maximum by increasing either w or x until $wx = 1$, we can get an upper-bound on the maximum weight by setting $wx = 1$. This transforms the problem to $\max(\alpha^x/x)$ over all x given that $0 < \zeta \leq x \leq 1$.

We can use calculus to show that the maximum must occur at $x = \zeta$. Let $f(x) = \alpha^x/x$. To find the critical points, we solve

$$f'(x) = \alpha^x \ln(\alpha)/x - \alpha^x/x^2 = 0$$

This gives a single critical point at $x_c = 1/\ln(\alpha)$. Since we are assuming that $\alpha < e$ then $x_c = 1/\ln(\alpha) > 1/\ln(e) = 1$. Therefore the maximum must occur at one of the endpoints since the critical point is past the right endpoint of $[\zeta, 1]$. Assume the maximum occurs at $x = 1$. Since $\alpha < e$,

$$f'(1) = \alpha \ln(\alpha) - \alpha < \alpha - \alpha = 0.$$

Therefore the slope of $f(x) = \alpha^x/x$ must be negative at $x = 1$. Since $f(x)$ is continuous, there must be a point $x_a < 1$ such that $f(x_a) > f(1)$. This is a contradiction since we assumed $f(1)$ was the maximum value in the interval. The maximum must occur at $x = \zeta$. This gives an upper-bound on any weight of $\max(\sigma, \alpha^\zeta/\zeta)$. ■

The next lemma deals with the effects of the demotions and promotions on a sequence of target concepts. Let $H(C) = \sum_{i=1}^{n} \left(u_{i,T} + \sum_{t=1}^{T-1} \max(0, u_{i,t} - u_{i,t+1}) \right)$ and let $\nu_t = \max[0, \delta - y_t(\mathbf{u_t} \cdot \mathbf{x_t} - 1)]$ where $\delta \in (0, 1]$.

Lemma 6.7 *If $\alpha < e$ then*

$$\log_\alpha \left(\frac{w_{max}}{\epsilon} \right) H(C) + N - \log_\alpha \left(\frac{\sigma}{\epsilon} \right) \sum_{i=1}^{n} u_{i,1} > (1+\delta)|P| - (1-\delta)|D| \ .$$

Proof Let $z_{i,t} = \log_\alpha (w_{i,t}/\epsilon)$. This is just the amount weight i has been increased by an α factor over the minimum value ϵ before the update on trial t. Based on this formula, $z_{i,1} = \log_\alpha (\sigma/\epsilon)$. Assume that the last trial is T. Because an update could occur on this final trial, the last index of z is $T+1$.

The value of $w_{i,t}$ is ϵ multiplied by $\alpha^{x_{i,t}}$ on every promotion and effectively multiplied by a number as small as $\alpha^{-x_{i,t}}$ on every demotion. The multiplier on demotions may be larger since the weight value can not go below ϵ. For example, if the current weight value is ϵ, a demotion would not change the value; therefore, in this case, the effective multiplier is 1. Using the definition of z, gives $z_{i,t+1} - z_{i,t} = x_{i,t}$ on a promotion, $z_{i,t+1} - z_{i,t} \geq -x_{i,t}$ on a demotion, and $z_{i,t+1} - z_{i,t} = 0$ on a trial without an update.

Let P be the set of promotion trials and let D be the set of demotion trials. At this point, we want to weight the change in the z values by u, the target weights. This will allow us to relate the z values to the mistakes.

$$\sum_{i=1}^{n}\sum_{t=1}^{T} u_{i,t}(z_{i,t+1} - z_{i,t}) \geq \sum_{t\in P}\sum_{i=1}^{n} u_{i,t}x_{i,t} - \sum_{t\in D}\sum_{i=1}^{n} u_{i,t}x_{i,t} \ .$$

Let $\hat{P} = \{t \mid t \in P \text{ and } \nu_t > 0\}$ and $\hat{D} = \{t \mid t \in D \text{ and } \nu_t > 0\}$. These are the noisy promotion and demotion trials. Using this notation, we can break up the summations. The last formula is equal to

$$\sum_{t\in P-\hat{P}}\sum_{i=1}^{n} u_{i,t}x_{i,t} - \sum_{t\in D-\hat{D}}\sum_{i=1}^{n} u_{i,t}x_{i,t} + \sum_{t\in \hat{P}}\sum_{i=1}^{n} u_{i,t}x_{i,t} - \sum_{t\in \hat{D}}\sum_{i=1}^{n} u_{i,t}x_{i,t} \ .$$

Since for every non-noisy promotion $\sum_{i=1}^{n} u_{i,t}x_{i,t} \geq (1+\delta)$, and every non-noisy demotion $\sum_{i=1}^{n} u_{i,t}x_{i,t} \leq (1-\delta)$, then the last formula is greater or equal to

$$(1+\delta)|P-\hat{P}| - (1-\delta)|D-\hat{D}| + \sum_{t\in \hat{P}}\sum_{i=1}^{n} u_{i,t}x_{i,t} - \sum_{t\in \hat{D}}\sum_{i=1}^{n} u_{i,t}x_{i,t}$$

$$= (1+\delta)|P| - (1-\delta)|D| - \sum_{t\in\hat{P}}\left((1+\delta) - \sum_{i=1}^{n} u_{i,t}x_{i,t}\right) - \sum_{t\in\hat{D}}\left((1-\delta) + \sum_{i=1}^{n} u_{i,t}x_{i,t}\right).$$

Using the definitions of noisy promotion and demotion trials along with the definition $N = \sum_{t=1}^{T} \nu_t$, we can use the previous equations to conclude that

$$\sum_{i=1}^{n}\sum_{t=1}^{T} u_{i,t}(z_{i,t+1} - z_{i,t}) \geq (1+\delta)|P| - (1-\delta)|D| - N \ .$$

Based on the definition of z, $0 \leq z_{i,t} < \log_\alpha\left(\frac{w_{max}}{\epsilon}\right)$. We can use this constraint to compute an upper-bound. Using the fact that $z_{i,1} = \log_\alpha(\sigma/\epsilon)$,

$$(1+\delta)|P| - (1-\delta)|D| - N \leq \max\left(\sum_{i=1}^{n} u_{i,1}(z_{i,2} - z_{i,1}) + \cdots + u_{i,T}(z_{i,T+1} - z_{i,T})\right)$$

$$= \max\left(\sum_{i=1}^{n} -z_{i,1}u_{i,1} + z_{i,2}(u_{i,1} - u_{i,2}) + \cdots + z_{i,T}(u_{i,T-1} - u_{i,T}) + z_{i,T+1}u_{i,T}\right)$$

$$= \max\left(\sum_{i=1}^{n} z_{i,2}(u_{i,1} - u_{i,2}) + \cdots + z_{i,T}(u_{i,T-1} - u_{i,T}) + z_{i,T+1}u_{i,T}\right) - \sum_{i=1}^{n} u_{i,1}z_{i,1}$$

$$< \log_\alpha\left(\frac{w_{max}}{\epsilon}\right)\sum_{i=1}^{n}\left(u_{i,T} + \sum_{t=1}^{T-1}\max(0, u_{i,t} - u_{i,t+1})\right) - \log_\alpha\left(\frac{\sigma}{\epsilon}\right)\sum_{i=1}^{n} u_{i,1} \quad .$$

Rearranging the terms proves the lemma. ■

In its current form, the above lemma is not very intuitive. However, there is another way to look at the bound. Define $u_{i,0} = 0$, $u_{i,T+1} = 0$, and $h(i) = \sum_{t=1}^{T}\max(0, u_{i,t} - u_{i,t+1})$. Notice that $H(C) = \sum_{i=1}^{n} h(i)$. The value $h(i)$ is equivalent to summing the local maximums and subtracting the local minimums for target weight $\mathbf{u}_i$. For example, if the sequence of $\mathbf{u}_i$ is $(0, .1, .3, .5, .5, .2, .1, .4, .2, 0)$ then $h(i) = .5 - .1 + .4 = .8$. This is because $u_{i,t} - u_{i,t+1}$ is only positive while the target weights are decreasing in value. The target weights decrease in value from a local maximum, $u_{i,a}$, to the next local minimum, $u_{i,b}$. The sum of these differences, as we go from $u_{i,a}$ to $u_{i,b}$, is just $u_{i,a} - u_{i,b}$.

This suggests how an adversary can maximize the number of mistakes. The adversary should increase a weight as much as possible during the maximum target value and decrease the weight to ϵ during the minimum. This requires that the adversary knows the future behavior of the sequence of concepts in order to know if the current concept is a local maximum or a local minimum. For some practical problems, the adversary may not have this knowledge. However, the bounds in this chapter assume the adversary knows the future sequence of concepts.

At this point, we want to prove the main theorem. We have attempted to set the parameters and arrange the inequalities to optimize the mistake bound for small δ.

Theorem 6.1 *If $\alpha = 1+\delta$, $\epsilon = \sigma = \frac{\delta}{50\lambda}$ then the number of mistakes made by Tracking Unnormalized Winnow is less than*

$$(2.05 + \delta)\left(\frac{\zeta H(C)}{\delta(1+\delta)} + \frac{\ln\left(\frac{50\lambda}{\delta\zeta}\right) H(C)}{\delta^2} + \frac{N}{\delta(1+\delta)}\right) \ .$$

Proof First we want to show that $w_{max} \leq \alpha^\zeta/\zeta$. Based on Lemma 6.6, we know that $w_{max} \leq \max(\sigma, \alpha^\zeta/\zeta)$. Therefore, we just need to show that $\sigma \leq \alpha^\zeta/\zeta$. Using the facts that $\zeta \leq \lambda$, $\delta \leq 1$, and that $\alpha > 1$,

$$\sigma = \frac{\delta}{50\lambda} \leq \frac{1}{50\zeta} < \frac{\alpha^\zeta}{\zeta} \ .$$

Next we want to substitute Lemma 6.5 into Lemma 6.7 to get an upper-bound on $|P|$. The lemma condition that $\alpha < e$ is satisfied since $\alpha = 1 + \delta \leq 2 < e$. The condition that $\epsilon < 1/\lambda$ is satisfied since $\epsilon\lambda = \delta/50 < 1$. Now we can proceed with the substitution. Using the fact that $\epsilon = \sigma$, we derive

$$H(C)\log_\alpha\left(\frac{\alpha^\zeta}{\epsilon\zeta}\right) + N > (1+\delta)|P| - (1-\delta)\frac{\alpha}{1-\epsilon\lambda}|P| \ .$$

Solving for $|P|$ gives

$$|P| < \frac{H(C)\log_\alpha\left(\frac{\alpha^\zeta}{\epsilon\zeta}\right) + N}{(1+\delta) - (1-\delta)\frac{\alpha}{1-\epsilon\lambda}}$$

as long as $(1+\delta) > \frac{\alpha(1-\delta)}{1-\epsilon\lambda}$. It is not difficult to verify this for our choices of α and ϵ.

To get a mistake bound add $|P|$ to both sides of Lemma 6.5 and substitute in the previous result to get a bound on the number of mistakes.

$$|P| + |D| < \left(1 + \frac{\alpha}{1-\epsilon\lambda}\right)|P| < \left(1 + \frac{\alpha}{1-\epsilon\lambda}\right)\frac{H(C)\log_\alpha\left(\frac{\alpha^\zeta}{\epsilon\zeta}\right) + N}{(1+\delta) - (1-\delta)\frac{\alpha}{1-\epsilon\lambda}}$$

$$= \frac{1 - \epsilon\lambda + \alpha}{(1+\delta)(1-\epsilon\lambda) - (1-\delta)\alpha}\left(\zeta H(C) + \frac{H(C)\ln\left(\frac{1}{\epsilon\zeta}\right)}{\ln(\alpha)} + N\right) \ .$$

Substituting in the values $\alpha = 1 + \delta$ and $\epsilon = \frac{\delta}{50\lambda}$, the preceding equation is equal to

$$\frac{100 + 49\delta}{49\delta(1+\delta)}\left(\zeta H(C) + \frac{H(C)\left(\ln\left(\frac{50\lambda}{\delta\zeta}\right)\right)}{\ln(1+\delta)} + N\right) \ .$$

To make the bound more intuitive, we use the fact that $\ln(1+\delta) \geq \delta - \delta^2/2$. (One can prove this using the Taylor formula with a fourth term remainder.) The previous equation is less than or equal to

$$(2.05+\delta)\left(\frac{\zeta H(C)}{\delta(1+\delta)} + \frac{\ln\left(\frac{50\lambda}{\delta\zeta}\right)H(C)}{\delta^2} + \frac{N}{\delta(1+\delta)}\right) \ .$$

■

The previous theorem can be slightly improved when $\epsilon < 1/n$. In Appendix A, we show that a good choice of σ for the fixed concept Unnormalized Winnow algorithm is $1/n$. This is also true for the tracking version. As long as $\epsilon < 1/n$, we can set $\sigma = 1/n$ to get a small decrease in the number of mistakes.

Theorem 6.8 *If $\alpha = 1+\delta$, $\epsilon = \frac{\delta}{50\lambda} < 1/n$, and $\sigma = 1/n$ then the number of mistakes made by Tracking Unnormalized Winnow is less than*

$$\frac{2.05\left(1-\frac{\delta n}{50\lambda}\right)}{\delta^2} + (2.05+\delta)\left(\frac{\zeta H(C)}{\delta+\delta^2} + \frac{\ln\left(\frac{50\lambda}{\delta\zeta}\right)H(C) - \ln\left(\frac{50\lambda}{\delta n}\right)\sum_{i=1}^{n} u_{i,1}}{\delta^2} + \frac{N}{\delta+\delta^2}\right) \ .$$

Proof First we want to show that $w_{max} \leq \alpha^\zeta/\zeta$. Based on Theorem 6.6, we know that $w_{max} \leq \max(\sigma, \alpha^\zeta/\zeta)$. Therefore, we just need to show that $\sigma \leq \alpha^\zeta/\zeta$. Using the facts that $\alpha > 1$ and $0 < \zeta \leq 1$

$$\sigma = \frac{1}{n} \leq 1 < \frac{\alpha^\zeta}{\zeta} \ .$$

Next we want to substitute Lemma 6.5 into Lemma 6.7 to get an upper-bound on $|P|$. The lemma condition that $\alpha < e$ is satisfied since $\alpha = 1+\delta \leq 2 < e$. The condition that $\epsilon < 1/\lambda$ is satisfied since $\epsilon\lambda = \delta/50 < 1$. Now we can proceed with the substitution to derive an upper-bound on $|P|$.

$$|P| < \frac{\frac{\alpha(1-\delta)(\sigma-\epsilon)n}{(1-\alpha)(1-\epsilon\lambda)} + \log_\alpha\left(\frac{\alpha^\zeta}{\epsilon\zeta}\right)H(C) - \log_\alpha\left(\frac{\sigma}{\epsilon}\right)\sum_{i=1}^{n} u_{i,1} + N}{1+\delta-\frac{\alpha(1-\delta)}{1-\epsilon\lambda}}$$

as long as $(1+\delta) > \frac{\alpha(1-\delta)}{1-\epsilon\lambda}$. It is not difficult to verify this for our choices of α and ϵ.

To get a mistake bound add $|P|$ to both sides of Lemma 6.5.

$$|P| + |D| < \frac{\alpha(\sigma - \epsilon)n}{(1-\alpha)(1-\epsilon\lambda)} + \left(1 + \frac{\alpha}{1-\epsilon\lambda}\right)|P| \ .$$

Next substitute the previous upper-bound on $|P|$ and the values $\alpha = 1 + \delta$, $\sigma = 1/n$, and $\epsilon = \frac{\delta}{50\lambda}$ into the preceding formula.

$$\frac{100\lambda - 2\delta n}{49\lambda\delta^2} + \frac{100 + 49\delta}{49\delta(1+\delta)}\left(\zeta H(C) + \frac{\ln\left(\frac{50\lambda}{\delta\zeta}\right)H(C) - \ln\left(\frac{50\lambda}{\delta n}\right)\sum_{i=1}^{n} u_{i,1}}{\ln(1+\delta)} + N\right) \ .$$

To make the bound more intuitive, we use the fact that $\ln(1+\delta) \geq \delta - \delta^2/2$. The above formula is less than or equal to

$$\frac{2.05\left(1 - \frac{\delta n}{50\lambda}\right)}{\delta^2} + (2.05+\delta)\left(\frac{\zeta H(C)}{\delta + \delta^2} + \frac{\ln\left(\frac{50\lambda}{\delta\zeta}\right)H(C) - \ln\left(\frac{50\lambda}{\delta n}\right)\sum_{i=1}^{n} u_{i,1}}{\delta^2} + \frac{N}{\delta + \delta^2}\right) \ .$$

■

Notice that this bound is similar to the bound in Theorem 6.1. The only difference are the extra terms

$$\frac{2.05\,(1 - \epsilon n)}{\delta^2} \qquad \text{and} \qquad \frac{-(2.05+\delta)\ln\left(\frac{1}{\epsilon n}\right)\sum_{i=1}^{n} u_{i,1}}{\delta^2} \ .$$

Both of these terms go to zero as ϵ approaches $1/n$. Also, because $\ln(1/x) \geq 1 - x$ for $x \in (0, 1]$ and $\sum_{i=1}^{n} u_{i,1} \geq 1$, the net effect of these terms is to decrease the number of mistakes up until this point. Therefore, according to these theorems, one should set $\sigma = \max(1/n, \epsilon)$ to minimize the mistake bound. However, as the number of concept shifts increase, these terms have a smaller effect because $H(C)$ dominates $\sum_{i=1}^{n} u_{i,1}$. In these cases, this refinement has a negligible effect on the mistake bound.

6.3 Complemented Algorithm

The Unnormalized Winnow algorithm is well suited to solving certain types of learning problems. Given our notation, if the target function is composed of shifting disjunctions where each disjunction has k literals then the target weights can be set to 2 and $\delta_t = 1$ for each concept. If the target function is composed of shifting conjunctions where

Complemented Tracking Unnormalized Winnow$(\alpha, \sigma, \epsilon)$

Parameters

$\alpha > 1$ is the update multiplier.

$\epsilon > 0$ is the minimum value of the weights.

$\sigma \geq \epsilon$ is the starting value of the weights.

Initialization

$t \leftarrow 1$ is the current trial.

$\forall i \in \{1, \ldots, n\}$ $w_{i,1} = \sigma$ are the weights.

Trials

Instance: $\mathbf{x}_t \in [0,1]^n$. Let $\bar{x}_{i,t} = 1 - x_{i,t}$

Prediction: If $\mathbf{w_t} \cdot \bar{\mathbf{x}}_\mathbf{t} \geq 1$

Predict $\hat{y}_t = -1$ else predict $\hat{y}_t = 1$.

Update: Let $y_t \in \{-1, 1\}$ be the correct label.

If ($y_t = -1$ and $\hat{y}_t = 1$) then (promotion step)

$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = \alpha^{\bar{x}_{i,t}} w_{i,t}$.

Else If ($y_t = 1$ and $\hat{y}_t = -1$) then (demotion step)

$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = \max(\epsilon, \alpha^{-\bar{x}_{i,t}} w_{i,t})$.

Else

$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = w_{i,t}$.

$t \leftarrow t + 1$.

Figure 6.2: Pseudo-code for Complemented Tracking Unnormalized Winnow

each conjunction has k literals then the target weights can be set to $2/(2k-1)$ and $\delta_t = 1/k$ for each concept. According to our theorems, the target function for tracking conjunctions gives a mistake bound close to k times worse than tracking disjunctions. However, there is not much difference between a conjunction and a disjunction. The only difference is based on the label of true and false.

As described in Section 2.3.1, we can complement the Unnormalized Winnow algorithm to allow it to treat conjunctions like disjunctions. We give the algorithm for the Tracking Complemented Unnormalized Winnow in Figure 6.2. The algorithm is similar to the Tracking Unnormalized Winnow algorithm except that it works with a slightly modified set of instances and labels. Every attribute $x_{i,t}$ is converted to $\bar{x}_{i,t} = (1 - x_{i,t})$. This is called complementing the attribute [Lit89]. In the same way, every label, including the algorithm's prediction, is flipped. This allows the algorithm to change the target function. For example, this transformation converts a conjunction into a disjunction.

To express the full bound of the Complemented Tracking Unnormalized Winnow,

we need some new notation.

Terms used in the complemented mistake bound

Let $\hat{u}_{i,t} = u_{i,t}/(-1 + \sum_{i=1}^{n} u_{i,t})$.

Let $\left\|\bar{X}_t\right\|_1 = (\bar{x}_{1,t}, \bar{x}_{2,t}, \ldots, \bar{x}_{n,t})$.

Let $\hat{\lambda} \geq \max_{t\in\{1,\ldots,T\}} \left\|\bar{X}_t\right\|_1$.

Let $\hat{\zeta} = \min_{t\in\{1,\ldots,T\},\ i\in\{1,\cdots,n\}}\{\bar{x}_{i,t} \mid \bar{x}_{i,t} > 0\}$.

Let $\hat{H}(C) = \sum_{i=1}^{n}(\hat{u}_{i,T} + \sum_{t=1}^{T-1} \max(0, \hat{u}_{i,t} - \hat{u}_{i,t+1}))$.

Let $0 < \hat{\delta} \leq \min_{t\in\{1,\ldots,T\}} \delta_t/(-1 + \sum_{i=1}^{n} u_{i,t})$.

Let $\hat{N} = N/(-1 + \sum_{i=1}^{n} u_{i,t})$

Theorem 6.9 *For instances generated from a concept sequence C, if $\alpha = 1 + \hat{\delta}$, $\epsilon = \sigma = \frac{\hat{\delta}}{50\hat{\lambda}}$ then the number of mistakes made by Complemented Unnormalized Tracking Winnow is less than*

$$(2.05 + \hat{\delta})\left(\frac{\hat{\zeta}\hat{H}(C)}{\hat{\delta}(1+\hat{\delta})} + \frac{\ln\left(\frac{50\hat{\lambda}}{\hat{\delta}\hat{\zeta}}\right)\hat{H}(C)}{\hat{\delta}^2} + \frac{\hat{N}}{\hat{\delta}(1+\hat{\delta})}\right) .$$

Proof The target function from concept C_t is of the form

$$\text{Predict 1 if} \quad \sum_{i=1}^{n} u_{i,t}x_{i,t} \geq 1 + \delta;$$

$$\text{Predict -1 if} \quad \sum_{i=1}^{n} u_{i,t}x_{i,t} \leq 1 - \delta.$$

This is equivalent to

$$\text{Predict 1 if} \quad \sum_{i=1}^{n} u_{i,t}(1 - x_{i,t}) \leq \sum_{i=1}^{n}(u_{i,t}) - 1 - \delta;$$

$$\text{Predict -1 if} \quad \sum_{i=1}^{n} u_{i,t}(1 - x_{i,t}) \geq \sum_{i=1}^{n}(u_{i,t}) - 1 + \delta$$

To get this closer to a form used by Tracking Unnormalized Winnow, we normalize.

$$\text{Predict 1 if} \quad \frac{\sum_{i=1}^{n} u_{i,t}(1 - x_{i,t})}{\sum_{i=1}^{n}(u_{i,t}) - 1} \leq 1 - \frac{\delta}{\sum_{i=1}^{n}(u_{i,t}) - 1};$$

$$\text{Predict -1 if} \quad \frac{\sum_{i=1}^{n} u_{i,t}(1 - x_{i,t})}{\sum_{i=1}^{n}(u_{i,t}) - 1} \geq 1 + \frac{\delta}{\sum_{i=1}^{n}(u_{i,t}) - 1};$$

If we flip the labels of this target function then it is in a form that can be used by Tracking Unnormalized Winnow. Therefore, we use Tracking Unnormalized Winnow to learn the target function with the complemented attributes, but we flip the labels returned by the environment.

When the Unnormalized Winnow algorithm makes a prediction, it is predicting the flipped label. Therefore, we flip the answer returned by Tracking Unnormalized Winnow to give the predicted label for the original problem. Plugging this target function into Theorem 6.1 and taking into account the complemented attributes proves the theorem. ■

A similar result can be derived from Theorem 6.8. Therefore, the complemented algorithm should set $\sigma = \max(1/n, \epsilon)$ to minimize the number of mistakes.

6.4 Analyzing the mistake bound

In this section, we analyze the Tracking Unnormalized Winnow mistake bound. This analysis applies to both our previous proofs. To make this exposition clear we will use an asymptotic form of the bound. The number of mistakes of Tracking Unnormalized Winnow is equal to

$$O\left(H(C)\ln\left(\frac{50\lambda}{\zeta\delta}\right)/\delta^2 + N/\delta\right) \quad .$$

6.4.1 Advantages of the Algorithm

One of the advantages of the tracking version of Unnormalized Winnow is the relative insensitivity to the number of irrelevant attributes. This can be seen by looking at $\lambda \geq \max_{t\in\{1,\ldots,T\}} \|X_t\|_1$. Based on this definition, we can always set λ to the number of attributes. Imagine we have a learning problem with n_1 attributes. Let U be the target function that optimizes the mistake bound. Now assume that we increase the number of attributes by n_2. Since the target concept must still be valid, the only effect on the bound is the potential increase in the value of λ from n_1 to $n_1 + n_2$. Since the bound only contains λ inside the logarithm function, this at worst causes a small increase

in the bound. Of course, the bound may not increase at all. If the added attributes allow a better target function then this new target can lower the bound. Also, the new attributes may not increase the value of λ; they can increase the value of λ by anything from 0 to n_2.

A related benefit of the algorithm is that the mistake bound only depends on λ as opposed to n. For sparse problems, with few active attributes ($x_i > 0$), this could have a large effect on the bound. Most upper-bounds on mistakes, for algorithms with multiplicative updates, depend on the number of attributes [Lit89, AW98, HW98, GLS01]. However, in Appendix A, we show how to use σ to extend this benefit to the normal version of Unnormalized Winnow.

There is one practical problem with sparse instances. Since Winnow algorithms only have positive weights, a trick is used to allow negative weights. For each attribute x_i, a new complemented attribute $1 - x_i$ is added. A positive weight on $1 - x_i$ can effectively allow a negative weight on x_i. However, these extra attributes will force $\lambda = n$. Therefore, the advantages of a small λ are only possible if one can assume the target concept only has positive weights and does not need complemented attributes.

It is possible to get similar sparse instance benefits with Winnow type algorithms by using an infinite attribute algorithm [BHL91]. Infinite attribute algorithms take advantage of the fact that only attributes that are active during updates affect the algorithm. Therefore the number of attributes that are involved in updates is just the maximum number of attributes active per trial times the mistake bound. Combining this with the logarithmic nature of the Winnow bounds gives a mistake bound that only depends logarithmically on the maximum number of attributes active per trial. However there are certain advantages to the proofs in this chapter. First $\max_{t \in T} \|X_t\|_1$ may be small, yet the total number of active attributes may be large. Second when a potentially infinite amount of noise is involved in on-line learning, there cannot be a finite mistake bound, and a large number of attributes could eventually be active during mistakes. Since the analysis in this chapter does not depend on the total number of active attributes during mistakes, these problems do not occur.

6.4.2 Algorithm Problems and Solutions

One important limitation of the preceding result is that the mistake bound is allowed to grow arbitrarily large when $\zeta = \min\{x_{i,t} \mid x_{i,t} > 0\}$ approaches zero. There are several ways to overcome this problem.

In Appendix E, we give an alternative proof that shows Tracking Unnormalized Winnow has a finite mistake bound even when ζ is allowed to approach 0. The proof is similar to our previous proofs. We assume that $\alpha = 1 + \delta$ and that $\epsilon = \sigma = \frac{\delta}{50\lambda}$. In addition, we assume that $\lambda \geq (1 + \delta)/50$. With these assumptions, we get an upper-bound of

$$\frac{2.05+\delta}{1+\delta}\left[\frac{2H(C)}{\delta^2(2-\delta)}\left(\ln\frac{2.05H(C)}{\delta^2(2+\delta-\delta^2)}\right)^2+\left(\frac{2\ln(50\lambda)\,H(C)}{\delta^2(2-\delta)}+\frac{N}{\delta}\right)\ln\frac{2.05H(C)}{\delta^2(2+\delta-\delta^2)}\right].$$

However, it is clear this bound is not tight because the proof assumes the adversary maximally increases the weight of every relevant attribute on every promotion. This is not possible especially with shifting concepts. A more effective way to avoid small ζ values is to increase the value of ζ by transforming the instances. With the appropriate setting of parameters this transformation gives a better bound.

In Figure 6.3, we give a procedure called Shift that shifts the values of attributes that are too close to zero. Any value in $(0, m/2]$ gets shifted to 0; any value in $(m/2, m)$ gets shifted to m. To show how these transformed instances change the learning problem, we need some extra notation. Let $v = \max_{t\in\{1,\ldots,T\}} \sum_{i=1}^{n} u_{i,t}$. This is just the maximum sum of target weights on any single trial. The smaller v, the better the bound. This corresponds to the assumption that the concept has few relevant attributes. Even in problems with shifting concepts, while the total number of attributes involved might be large, the number in a single trial might be relatively small.

Corollary 6.10 *Let $m = \delta/v$. If procedure Shift$(\mathbf{x}, m)$ is applied to all instances generated by concept sequence C then the number of mistakes made by Tracking Unnormalized Winnow is at most*

$$O\left(H(C)\ln\left(\frac{v\lambda}{\delta}\right)/\delta^2 + N/\delta\right) \quad .$$

Shift$(\mathbf{x}, m)$
Parameters
$m > 0$ is the new ζ value.
$\mathbf{x} = (x_1, \ldots, x_n) \in [0,1]^n$ is the instance.
Transformation
if $x_i \in [0, m/2]$
$x_i \leftarrow 0$
else if $x_i \in (m/2, m]$
$x_i \leftarrow m$
else
$x_i \leftarrow x_i$

Figure 6.3: Instance transformation for small ζ values.

Proof We use the hat notation to refer to a parameter dealing with the transformed instance. The $H(C)$ function is the same since the target function is the same. We have $\hat{\zeta} \geq \delta/v$ because the transformation forces this new minimum value, and $\hat{\lambda} \leq 2\lambda$ since we are at most doubling the value of any single attribute. To compute $\hat{\delta}$, consider the case where the label is 1. A non-noisy instance has $\sum_{i=1}^{n} u_{i,t} x_{i,t} \geq 1 + \delta$.

$$\sum_{i=1}^{n} u_{i,t}\hat{x}_{i,t} \geq \sum_{i=1}^{n} u_{i,t}(x_{i,t} - m/2) \geq \sum_{i=1}^{n} u_{i,t}x_{i,t} - vm/2 \geq 1 + \delta - \delta/2 \ .$$

Therefore we can set $\hat{\delta} = \delta/2$. The same result holds for -1 labels. Noise is handled in a similar way. Again consider the case where the label is 1. The noise is defined as $\nu_t = \max(0, 1 + \delta_t - \sum_{i=1}^{n}(u_{i,t}x_{i,t}))$. Therefore,

$$1 + \hat{\delta} - \sum_{i=1}^{n} u_{i,t}\hat{x}_{i,t} \leq 1 + \hat{\delta} - \sum_{i=1}^{n} u_{i,t}(x_{i,t} - m/2)$$

$$\leq 1 + \hat{\delta} + vm/2 - \sum_{i=1}^{n} u_{i,t}x_{i,t} = 1 + \delta - \sum_{i=1}^{n} u_{i,t}x_{i,t} \ .$$

This shows that the noise can only decrease on the transformed instances. The same result holds for -1 labels. Plugging these values into Theorem 6.1 gives the result. ■

One problem with this transformation is the addition of extra algorithm parameters. Besides needing to know values for δ and λ, we need a value for v in order to set the parameters. A standard technique for this problem is to run several copies of the algorithm with different choices for the parameters. These copies would be used

as experts in an algorithm such as the Weighted Majority Algorithm (WMA) [Lit88, LW94, CBFH$^+$97]. As long we do not run an extremely large number of copies, the number of mistakes made by WMA should be close to the performance of the algorithm with the best parameter values. See Section 2.5 for more information.

6.5 Bounds on Specific Problems

In this section, we compare our mistake bounds to other published results. This comparison includes fixed concepts, shifting disjunctions, and shifting linear-threshold functions.

6.5.1 Fixed Concept

First we consider the bound for a fixed concept, C^f. To fix concept C^f, assume $\forall t_1 \in \{1, \ldots, T\}$ and $\forall t_2 \in \{1, \ldots, T\}$ that $u_{i,t_1} = u_{i,t_2}$.

Corollary 6.11 *When instances are generated from C^f, if $\alpha = 1 + \delta$, $\epsilon = \frac{\delta}{50\lambda}$, and $\sigma = \max(1/n, \epsilon)$ then the number of mistakes of tracking Winnow when $\sigma = 1/n$ is at most*

$$\frac{2.05\left(1 - \frac{\delta n}{50\lambda}\right)}{\delta^2} + (2.05 + \delta)\left(\frac{\zeta \sum_{i=1}^{n} u_{i,1}}{\delta + \delta^2} + \frac{\ln\left(\frac{n}{\zeta}\right)\sum_{i=1}^{n} u_{i,1}}{\delta^2} + \frac{N}{\delta + \delta^2}\right) .$$

and the number of mistakes when $\sigma > 1/n$ is at most

$$(2.05 + \delta)\left(\frac{\zeta \sum_{i=1}^{n} u_{i,1}}{\delta + \delta^2} + \frac{\ln\left(\frac{50\lambda}{\delta\zeta}\right)\sum_{i=1}^{n} u_{i,1}}{\delta^2} + \frac{N}{\delta + \delta^2}\right)$$

Proof When $\epsilon \leq 1/n$ use Theorem 6.8 with the fact that $H(C) = \sum_{i=1}^{n} u_{i,1}$. When $\epsilon > 1/n$ use Theorem 6.1. ■

Comparing these bound with the general concept tracking bound, it is clear the main difference is the value of the $H(C)$ function. The $H(C)$ function encodes the extra difficulty of tracking a changing concept.

For the Unnormalized Winnow algorithm, when $\alpha = 1 + \delta$ and the starting weights are σ, the number of mistakes is at most

$$\frac{(2+\delta)\left(n\epsilon + \sum_{i=1}^{n} u_{i,1} \ln u_{i,1} + \sum_{i=1}^{n} u_{i,1} \ln(1/\epsilon) - \sum_{i=1}^{n} u_{i,1}\right)}{\delta^2} + \frac{(2+\delta^2/5)N}{\delta} \quad .$$

Using $\sigma = 1/n$, the number of mistakes is less than or equal to

$$\frac{(2+\delta)\left(1 + \sum_{i=1}^{n} u_{i,1} \ln u_{i,1} + \ln n \sum_{i=1}^{n} u_{i,1} - \sum_{i=1}^{n} u_{i,1}\right)}{\delta^2} + \frac{(2+\delta^2/5)N}{\delta} \quad .$$

See Appendix A for more details.

Looking at just the main terms in the bounds, several differences are evident. The term $\ln(1/\zeta)$ in Tracking Unnormalized Winnow is partially the result of the weights getting larger than the corresponding weights in the target function. There is no corresponding term in normal Winnow bound since the weights of the algorithm must eventually get close to the target weights. Intuitively, if the weights do not get close then the adversary can make more mistakes exploiting this difference. However, this is not the best strategy for a shifting concept. If the adversary waits until a relevant attribute x_i becomes irrelevant then it can use all the weight in this newly irrelevant attribute to help perform demotions on any currently relevant attributes. The adversary can cause more demotions when x_i becomes irrelevant than when it is relevant. Therefore, the adversary tries to build up a large weight on x_i while it is relevant and then uses this weight for demotions when x_i becomes irrelevant. More precisely, as can be seen in Lemma 6.7, the adversary should maximize the weight values when u_i reaches a global maximum and minimize the weight value when u_i reaches a global minimum.

Another difference is how the standard Winnow bound depends on $\ln n$. In the target tracking proof, the bound depends on $\ln(\lambda/\delta)$, where λ is at least $\max \|X\|_1$. This difference is largely an effect of the starting weight values used by the algorithms. If we use the same starting weight value as the Tracking Unnormalized Winnow and use a slightly different representation of the concept, we can give a bound for Unnormalized Winnow that performs well on sparse instances. The proof exploits the fact that any set of concept weights that correctly classifies the instances gives a mistake bound. We use target weights that do well for sparse instances.

Theorem 6.12 *When instances are generated from C^f, if $\alpha = 1+0.98\delta$, $\sigma = \frac{\delta}{50\lambda}$, and only k target weights have $u_i > \sigma$ then, without loss of generality, we can assume the first k attributes have $u_i > \sigma$ and that the number of mistakes for the Unnormalized Winnow algorithm is less than*

$$\frac{(2.09 + 1.03\delta)\left(\sum_{i=1}^{k} u_i \ln u_i + \ln\left(\frac{50\lambda}{\delta}\right)\sum_{i=1}^{k} u_i\right)}{\delta^2} + \frac{(2.05 + \delta^2/5)N}{\delta}.$$

The proof can be found in Appendix A. This bound is similar to the second tracking Winnow bound except that it does not have a ζ parameter. The similar bound for the two algorithms shows that the key to getting a good bound on sparse instances for the Unnormalized Winnow algorithms is to properly set the starting weight of the attributes. One should set the starting weight values to whatever gives the minimum number of mistakes. Setting $\sigma = \max(50\lambda/\delta, 1/n)$ is a good choice.

6.5.2 Tracking Disjunctions

As a second example, we give a bound for shifting disjunctions with Boolean attributes. Let C^d be a concept sequence such that each $u_i^j \in \{0, 2\}$ and $H(C^d) = 2Z^+$. This gives $\delta = 1$ and corresponds to a sequence where Z^+ is the number of disjunction literals that are added to the concept either at the start of the algorithm or during the trials.

Corollary 6.13 *When instances $X \in \{0,1\}^n$ are generated from concept C^d with noise N, if $\alpha = 2$ and $\sigma = \frac{1}{50\lambda}$, then the number of mistakes is less than*

$$6.1Z^+ \ln \lambda + 27Z^+ + 1.53N \ .$$

Proof Concept sequence C^d has $H(C^d) = 2Z^+$, $\zeta = 1$ and $\delta = 1$. Substituting these values into Theorem 6.1 gives the result. ■

A similar bound for this algorithm is given in [AW98]. Let Z^- equal to the number of disjunction literals that are removed from the concept and let $Z = Z^+ + Z^-$. For the deterministic disjunction tracking algorithm given in Theorem 4 of [AW98] the number

of mistakes is less than or equal to

$$4.32Z \ln n + 16.9Z + \min(n, Z)(4.32 \ln n + 5.79) + 0.232 + 1.2N \ .$$

The bounds we give are similar and have a distinct advantage for sparse instances. However, our bounds are worse for large amounts of noise. This is largely due to our parameter choices though it is also effected by the approximations we use to deal with small δ values. Looking over Theorem 6.1, we can remove some of the approximations and set $\alpha = 1.36$ and $\sigma = \frac{1}{35\lambda}$ to derive a mistake bound of [2]

$$7.81Z^+ \ln \lambda + 30.2Z^+ + 1.2N \ .$$

The lower α value is used to improve the performance of the algorithm on noise. In general, there is a trade-off with the α parameter. One value of α optimizes the first portion of the bound that does not deal with noise, but a smaller α value lowers the constant on the noise term. This can be seen in the proof of Theorem 6.1.

We give lower bounds for learning disjunctions by slightly modifying the results of [Lit89, AW98] to handle sparse instances. In [AW98], they prove that any deterministic learning algorithm must generate at least

$$\lfloor (Z+1)/2 \rfloor \lfloor \log_2 (n) \rfloor + N$$

mistakes in the worst case when $n \geq 2$. The noise function corresponds to the target function mentioned at the start of the section. Therefore, N is always an even integer. It is not difficult to generalize the proof in [AW98] to handle sparse binary instances.

Theorem 6.14 *Let λ' equal the maximum number of attributes set to 1 in any instance. If $\lambda' \leq n/2$ then the maximum number of mistakes made by any deterministic algorithm when learning disjunctions is at least*

$$Z^+ \lfloor \log_2(2\lambda') \rfloor + N$$

[2] This upper-bound corrects an error found in [Mes03] where the bound was mistakenly given as $3.98Z^+ \ln \lambda + 15.37Z^+ + 1.2N$.

Proof For convenience assume $\lambda' = 2^v$. We create a sequence of concepts where each concept has only a single attribute in the disjunction. However, we do not specify which attribute is in the disjunction and instead specify the disjunction based on the predictions of the learning algorithm in order to force mistakes.

We break the proof into two pieces. Consider the first $Z^+ - 1$ concepts. For each of these concepts, the adversary does not generate any noisy instances. The adversary always forces the label to be the opposite of the algorithm's prediction. By setting λ attributes to 1, after the first trial at least λ attributes must be consistent with the label. During the next trial, the algorithm sets $\lambda/2$ attributes to 1. Again at most $\lambda/2$ attributes can be consistent with the label. This repeats until there is only one attribute consistent with the label. Over the $Z^+ - 1$ concepts, this forces at least $(Z^+ - 1)\log_2(2\lambda)$ mistakes.

For the final concept, the adversary proceeds in the same way until there are only two possible attributes that match the concept. This forces an additional $\log_2(\lambda)$ mistakes. For the next $N+1$ trials, the adversary uses the same instance. The instance is 1 for the first possible attribute and 0 for the other attributes. Recall that, based on the target function for disjunctions, N is an even integer. The adversary forces the label to be the opposite of the algorithm's prediction for each of these trials. Notice that one of the two attributes must be consistent with the labels for $N/2+1$ of the trials. Therefore only $N/2$ of the trials are noisy instances. Based on the target function, $\nu_t = 2$ for each of these noisy instances. Therefore the total amount of noise is at most N and the total number of mistakes is $Z^+ \log_2(2\lambda') + N$. ■

Therefore our bound for disjunctions with sparse instances is within a constant of the lower bound.

6.5.3 Tracking Linear-threshold Functions

To get a better feel for the strength of the Tracking Unnormalized Winnow bound, we compare it to the Unnormalized Winnow bound on a modified on-line tracking problem. Assume the environment gives additional information to the learning algorithm by revealing when the concept is going to change. A straightforward use of the Unnormalized Winnow algorithm to solve this problem is to reset the weights on every concept change. The mistake bound for this problem can be derived from Theorem A.3 by summing the bounds for each concept.

Assume there are k concepts and let the weights for concept j be specified by $\mathbf{u}^j$ and the margin by δ_j. The largest term for learning the target function is

$$\sum_{j=1}^{k} \frac{(2+\delta)\ln(n)\left(\sum_{i=1}^{n} u_i^j\right)}{(\delta_j)^2}.$$

If all the δ_j are identical one can just sum up all the target weights. Otherwise the sum is weighted with smaller δ_j causing a greater increase in the mistake bound.

Now consider the Tracking Unnormalized Winnow algorithm. From Theorem 6.1, its largest term for learning the target function is

$$\frac{(2.05+\delta)\ln\left(\frac{50\lambda}{\delta\zeta}\right)H(C)}{\delta^2}.$$

This algorithm has some advantages over the previous Unnormalized Winnow modification. First, it does not need to know when the concept changes. Second, we effectively sum the weights in the $H(C)$ function. However if a concept change shares relevant attributes from the previous concept then $H(C)$ is lowered. Another potential advantage is when λ is small; however, as we have seen, the Normalized Winnow algorithm can adjust its parameters to change the $\ln n$ factor in the above bound to $\ln(50\lambda/\delta)$.

The previous observation also results in a disadvantage of Tracking Unnormalized Winnow. When $\ln\left(\frac{50\lambda}{\delta\zeta}\right)$ is much larger than $\ln(n)$, the tracking bound is probably worse. However because of the logarithmic nature of the bound, and the fact that $\lambda \leq n$, the tracking bound is generally not much worse.[3] Another disadvantage is the

[3] If ζ is very small then one can use Corollary 6.10.

fact that Tracking Unnormalized Winnow depends on the smallest δ_j value over all the concepts. However, this is somewhat an artifact of the analysis. Therefore, somewhat surprisingly, the Tracking Unnormalized Winnow algorithm has a comparable bound to a "cheating" version of Unnormalized Winnow. It even has the potential to do better for problems that involve a gradually shifting target function.

A related concept tracking algorithm is based on a modification of ALMA.[4] Pseudo-code for this modified ALMA algorithm can be found in Figure 6.4. It is a simplification of the original ALMA algorithm, and it provably allows the algorithm to track linear-threshold functions [KSW02]. The only difference between the tracking and normal versions of ALMA is that the tracking algorithm uses a fixed update constant η and a fixed algorithm margin $\hat{\delta}$, while the original version dynamically adjusts these parameters based on the instances and the number of updates. However, both algorithms have a parameter p that allows the algorithms to have Perceptron like bounds when $p = 2$ and Winnow like bounds when $p = O(\log n)$.

To help compare Tracking ALMA and Tracking Unnormalized Winnow, remember that $v = \max_{t \in \{1,\ldots,T\}} \sum_{i=1}^{n} u_{i,t}$. The bound for ALMA when $p = \log n$ is

$$O\left(\frac{vH(C)\ln n}{\delta^2} + \frac{N}{\delta}\right) .$$

When δ and v are small and λ is large, this compares favorably to the Winnow bound in Corollary 6.10 of

$$O\left(\frac{H(C)\ln\left(v\lambda/\delta\right)}{\delta^2} + \frac{N}{\delta}\right) .$$

However, when δ is small both bounds may become too large to be useful because of the common $1/\delta^2$ factor.

We need some new notation to give the bounds for ALMA when $p = 2$. Let $X_{max} = \max_{t \in \{1,\ldots,T\}} ||X_t||_2$, $U_{max} = \max_{t \in \{1,\ldots,T\}} ||U_t||_2$, and $H_2(C) = \sum_{t=1}^{T} ||U_t - U_{t+1}||_2$. The bound for ALMA when $p = 2$ is

$$O\left(\frac{X_{max}^2 U_{max} H_2(C)}{\delta^2} + \frac{N}{\delta}\right) .$$

[4] More information on ALMA can be found in Section 2.3.5.

Tracking ALMA$(p, \hat{\delta}, \eta)$

Parameters

$\hat{\delta} \geq 0$ controls the algorithm margin.

$\eta > 0$.

$p \geq 2$ and $q = \frac{p}{p-1}$ controls the norms.

Initialization

$t \leftarrow 1$ is the current trial.

$\forall i \in \{1, \ldots, n\}$ $w_{i,t} = 0$ are the algorithm weights.

Trials

Instance: $\mathbf{x}_t \in [0,1]^n$.

Prediction: If $\mathbf{w_t} \cdot \mathbf{x_t} \geq 0$ predict $\hat{y}_t = 1$ else predict $\hat{y}_t = -1$.

Update: Let $y_t \in \{-1, 1\}$ be the correct label.

If $y_t(\mathbf{w_t} \cdot \mathbf{x_t}) \leq \hat{\delta}$.

Let $f(w_i, \mathbf{w}) = sign(w_i)|w_i|^{q-1}/\|\mathbf{w}\|_q^{q-2}$.

Let $f^{-1}(z_i, \mathbf{z}) = sign(z_i)|z_i|^{p-1}/\|\mathbf{z}\|_p^{p-2}$.

$\forall i \in \{1, \ldots, n\}$ $z_{i,t} = f(\mathbf{w_t}, w_{i,t}) + \eta y_t x_{i,t}$.

$\forall i \in \{1, \ldots, n\}$ $w'_{i,t} = f^{-1}(\mathbf{z_t}, z_{i,t})$.

$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = w'_{i,t} / \max(1, \|\mathbf{w'_t}\|_q)$.

Else

$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = w_{i,t}$.

$t \leftarrow t + 1$.

Figure 6.4: Pseudo-code for Tracking ALMA.

Comparing against the Winnow bound, this bound might be better for sparse instances especially when the target concept has negative weights. This is because λ needs to be set to n when complemented attributes are used in Tracking Unnormalized Winnow. For these types of problems the X_{max}^2 term in ALMA might be smaller than the $\ln(vn/\delta)$ term in Winnow. It may also do better for large concept shifts since the $H_2(C)$ function involves the 2-norm instead of the 1-norm. However, these advantage are likely offset by the extra term U_{max} if the maximum concept size is large.

A current disadvantage of the Tracking Unnormalized Winnow algorithm is computational cost. As explained in Appendix D, linear-threshold algorithms can be implemented in a form that accepts sparse instances. These instances only encode the attributes that are non-zero. Letting m_t be the number of non-zero attributes on trial t, all linear-threshold algorithms in this thesis can be implemented to perform predictions in $O(m_t)$. In addition, we show that many linear-threshold algorithms can perform updates in $O(m_t)$. This includes ALMA, and because the tracking version of ALMA is

just a simplification, this also includes Tracking ALMA. Unfortunately, we currently do not have an implementation of Tracking Unnormalized Winnow that performs efficient updates on sparse instances. A straightforward implementation of updates takes $O(n)$ time with Tracking Unnormalized Winnow.

6.6 Summary

In this chapter, we give a proof for a tracking version of Unnormalized Winnow that shows that the bounds for learning shifting linear-threshold functions have many of the same advantages that the traditional Winnow algorithm has on fixed concepts. These benefits include a weak dependence on the number of irrelevant attributes, inexpensive run-time, and robust behavior against noise. We also show how the performance of this algorithm does not depend on the number of attributes and instead depends on $\max_{t \in \{1,\ldots,T\}} \ln(\|X_t\|_1)$. This is similar to the infinite attribute model but has advantages when dealing with real world constraints such as noise.

To get Winnow to track concepts, we use the standard technique of setting a minimum value for the weights. However, given our proof technique, this gives a mistake bound that is allowed to grow arbitrarily large when $\zeta = \min\{x_{i,t} \mid x_{i,t} > 0\}$ approaches zero. One solution is to transform the instances such that ζ is not allowed to get arbitrarily small. By making the appropriate assumptions, it is possible to allow the effects of these small shifts in the value of attributes to be characterized in the δ parameter of the learning problem. Such a transformation allows us to improve the worst-case mistake bounds.

Chapter 7

Experiments with Shifting Distributions

In this chapter, we give the results of experiments with linear-threshold algorithms on concept tracking problems. Recall that concept tracking allows the target function to change during the on-line learning trials. Our experiments include the tracking algorithms from Chapter 6 and, for comparison purposes, the fixed concept algorithms from Chapter 2. To improve the performance on tracking problems, we also perform experiments using the instance recycling from Chapter 4 and a modification of the voting technique from Chapter 3.

Our model for instance generation is a shifting distribution. For each trial, an instance is generated by sampling from a distribution; however, this distribution is allowed to change. In particular, the label probabilities can change causing a shift in the optimal target function. This is a realistic and flexible instance generation model. While it can represent something as difficult as an adversary, the difficulty is limited by how much the distribution is allowed to change.

For example, when predicting the weather, we do not think an adversary is attempting to maximize the mistakes. However, it is also unrealistic to think that weather prediction should be based on a fixed target function. There may be changes in the target function caused by variables not directly accounted for in our model, for example, El Niño or global warming. These effects can cause a slow or infrequent change in the target function and make a shifting distribution a realistic model for instance generation.[1]

Our first set of experiments deal with just the basic linear-threshold algorithms. Our goal is to compare the performance of Tracking Unnormalized Winnow and Tracking

[1]We give a more formal definition of the shifting distribution model in the next chapter.

ALMA. We test a range of parameters for both algorithms. We also compare the performance of the tracking algorithms to the fixed concept algorithms of Chapter 2. As we will see, the fixed concept algorithms do surprisingly well on our tracking problems.

Next, we perform experiments with the voting techniques from Chapter 3 and the instance recycling technique from Chapter 4. These techniques are designed to improve the performance of adversarial algorithms when instances are generated by a fixed distribution. In this chapter, we show they can also improve performance when the instances are generated by a shifting distribution.

The instance recycling technique is identical to what was described in Chapter 4, but the main voting technique from Chapter 3 is primarily designed for fixed distributions. Therefore, we use the Bagging technique described in Section 3.4.5 and a new voting technique based on the VR-Combine algorithm from Section 3.3.7. The new voting technique creates an instance based on the output of a set of basic algorithms. Therefore if there are v basic algorithms then the new instance will have v attributes. This information and the original label are input into the VR-Combine algorithm. The VR-Combine algorithm attempts to minimize the number of mistakes by finding a hypothesis that combines the predictions of these basic algorithms.

The remainder of the chapter is organized as follows. Section 7.1 gives information on the data sets and our experimental methodology. Section 7.2 gives experiments with the basic linear-threshold learning algorithms. In Section 7.3, we explain the voting techniques and give the results of experiments using these techniques. In Section 7.4, we explain why instance recycling works with shifting target concepts and give the results of experiments with recycling. Finally, in Section 7.5, we give a rough cost analysis of the voting and recycling techniques along with experiments combining the two techniques.

7.1 Experimental Framework

Our primary algorithm for experimentation is the tracking version of Unnormalized Winnow. Tracking Unnormalized Winnow has a number of parameters and options.

First, for every attribute x_i, we add a complemented attribute $1-x_i$. This increases the number of attributes to $2n$ and makes $\|X_t\|_1 = n$ for all trials. This forces $\lambda \geq n$ and ruins the theoretical advantage of Tracking Unnormalized Winnow on problems with a small $\|X_t\|_1$. However, it allows Unnormalized Winnow to represent negative weights. More information on Tracking Unnormalized Winnow can be found in Chapter 6.

Based on Theorem 6.8, when $\lambda = n$ we should set $\sigma = 1/(2n)$. Of course, this is based on the assumption that the instances are generated by an adversary and that our proof is tight. However, experiments on artificial data suggest $\sigma = 1/(2n)$ is a reasonable choice when $\lambda = n$. In addition, $\sigma = 1/(2n)$ matches the value used for the experiments in Chapter 3 with Unnormalized Winnow. Therefore, we can compare the normal versus tracking algorithms based on the effect of the ϵ parameter.

For our experiments, we use ϵ values from the set $\{0.00005/n, 0.005/n, 0.05/n, 0.5/n\}$ and α values from the set $\{1.05, 1.2, 1.4, 1.7, 2.0\}$. These are the same values for α as used in previous chapters. We selected these parameter values based on experiments with artificial data. Using all combinations of these parameters gives 20 algorithms. While it could be beneficial to use a wider range of values, we limit the number of algorithms so that running all the algorithms is still efficient. Running all the algorithms allows one to use an algorithm such as WMA to make a number of mistakes close to the best of the 20 algorithms. See Section 2.5 for more details.

Another option for Tracking Unnormalized Winnow is the instance transformation as explained in Section 6.4.2. This transformation shifts the value of attributes that have a value close to zero. We do not use this instance transformation in our experiments since the technique is primarily intended for instances generated by an adversary. The sequence of instances needed by the adversary to cause problems for the algorithm is complex and unlikely to be generated by a distribution.[2]

In addition to Tracking Unnormalized Winnow, we also experiment with the complemented form of this algorithm, Tracking Complemented Unnormalized Winnow. More details on this algorithm can be found in Section 6.3. We use the same parameter values

[2]See Appendix E for more details.

and options as Tracking Unnormalized Winnow. For brevity, we abbreviate the Tracking Unnormalized Winnow as TUWin(α, σ) and Tracking Complemented Unnormalized Winnow as TCUWin(α, σ).

The tracking ALMA algorithm given in Figure 6.4 has three parameters; however, we only use the p parameter in our experiments. We start with the minimum $p = 2$ value and linearly work our way up to 6.5 in steps of 0.25. According to theoretical results, larger p values could be beneficial[Gen03]; however, in preliminary experiments we found that p values over 6.5 always increased the number of mistakes. Just in case, we include the theoretically motivated value of $p = \ln n$ as our final test case.

The $\hat{\delta}$ parameter controls the margin of the algorithm. As explained in Section 2.3.5, we do not perform experiments with the algorithm margin because our artificial data experiments have shown the parameter is difficult to set properly for various algorithms. The η parameter should have no effect on the algorithm though we do find using a small η value is helpful in controlling numerical stability issues. We use the value $\eta = 0.0001$ for all our experiments. In our experiments, the abbreviation for Tracking ALMA is TALMA(p).

7.1.1 Data Sets

Since machine learning research has primarily focused on learning fixed concepts it is difficult to find many tracking data sets. Instead, we generate our own tracking data sets using the fixed distribution data sets from Chapter 3. These data sets primarily come from the UCI repository [DNMml].

Recall that the data sets we use are multi-class and often have many labels. Again, we convert these data sets into binary concepts, but instead of just a single concept we force the concept to change every 2000 trials. To generate tracking problems, we only use data sets that have at least five labels. We use the labels to create a tracking problem composed of five different concepts. Each concept is based on a single label. All instances with this label are instances of the concept; the remaining instances are not in the concept. We sample 2000 instances from the data set and give them each a new binary label according to the current concept. After 2000 instances, we switch to

Name	attributes	shifting label order
covtype	144	1, 2, 3, 4, 5
isolet	617	1, 2, 3, 4, 5
letter	160	A, B, C, D, E
nursery	26	not_recom, recommend, very_recom, priority, spec_prior
optdigits	640	0, 1, 2, 3, 4
page-blocks	100	1, 2, 3, 4, 5
pendigits	160	0, 1, 2, 3, 4
sat	360	1, 2, 3, 4, 5
segmentation	190	brickface, sky, foliage, cement, window
shuttle	90	1, 2, 3, 4, 5
yeast	80	CYT, ERL, EXC, ME1, ME2
mfeat	6490	0, 1, 2, 3, 4
news	32889	alt.atheism, comp.graphics, comp.os.ms-windows.misc, comp.sys.ibm.pc.hardware, comp.sys.mac.hardware
reuters	18307	acq, crude, dlr, earn, grain
web	22123	course, department, faculty, other, project

Table 7.1: A description of the shifting data sets used in our experiments.

the next concept. We repeat this procedure changing the concept every 2000 instances until we reach 10000 trials. In Table 7.1, we give the shifting concepts used in this chapter along with the order of labels used to generate the data. More information on these data sets can be found in Section 3.4.1.

7.1.2 Statistics

For all the experiments in this chapter, we report results based on an average of 50 bootstrap samples [HMM$^+$03]. Each bootstrap sample is generated by sampling independently with replacement from all the instances in a problem. We sample until we have a sequence of 2000 instances from a concept. We repeat this for all five concepts. The instances are labeled with a binary label as explained in the previous section. This gives 50 bootstrap samples of the tracking problem. For statistical significance, we give a confidence interval for a bootstrap sample based on a t-test with a 95% confidence interval [DeG86].

Ideally, our result would include a graph that plots the average number of mistakes at each trial. In Figure 7.1, we give one such plot for the mfeat data set using the

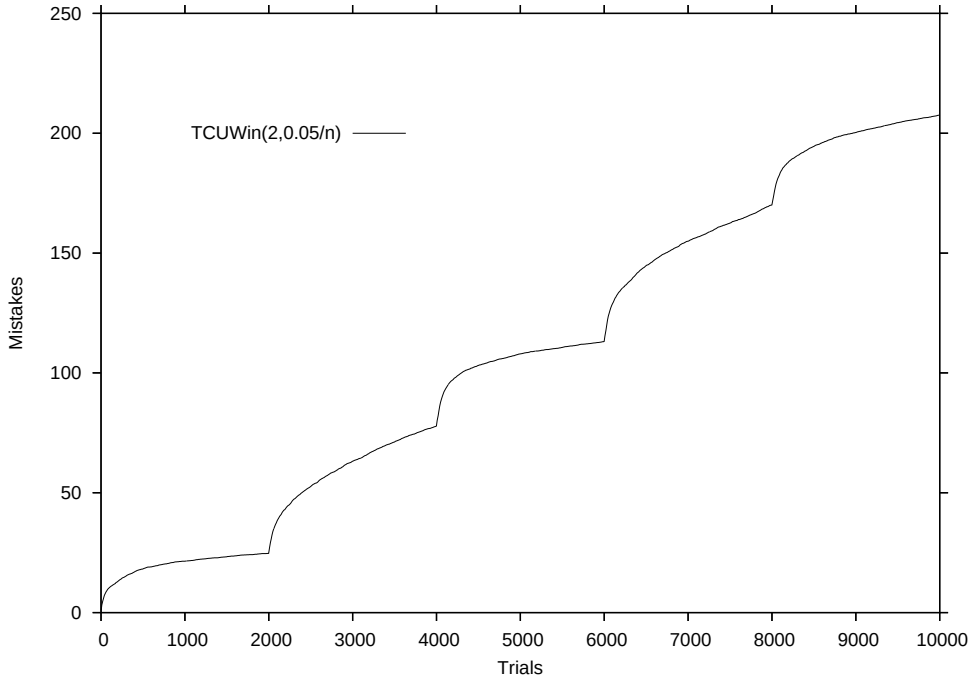

Figure 7.1: Mistake curve for tracking version of mfeat concept.

Tracking Complemented Unnormalized Winnow algorithm with $\alpha = 2.0$, $\epsilon = 0.05/n$, and $\sigma = 1/(2n)$. The plot is fairly smooth since it is an average over the 50 bootstrap samples. The concept shifts can be seen every 2000 trials.

Notice how the algorithm has a greater error rate at the beginning of a concept shift and then settles towards a fixed slope line. This is typical for our average plots when dealing with distributions generating the data. At the beginning there is a learning phase. This is followed by a phase where the algorithm does not improve its average error rate. This transition is gradual, and when learning is difficult, such as when there are many attributes, it can require more trials then available to get to the fixed error phase. Also notice how the learning curves from the 5 concepts are different. The algorithm has more difficulty with certain concepts.

Unfortunately, we have too many algorithms and problems to display the results with mistake plots. Instead, we report the total number of mistakes over all 15 tracking problems and/or we report the total number of mistakes for each tracking problem. In both cases, we use the notation $\hat{\mathrm{M}}(B)$ for the confidence interval on the expected

number of mistakes made by algorithm B. When giving the results for individual tracking problems, because of the number of algorithms, we only give the results for the best algorithm from a set of algorithms. For example, we might consider all Tracking ALMA algorithms and report the parameter and mistake total that corresponds to the algorithm with the fewest mistakes.

When dealing with a set of algorithms, reporting only the results for the algorithm that makes fewest mistakes can bias the results. To be safe, we use a confidence interval of 95.5% when reporting the results on the best algorithm from a set. Because each set contains at most 20 algorithms, the union bound gives a confidence of at least 95% that every interval from the set contains the expected value. Therefore, the algorithm with the fewest mistakes has a confidence interval of at least 95%.

7.2 Basic Tracking Algorithms

Our first results are in Table 7.2. On each data set, we give the mistakes for three algorithms: one from the set of Tracking ALMA algorithms, one from the set of Tracking Unnormalized Winnow algorithms, and one from the set of Tracking Complemented Unnormalized Winnow algorithms.

There are several facts that stand out in the table. First, the uncomplemented form of Tracking Unnormalized Winnow almost always does worse than the complemented form. This parallels our results on fixed concepts from Chapter 3 and seems to be a property of these learning tasks. Second, neither the Tracking Complemented Unnormalized Winnow or Tracking ALMA algorithm dominate on the problems. Each does best on several problems. In particular, the Tracking ALMA algorithm does best on the three text problems: news, reuters, and web. These problems have sparse instances and the Tracking ALMA algorithm exploits this with a small p value. Last, the Tracking Complemented Unnormalized Winnow algorithm does the best on 10 of the data sets. For 8 of these problems, the best multiplier is at the maximum value tested, and for all of the problems the best ϵ is at least $0.1/2n$. This suggests there might be more room for improvement with even larger α and ϵ values.

Concept Name	$\hat{M}$(Best TUWin)	$\hat{M}$(Best TCUWin)	$\hat{M}$(Best TALMA)
covtype	TUWin($1.4,\frac{.1}{2n}$) 1662 ± 16	TCUWin($2.0,\frac{.1}{2n}$) $\mathbf{1608} \pm 15$	ALMA(3.25) 1646 ± 16
isolet	TUWin($1.05,\frac{1}{2n}$) 516 ± 9	TCUWin($2.0,\frac{.1}{2n}$) $\mathbf{401} \pm 7$	ALMA(5.0) 517 ± 11
letter	TUWin($1.4,\frac{.1}{2n}$) 534 ± 10	TCUWin($2.0,\frac{1}{2n}$) $\mathbf{444} \pm 9$	ALMA(2.5) 524 ± 11
nursery	TUWin($1.05,\frac{.0001}{2n}$) 796 ± 14	TCUWin($1.4,\frac{.1}{2n}$) 755 ± 15	ALMA(3.0) $\mathbf{705} \pm 14$
optdigits	TUWin($1.2,\frac{1}{2n}$) 472 ± 7	TCUWin($2.0,\frac{.1}{2n}$) $\mathbf{358} \pm 6$	ALMA(2.5) 425 ± 7
page-blocks	TUWin($1.7,\frac{.1}{2n}$) 515 ± 14	TCUWin($2.0,\frac{.1}{2n}$) 484 ± 13	ALMA(2.0) $\mathbf{477} \pm 13$
pendigits	TUWin($1.05,\frac{1}{2n}$) 510 ± 8	TCUWin($2.0,\frac{.1}{2n}$) $\mathbf{384} \pm 8$	ALMA(2.25) 510 ± 9
sat	TUWin($1.05,\frac{1}{2n}$) 843 ± 13	TCUWin($2.0,\frac{.1}{2n}$) $\mathbf{732} \pm 13$	ALMA(2.25) 814 ± 12
segmentation	TUWin($1.4,\frac{.1}{2n}$) 824 ± 16	TCUWin($2.0,\frac{.1}{2n}$) $\mathbf{614} \pm 12$	ALMA(3.25) 652 ± 13
shuttle	TUWin($1.4,\frac{.1}{2n}$) 554 ± 13	TCUWin($1.7,\frac{.1}{2n}$) $\mathbf{488} \pm 13$	ALMA(2.75) 609 ± 14
mfeat	TUWin($1.7,\frac{.1}{2n}$) 299 ± 7	TCUWin($2.0,\frac{.1}{2n}$) $\mathbf{208} \pm 4$	ALMA(5.75) 279 ± 6
yeast	TUWin($1.2,\frac{.1}{2n}$) 966 ± 12	TCUWin($2.0,\frac{1}{2n}$) $\mathbf{916} \pm 13$	ALMA(2.75) 949 ± 12
news	TUWin($2.0,\frac{1}{2n}$) 804 ± 12	TCUWin($2.0,\frac{1}{2n}$) 792 ± 11	ALMA(2.75) $\mathbf{662} \pm 10$
reuters	TUWin($2.0,\frac{1}{2n}$) 693 ± 11	TCUWin($2.0,\frac{1}{2n}$) 733 ± 12	ALMA(2.75) $\mathbf{559} \pm 10$
web	TUWin($1.7,\frac{1}{2n}$) 1133 ± 11	TCUWin($1.7,\frac{1}{2n}$) 1111 ± 11	ALMA(2.75) $\mathbf{1015} \pm 13$

Table 7.2: Average number of mistakes for best algorithm on 15 tracking problems.

In general, the Tracking Unnormalized Winnow algorithm has a fairly coarse parameter spread in our experiments. A finer and larger grid of parameters yields substantial improvements for TUWin but not much improvement in Tracking ALMA. This is most likely because of the single parameter used for Tracking ALMA which we already test with 20 values. To keep the number of TUWin algorithms similar, we explore less of their parameter space. We place this limit on the number of algorithms to keep the computational cost of concurrently running all the algorithms reasonable.

A natural choice for comparison with the tracking algorithms is their non-tracking predecessors. In Table 7.3, we give the results for the algorithms from Chapter 3 on our tracking experiments. We use the same parameters for all of these algorithms as in Section 3.4 except for the ALMA algorithm. For ALMA, we start with the minimum $p = 2$ value and linearly work our way up to 6.5 in steps of 0.25. These are the same values previously used for Tracking ALMA. In the table, we only report the mistakes for a single algorithm out of various sets of algorithms. Our sets consist of the Balanced algorithms, the Unnormalized Winnow algorithms, the Normalized Winnow algorithms, Perceptron, and the ALMA algorithms. We report the mistake count for the best algorithm from the set. All of these algorithms do not currently have proofs for tracking based mistake bounds.

In the comparison between the tracking and normal version of Complemented Unnormalized Winnow, the tracking version always makes fewer mistakes. This decrease in mistakes can vary greatly. For a few problems the decrease is not significant, but for several the decrease is over 10%. However, one might be suprised that the performance gain of the tracking version is not larger.

In principle, the reason the normal versions of Winnow have difficulty with tracking is that the algorithm weights get too small. If the concept shifts and an attribute with a small weight becomes relevant, the algorithm might need to make a large number of mistakes to increase the value. For our experiments, this issue does not cause a large increase in the number of mistakes. This is most likely a result of the instances being generated by a distribution. An adversary might be needed to create a large difference between the tracking and non-tracking versions of Unnormalized Winnow.

Concept	$\hat{M}$(Bal)	$\hat{M}$(UWin)	$\hat{M}$(NWin)	$\hat{M}$(Per)	$\hat{M}$(ALMA)
	Bal(1.4)	UCWin(2.0)	NWin(1.7,.7)	Per	ALMA(3.75)
covtype	1647 ± 17	1640 ± 17	$\mathbf{1610} \pm 18$	1693 ± 15	1634 ± 16
	Bal(1.7)	UCWin(2.0)	NWin(2.0,.7)	Per	ALMA(4.5)
isolet	517 ± 9	420 ± 8	$\mathbf{346} \pm 9$	563 ± 10	543 ± 12
	Bal(1.2)	UCWin(2.0)	NWin(1.4,.7)	Per	ALMA(3.0)
letter	524 ± 11	478 ± 10	$\mathbf{463} \pm 10$	536 ± 12	524 ± 11
	Bal(1.4)	UCWin(1.4)	NWin(1.2,.7)	Per	ALMA(2.5)
nursery	703 ± 15	759 ± 15	691 ± 14	720 ± 12	$\mathbf{661} \pm 15$
	Bal(1.2)	UCWin(1.7)	NWin(1.4,.7)	Per	ALMA(2.0)
optdigits	422 ± 7	410 ± 9	$\mathbf{329} \pm 7$	433 ± 7	464 ± 8
	Bal(1.05)	UCWin(2.0)	NWin(1.7,.7)	Per	ALMA(2.0)
page-blocks	476 ± 13	502 ± 14	$\mathbf{476} \pm 14$	$\mathbf{476} \pm 13$	$\mathbf{476} \pm 12$
	Bal(1.2)	UCWin(1.7)	NWin(1.2,.7)	Per	ALMA(2.0)
pendigits	505 ± 8	448 ± 10	$\mathbf{373} \pm 7$	518 ± 7	542 ± 10
	Bal(1.2)	UCWin(1.7)	NWin(1.4,.7)	Per	ALMA(2.25)
sat	814 ± 13	767 ± 13	$\mathbf{695} \pm 12$	814 ± 13	804 ± 13
	Bal(1.4)	UCWin(2.0)	NWin(1.4,.7)	Per	ALMA(2.75)
segmentation	667 ± 13	677 ± 13	$\mathbf{644} \pm 12$	712 ± 14	702 ± 14
	Bal(1.4)	UCWin(1.7)	NWin(1.7,.7)	Per	ALMA(3.25)
shuttle	612 ± 14	530 ± 15	$\mathbf{490} \pm 12$	701 ± 14	596 ± 14
	Bal(1.4)	UCWin(2.0)	NWin(1.7,.7)	Per	ALMA(5.0)
mfeat	300 ± 6	258 ± 6	$\mathbf{197} \pm 6$	338 ± 9	346 ± 9
	Bal(1.2)	UCWin(2.0)	NWin(1.05,.7)	Per	ALMA(2.25)
yeast	953 ± 12	936 ± 12	$\mathbf{911} \pm 12$	952 ± 14	925 ± 14
	Bal(1.4)	UCWin(2.0)	NWin(2.0,.5)	Per	ALMA(2.0)
news	$\mathbf{659} \pm 10$	821 ± 12	805 ± 13	677 ± 11	699 ± 11
	Bal(1.2)	UWin(2.0)	NWin(2.0,.3)	Per	ALMA(2.0)
reuters	$\mathbf{556} \pm 9$	745 ± 12	717 ± 12	567 ± 9	573 ± 9
	Bal(1.2)	UCWin(1.7)	NWin(1.7,.5)	Per	ALMA(2.25)
web	$\mathbf{1011} \pm 11$	1262 ± 12	1187 ± 11	1068 ± 11	1137 ± 13

Table 7.3: Average number of mistakes for best algorithm on 15 tracking problems.

The difference between Tracking ALMA and ALMA is less clear; it is close to an even split between the problems. The normal version of ALMA attempts to dynamically adjust its η parameter to improve performance.[3] This technique could fail with a tracking problem where the concept is changing. A similar problem occurs with how the normal version of ALMA handles noisy instances. The dynamic adjustment of its η parameter causes certain types of noise to have a disproportionate effect on the algorithm.[4] Either of these effects could explain the decrease in performance over the tracking version of ALMA. Of course, the dynamic η parameter must also explain why the normal ALMA algorithm just as often makes fewer mistakes than Tracking ALMA. However, the improvement is often small.

When considering all the algorithms, even though Normalized Winnow and Balanced Winnow do not currently have proofs for tracking problems, they give the best performance on several problems. The Balanced Winnow algorithm performs best on the three text based problems. These problems have a large number of attributes with only a small fraction of these attributes having a non-zero value. The Normalized Winnow algorithm does well on most of the other problems. It is the best of all the algorithms on seven of the tracking problems. These results are not surprising given our comparison of the algorithms in Section 2.4.2 and the previous evidence that small algorithm weights do not have a large effect on performance. However, while the minimum weight modification in Tracking Unnormalized Winnow may not have a large impact, it is clear that it does improve performance. It is possible that a similar modification could benefit the other Winnow algorithms on tracking problems.

7.3 Voting

The main voting procedure in Chapter 3 votes based on the predictions of several hypotheses generated during the trials. These hypotheses can be spread out over all the previous trials. This type of procedure cannot work when dealing with a changing

[3] See Section 2.3.5 for more details.

[4] See Section 2.4.1 for more details.

target function. Older hypotheses might have been accurate in previous trials, but the target function can be completely different in the current trial.

In order to improve the performance with a shifting target function, we use voting procedures that only use the current hypotheses of any basic algorithms. We consider two techniques. The first is the bagging technique of [OR01] first described in Section 3.4.5.

7.3.1 Bagging

The on-line bagging technique is based on running several versions of a basic algorithm where each algorithm gets a slightly different stream of instances. The different streams are based on randomly repeating instances in the original sequence. For each instance in the original sequence, a number k is generated based on a Poisson distribution with mean 1; this is the number of times the instance is repeated. There are a total of h algorithms running the same basic algorithm but each gets a different stream of instances based on sampling. The bagging algorithm predicts based on the majority vote of these h hypotheses.

We already described some of the problems with on-line bagging in Section 3.4.5. However, the main advantage when dealing with tracking is that the bagging procedure uses the current hypotheses from the h basic algorithms. Since each of these hypotheses has seen a slightly different stream of instances, the hope is that the majority of these hypotheses will have a higher prediction accuracy than any single hypotheses in the group. This is the main motivation of the batch bagging technique described in [Bre96].

An additional problem that we have not yet discussed with on-line bagging is the cost. Since we found the other voting techniques in Chapter 3 to be superior, we did not discuss the cost, which can be excessive. Because the algorithm is essentially running h times, the cost increases by a factor of h. While this may seem similar to the increases seen with the voting techniques of Chapter 3, in that case the h factor was only significant with the rather simple prediction procedure used by the voting. In this case, the whole algorithm increases by a factor of h. As we will see, in many cases this extra cost is difficult to justify.

7.3.2 VR-Combine

The VR-Combine technique is an application of the results from Chapter 5. Typically, we do not know which algorithm will give the fewest mistakes for a particular tracking problem. Therefore, we run a set of algorithms and give them all the same stream of instances. Assume the set has v algorithms. At this point, there are standard techniques to make a number of mistakes close to the fewest mistakes made by any algorithm in the set.[5] However, another alternative is to do something similar to bagging and combine the predictions of the v algorithms in an attempt to make fewer mistakes than any single algorithm.

The naive bagging approach would be to use a simple majority vote based on the predictions of the v algorithms. However, there is no guarantee that the majority vote will maximize the accuracy particularly since this naive technique violates the typical assumptions of bagging. A more sophisticated approach is to make the predictions using a weighted vote of the algorithm predictions. This is a convenient way to formulate the problem because the purpose of a linear-threshold algorithm is to learn such weights. We have covered several possible linear-threshold algorithms in this dissertation. However, instead of just using a linear-threshold algorithm, we can apply our previous results and use the combined voting and recycling algorithms of Chapter 5. The best of these algorithms is VR-Combine. Therefore we use the VR-Combine algorithm with the basic algorithms of Section 3.4 and the default parameters described in Section 5.2.1.

While justification of voting in VR-Combine was based on the assumption that the instances come from a fixed distribution, the essential idea of the voting techniques in Chapter 3 is to combine a set of different but relatively accurate hypotheses to improve performance. Even with shifting concepts, this motivation often remains true. Remember that VR-Combine is learning a way to combine the predictions of the current hypothesis of each algorithm that is being used on the tracking problem. As the concept shifts, this combination will most likely fall in accuracy, but the accuracy might still be high relative to the other possible combinations. In a sense, the shifting concept

[5] See Section 2.5 for more details on this approach.

causes extra noise in the VR-Combine instances, but this extra noise may not effect the optimal hypothesis for VR-Combine. However, even if voting does not improve performance, the VR-Combine algorithm is designed, at a minimum, to do as well as its best basic algorithm.[6]

Fortunately, unlike on-line bagging, the cost of this VR-Combine technique is relatively small. Instead of increasing the cost of the entire algorithm by some multiplicative factor we only add an extra term. In Section 7.5, we give more details on this cost.

While this application of the VR-Combine algorithm seems complicated when one delves into the details, the best way to view the VR-Combine algorithm is as a black box. This black box works according to the on-line model. It takes in instances formed from the predictions of the algorithms used for the tracking problem and returns a label. It receives the feedback label from the tracking problem and uses this to update its hypothesis.

7.3.3 Voting Experiments

We perform the same experiments as described in Section 7.1.1. However, along with giving results for each individual tracking problem, we also give a scatter plot to show the improvements of bagging, and we give tables that sum the number of mistakes over all fifteen tracking problems. The tables are useful to show the average effect of bagging for each algorithm.

In Figure 7.2, we give a scatter plot that contains a point for each algorithm/concept pair. We include the same set of tracking and non-tracking algorithms as used in the previous sections. This gives a total of 1395 points. The y coordinate of each points corresponds to the final number of mistakes made by a basic algorithm on the particular concept; the x coordinate corresponds to the final number of mistakes made by the bagging version of the basic algorithm. As can be seen, the majority of points are above the line $y = x$ showing that bagging reduces the number of mistakes on most data sets. Only 22 points show a basic algorithm doing better than bagging on a

[6]See Section 3.3.2 for details.

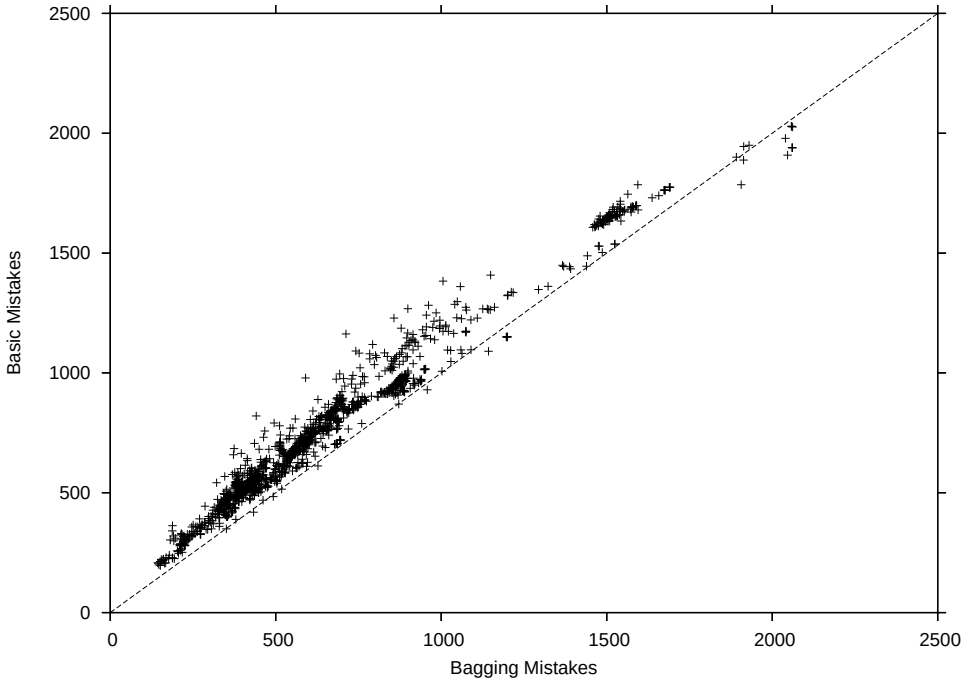

Figure 7.2: Scatter plot comparing basic algorithm to bagging algorithm.

concept.

In Table 7.4, we give the results of the tracking algorithms covered in Chapter 6. We use the same parameters and notation as described in Section 7.1. In order to make the table concise, we have left out the results of the Tracking Unnormalized Winnow algorithm and only report results for ALMA and Tracking Complemented Unnormalized Winnow. As we saw in Section 7.2, the ALMA and complemented Winnow algorithms give the best performance for our data sets. Therefore, we leave the results for Tracking Unnormalized Winnow out of all the tables in this chapter.

For each algorithm in Table 7.4, we give the results of the basic algorithm and the algorithm with bagging. We use 30 voting hypotheses for all our experiments with bagging. At the bottom of the table, we give the results for the VR-Combine algorithm. In the first column, VR-Combine uses the basic algorithms and has 60 attributes because the predictions of Tracking Unnormalized Winnow is included in the input. In the second column, the VR-Combine algorithm has 120 inputs because we include the basic algorithms and the predictions of the bagging algorithms. To

Name	$\hat{\mathrm{M}}$(Name)	$\hat{\mathrm{M}}$(B-Name)
TALMA(2.0)	10778 ± 27	$\mathbf{9125} \pm 27$
TALMA(2.25)	10566 ± 26	$\mathbf{8896} \pm 29$
TALMA(2.5)	10478 ± 27	$\mathbf{8771} \pm 28$
TALMA(2.75)	10432 ± 27	$\mathbf{8710} \pm 27$
TALMA(3.0)	10443 ± 31	$\mathbf{8678} \pm 28$
TALMA(3.25)	10450 ± 34	$\mathbf{8658} \pm 26$
TALMA(3.5)	10485 ± 32	$\mathbf{8667} \pm 28$
TALMA(3.75)	10518 ± 35	$\mathbf{8669} \pm 25$
TALMA(4.0)	10561 ± 32	$\mathbf{8683} \pm 30$
TALMA(4.25)	10606 ± 35	$\mathbf{8711} \pm 28$
TALMA(4.5)	10674 ± 33	$\mathbf{8722} \pm 29$
TALMA(4.75)	10719 ± 30	$\mathbf{8757} \pm 27$
TALMA(5.0)	10773 ± 29	$\mathbf{8778} \pm 29$
TALMA(5.25)	10835 ± 33	$\mathbf{8794} \pm 27$
TALMA(5.5)	10882 ± 30	$\mathbf{8843} \pm 28$
TALMA(5.75)	10922 ± 32	$\mathbf{8868} \pm 27$
TALMA(6.0)	10984 ± 33	$\mathbf{8902} \pm 27$
TALMA(6.25)	11038 ± 37	$\mathbf{8926} \pm 30$
TALMA(6.5)	11070 ± 29	$\mathbf{8953} \pm 30$
TALMA($\ln n$)	11109 ± 34	$\mathbf{8940} \pm 28$
TCUWin(1.05,$\frac{.0001}{2n}$)	13582 ± 35	$\mathbf{12662} \pm 33$
TCUWin(1.05,$\frac{.01}{2n}$)	13579 ± 35	$\mathbf{12657} \pm 31$
TCUWin(1.05,$\frac{.1}{2n}$)	13530 ± 34	$\mathbf{12631} \pm 32$
TCUWin(1.05,$\frac{1}{2n}$)	13238 ± 30	$\mathbf{12205} \pm 31$
TCUWin(1.2,$\frac{.0001}{2n}$)	11888 ± 33	$\mathbf{10742} \pm 31$
TCUWin(1.2,$\frac{.01}{2n}$)	11848 ± 33	$\mathbf{10702} \pm 33$
TCUWin(1.2,$\frac{.1}{2n}$)	11739 ± 32	$\mathbf{10608} \pm 28$
TCUWin(1.2,$\frac{1}{2n}$)	11251 ± 30	$\mathbf{9987} \pm 31$
TCUWin(1.4,$\frac{.0001}{2n}$)	11126 ± 34	$\mathbf{9655} \pm 29$
TCUWin(1.4,$\frac{.01}{2n}$)	11040 ± 32	$\mathbf{9575} \pm 29$
TCUWin(1.4,$\frac{.1}{2n}$)	10817 ± 31	$\mathbf{9369} \pm 29$
TCUWin(1.4,$\frac{1}{2n}$)	10534 ± 31	$\mathbf{8954} \pm 29$
TCUWin(1.7,$\frac{.0001}{2n}$)	10768 ± 32	$\mathbf{9127} \pm 28$
TCUWin(1.7,$\frac{.01}{2n}$)	10525 ± 31	$\mathbf{8954} \pm 27$
TCUWin(1.7,$\frac{.1}{2n}$)	10289 ± 31	$\mathbf{8697} \pm 29$
TCUWin(1.7,$\frac{1}{2n}$)	10338 ± 32	$\mathbf{8569} \pm 27$
TCUWin(2.0,$\frac{.0001}{2n}$)	10647 ± 34	$\mathbf{8891} \pm 27$
TCUWin(2.0,$\frac{.01}{2n}$)	10347 ± 34	$\mathbf{8634} \pm 28$
TCUWin(2.0,$\frac{.1}{2n}$)	10132 ± 33	$\mathbf{8406} \pm 25$
TCUWin(2.0,$\frac{1}{2n}$)	10360 ± 33	$\mathbf{8482} \pm 28$
VR-Combine	8020 ± 27	$\mathbf{7511} \pm 26$

Table 7.4: Total mistakes on basic tracking and bagging algorithms from 15 tracking concepts.

help distinguish between these algorithms, we call VR-Combine with just the basic algorithms VR-Combine1, and VR-Combine with the basic and bagging algorithms VR-Combine2.

The VR-Combine algorithm also has its own set of basic algorithms that it uses to generate a hypothesis. These are not necessarily related to the basic algorithms used for the tracking problem. In all cases in this chapter, the VR-Combine algorithms use the basic algorithms and default parameter settings as described in Chapter 5. This includes 33 basic algorithms where each algorithm performs recycling with 100 old instances and each instance can be used for a single update.

In Table 7.5, we give the results for all the fixed concept algorithms of Chapter 2. We use the same parameters and notation as described in Section 3.4. The table follows the conventions described in the previous paragraphs except that the results of all the algorithms are reported. In this case, VR-Combine1 has 33 inputs for the first column, and VR-Combine2 has 66 inputs for the second column.

As can be seen in the tables, bagging consistently improves the performance of the basic algorithm. To a first approximation, the better the basic algorithm the better the bagging algorithm. However, as we saw in Chapter 3, higher multipliers with the Winnow algorithms can give better performance with voting. The same is true with TALMA; the optimal p value for the TALMA basic algorithms tends to be slightly lower than the optimal value for bagging. In general, the advantage of larger parameters could be a result of greater hypotheses diversity. However, the effect is short lived. As the parameters continue to grow larger, the bagging algorithm performance starts to degrade.

By far the best algorithm is VR-Combine2. This algorithm makes over 25% fewer mistakes than the best basic algorithm, and over 10% fewer mistakes than the best bagging algorithm. Even VR-Combine1, using just the basic algorithms, out performs all of the bagging algorithms. This can be partially explained based on the implicit parameter selection of VR-Combine. For each problem, VR-Combine can weight the algorithms differently giving a preference for the best algorithm for a particular problem.

Name	$\hat{\mathrm{M}}$(Name)	$\hat{\mathrm{M}}$(B-Name)
Perceptron	10769 ± 28	$\mathbf{9133} \pm 25$
ALMA(2)	10853 ± 34	$\mathbf{9423} \pm 28$
ALMA($\ln n$)	11545 ± 37	$\mathbf{9610} \pm 31$
Bal(1.05)	10719 ± 27	$\mathbf{9063} \pm 28$
Bal(1.2)	10450 ± 29	$\mathbf{8819} \pm 27$
Bal(1.4)	10485 ± 32	$\mathbf{8712} \pm 27$
Bal(1.7)	10925 ± 32	$\mathbf{8816} \pm 26$
Bal(2.0)	11424 ± 36	$\mathbf{9012} \pm 32$
UWin(1.05)	13172 ± 41	$\mathbf{12109} \pm 33$
UWin(1.2)	12248 ± 33	$\mathbf{10793} \pm 29$
UWin(1.4)	12060 ± 39	$\mathbf{10137} \pm 29$
UWin(1.7)	12599 ± 39	$\mathbf{10142} \pm 30$
UWin(2.0)	13335 ± 39	$\mathbf{10463} \pm 30$
CUWin(1.05)	13581 ± 34	$\mathbf{12649} \pm 29$
CUWin(1.2)	11888 ± 33	$\mathbf{10727} \pm 30$
CUWin(1.4)	11129 ± 33	$\mathbf{9637} \pm 30$
CUWin(1.7)	10813 ± 31	$\mathbf{9129} \pm 26$
CUWin(2.0)	10850 ± 38	$\mathbf{8958} \pm 31$
NWin(1.05,.3)	14179 ± 32	$\mathbf{13680} \pm 30$
NWin(1.2,.3)	11733 ± 30	$\mathbf{10553} \pm 30$
NWin(1.4,.3)	11274 ± 31	$\mathbf{9695} \pm 28$
NWin(1.7,.3)	11498 ± 30	$\mathbf{9407} \pm 28$
NWin(2.0,.3)	11989 ± 33	$\mathbf{9501} \pm 28$
NWin(1.05,.5)	11786 ± 31	$\mathbf{10357} \pm 28$
NWin(1.2,.5)	10916 ± 32	$\mathbf{9365} \pm 28$
NWin(1.4,.5)	10587 ± 31	$\mathbf{8876} \pm 28$
NWin(1.7,.5)	10647 ± 27	$\mathbf{8651} \pm 30$
NWin(2.0,.5)	10967 ± 35	$\mathbf{8690} \pm 30$
NWin(1.05,.7)	12143 ± 36	$\mathbf{11621} \pm 32$
NWin(1.2,.7)	10676 ± 28	$\mathbf{9541} \pm 28$
NWin(1.4,.7)	10238 ± 29	$\mathbf{8830} \pm 30$
NWin(1.7,.7)	10110 ± 34	$\mathbf{8431} \pm 28$
NWin(2.0,.7)	10224 ± 35	$\mathbf{8335} \pm 27$
VR-Combine	7965 ± 26	$\mathbf{7462} \pm 28$

Table 7.5: Total mistakes on basic fixed concept and bagging algorithms from 15 tracking concepts.

To better help understand this behavior, we include tables that give the best algorithm for each of the fifteen individual tracking problems. Table 7.6 selects the best from the tracking algorithms, and Table 7.7 selects the best from the fixed concept algorithms. The first column of each table corresponds to the best basic algorithm. The second column gives the results of VR-Combine1. The third column gives the best bagging algorithm. The last column gives the results of VR-Combine2.

As can be seen in both tables, the best basic algorithm gives the worst performance and VR-Combine2 gives the best. The bagging algorithm and VR-Combine1 alternate for second place. A bagging algorithm wins in 17 of the 30 concepts; however, the best bagging algorithm changes from problem to problem. This can be seen in Table 7.4 and Table 7.5, where VR-Combine1 does better on average than any single bagging algorithm.

7.4 Instance Recycling

In this section, we apply the instance recycling technique from Chapter 4 to concept tracking problems. While instance recycling is designed to work on a range of problems, including adversarial problems, the implicit assumption of instance recycling is that the problem is based on a fixed target function. If the target function is allowed to change then the old instances might correspond to a different target function. These old instances are effectively noisy instances with respect to the current target function and can cause an increase in the number of mistakes.

On the positive side, if the old instances still have sufficient information about the current target function, they can help reduce the number of mistakes. For example, if the target function only changes infrequently then most of the old instances used for updates correspond to the correct target function. Another possibility is that the target function is changing every trial but only by a small amount. When instances are generated by a shifting distribution, these old instances can still be helpful since there is a high probability they correspond to the correct target function.

In both of the above cases, the amount of correct information in the old instances is

Concept	Basic	VR-Combine1	Bagging	VR-Combine2
covtype	TCUWin$(2.0, \frac{1}{2n})$ 1608 ± 15	1526 ± 16	TCUWin$(2.0, \frac{1}{2n})$ 1458 ± 17	$\mathbf{1448} \pm 15$
isolet	TCUWin$(2.0, \frac{1}{2n})$ 401 ± 7	322 ± 7	TCUWin$(2.0, \frac{1}{2n})$ 348 ± 6	$\mathbf{311} \pm 6$
letter	TCUWin$(2.0, \frac{1}{2n})$ 444 ± 9	342 ± 7	TCUWin$(2.0, \frac{1}{2n})$ 332 ± 7	$\mathbf{325} \pm 8$
nursery	ALMA(3.0) 705 ± 14	630 ± 15	ALMA($\ln n$) 588 ± 14	$\mathbf{577} \pm 14$
optdigits	TCUWin$(2.0, \frac{1}{2n})$ 358 ± 6	277 ± 6	TCUWin$(2.0, \frac{1}{2n})$ 250 ± 6	$\mathbf{246} \pm 6$
page-blocks	ALMA(2.0) 477 ± 13	373 ± 11	ALMA(2.0) 378 ± 12	$\mathbf{351} \pm 11$
pendigits	TCUWin$(2.0, \frac{1}{2n})$ 384 ± 8	327 ± 7	TCUWin$(1.7, \frac{1}{2n})$ 296 ± 8	$\mathbf{290} \pm 7$
sat	TCUWin$(2.0, \frac{1}{2n})$ 732 ± 13	616 ± 13	TCUWin$(2.0, \frac{1}{2n})$ 579 ± 10	$\mathbf{560} \pm 11$
segmentation	TCUWin$(2.0, \frac{1}{2n})$ 614 ± 12	526 ± 11	TCUWin$(2.0, \frac{1}{2n})$ 493 ± 10	$\mathbf{474} \pm 9$
shuttle	TCUWin$(1.7, \frac{1}{2n})$ 488 ± 13	419 ± 12	TCUWin$(2.0, \frac{1}{2n})$ 428 ± 12	$\mathbf{392} \pm 11$
mfeat	TCUWin$(2.0, \frac{1}{2n})$ 208 ± 4	152 ± 4	TCUWin$(2.0, \frac{1}{2n})$ 151 ± 4	$\mathbf{137} \pm 4$
yeast	TCUWin$(2.0, \frac{1}{2n})$ 916 ± 13	799 ± 11	TCUWin$(2.0, \frac{1}{2n})$ 828 ± 12	$\mathbf{779} \pm 12$
news	ALMA(2.75) 662 ± 10	471 ± 7	ALMA(5.0) 511 ± 8	$\mathbf{462} \pm 7$
reuters	ALMA(2.75) 559 ± 10	440 ± 9	ALMA(2.5) 433 ± 8	$\mathbf{411} \pm 8$
web	ALMA(2.75) 1015 ± 13	798 ± 9	ALMA(2.75) 846 ± 11	$\mathbf{747} \pm 10$

Table 7.6: Number of mistakes for best tracking algorithm on 15 tracking problems. Includes bagging and VR-Combine techniques. Each VR-Combine column combines algorithms from previous columns algorithms.

Concept	Basic	VR-Combine1	Bagging	VR-Combine2
covtype	NWin(1.7,.7) 1610 ± 18	1514 ± 17	NWin(2.0,.7) 1463 ± 16	**1439** ± 17
isolet	NWin(2.0,.7) 346 ± 9	290 ± 6	NWin(2.0,.7) 294 ± 7	**277** ± 7
letter	NWin(1.4,.7) 463 ± 10	339 ± 7	NWin(2.0,.7) 335 ± 8	**326** ± 7
nursery	ALMA($\ln n$) 670 ± 14	604 ± 15	NWin(1.05,.5) 583 ± 13	**547** ± 14
optdigits	NWin(1.4,.7) 329 ± 7	270 ± 6	NWin(2.0,.7) 246 ± 5	**241** ± 5
page-blocks	ALMA($\ln n$) 476 ± 12	368 ± 11	NWin(2.0,.7) 373 ± 11	**344** ± 11
pendigits	NWin(1.2,.7) 373 ± 7	324 ± 7	NWin(1.4,.7) 303 ± 7	**292** ± 7
sat	NWin(1.4,.7) 695 ± 12	598 ± 10	NWin(2.0,.7) 558 ± 9	**549** ± 10
segmentation	NWin(1.4,.7) 644 ± 12	544 ± 11	NWin(1.7,.7) 523 ± 11	**499** ± 12
shuttle	NWin(1.7,.7) 490 ± 12	420 ± 11	NWin(2.0,.7) 425 ± 12	**389** ± 11
mfeat	NWin(1.7,.7) 197 ± 6	142 ± 4	NWin(2.0,.5) 144 ± 5	**130** ± 4
yeast	NWin(1.05,.7) 911 ± 12	796 ± 11	NWin(2.0,.7) 829 ± 12	**780** ± 10
news	Bal(1.4) 659 ± 10	471 ± 7	Bal(2.0) 514 ± 8	**459** ± 7
reuters	Bal(1.2) 556 ± 9	444 ± 9	Bal(1.4) 431 ± 9	**407** ± 8
web	Bal(1.2) 1011 ± 11	841 ± 12	Bal(1.4) 849 ± 11	**784** ± 10

Table 7.7: Number of mistakes for best fixed concept algorithm on 15 tracking problems. Includes bagging and VR-Combine techniques. Each VR-Combine column combines algorithms from previous columns algorithms.

controlled by the age of the instances used in recycling. If the instances are from older trials, there is a greater chance that these older instances will be based on a target function that is significantly different than the current target function. Therefore, it is important to limit s, the size of the window of old instances, when dealing with tracking problems.

The other parameter for instance recycling is u, the number of times an instance can be used for updates. For tracking problems, we can assume a greater amount of noise in the instances used for recycling. Since noisy instances are more likely to be used for updates when using a mistake-driven algorithm,[7] a large u value could cause problems with the algorithms in this thesis. However, a large u value does allow the algorithm to make more use of the recycled instances without increasing their age. In Section 7.5.2, we perform experiments with the s and u parameters to see how they effect the results on our data sets.

7.4.1 Recycling Experiments

In this section, we give the results of experiments with instance recycling using the same tracking experiments as the previous section. Based on the previous explanation, we set the default parameters for instance recycling to $s = 50$ old instances and $u = 1$ updates per instance.

We start with a scatter plot to compare the basic algorithms to recycling. The scatter plot in Figure 7.3 contains a point for each algorithm/concept pair. We include the same set of tracking and non-tracking algorithms as used in the previous sections. This gives a total of 1395 points. The y coordinate of each points corresponds to the final number of mistakes made by a basic algorithm on the particular concept; the x coordinate corresponds to the final number of mistakes made by the recycled version of the basic algorithm. As can be seen, all but one point is above the $y = x$ line.

Table 7.8 gives the total number of mistakes made by the basic tracking algorithms. The first column has zero instances used for recycling; the second column uses 50

[7] See Section 4.4 for details.

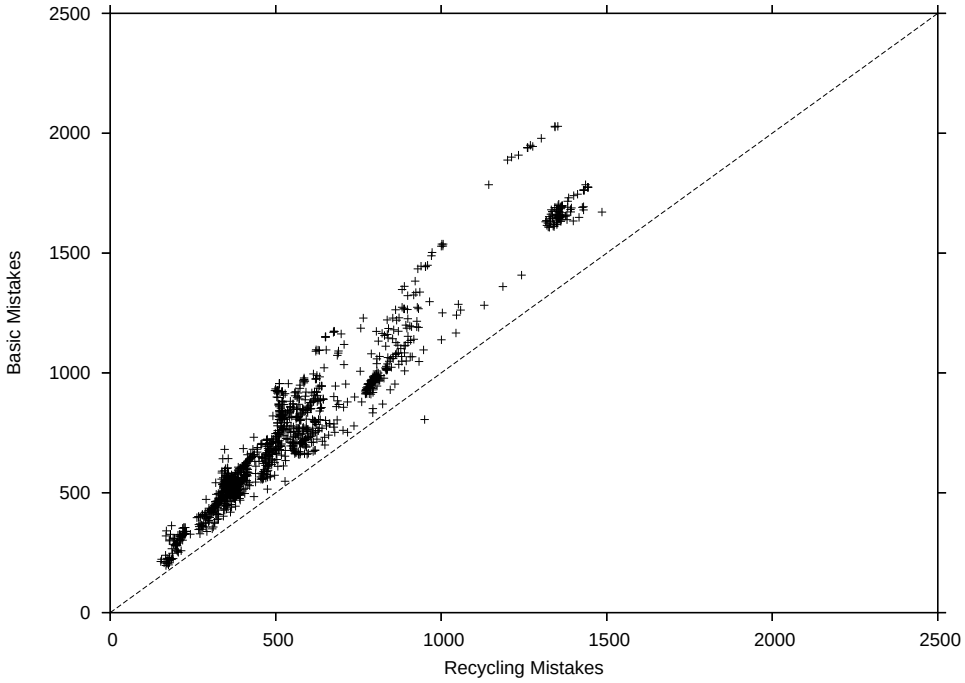

Figure 7.3: Scatter plot comparing basic algorithm to recycling version.

instances for recycling. As can be seen, the recycling gives a consistent and large decrease in the total number of mistakes over the 15 tracking concepts. In fact, the decrease for a particular algorithm is much larger than that seen by bagging. Also, just as with bagging, the optimal algorithm parameter is often slightly higher than the optimal parameter value for the corresponding basic algorithm. The results in Table 7.9 mirror these results but use the fixed concept algorithms of Chapter 2.

At the bottom of the tables, we include the results of using algorithm VR-Combine from Chapter 5 with the basic algorithms as input. In the first column VR-Combine1 uses the basic algorithms with no recycling. The second column uses the algorithms with recycling. We call this algorithm VR-Combine3 to help distinguish it from the other versions of VR-Combine. Again, there is a large improvement based on using the recycling. VR-Combine3, using the recycled tracking algorithms, makes only 6721 mistakes. This is an improvement of almost 10% over the previous best algorithm, VR-Combine2.

Table 7.10 gives the tracking algorithm with the fewest mistakes for each of the 15

Name	$\hat{\mathrm{M}}$(Name)	$\hat{\mathrm{M}}$(R-Name)
TALMA(2.0)	10778 ± 27	$\mathbf{8320} \pm 25$
TALMA(2.25)	10566 ± 26	$\mathbf{8130} \pm 30$
TALMA(2.5)	10478 ± 27	$\mathbf{8052} \pm 25$
TALMA(2.75)	10432 ± 27	$\mathbf{8018} \pm 26$
TALMA(3.0)	10443 ± 31	$\mathbf{8003} \pm 28$
TALMA(3.25)	10450 ± 34	$\mathbf{7990} \pm 28$
TALMA(3.5)	10485 ± 32	$\mathbf{8010} \pm 28$
TALMA(3.75)	10518 ± 35	$\mathbf{8033} \pm 26$
TALMA(4.0)	10561 ± 32	$\mathbf{8061} \pm 27$
TALMA(4.25)	10606 ± 35	$\mathbf{8098} \pm 28$
TALMA(4.5)	10674 ± 33	$\mathbf{8113} \pm 27$
TALMA(4.75)	10719 ± 30	$\mathbf{8158} \pm 28$
TALMA(5.0)	10773 ± 29	$\mathbf{8195} \pm 29$
TALMA(5.25)	10835 ± 33	$\mathbf{8221} \pm 26$
TALMA(5.5)	10882 ± 30	$\mathbf{8253} \pm 26$
TALMA(5.75)	10922 ± 32	$\mathbf{8296} \pm 29$
TALMA(6.0)	10984 ± 33	$\mathbf{8325} \pm 28$
TALMA(6.25)	11038 ± 37	$\mathbf{8355} \pm 29$
TALMA(6.5)	11070 ± 29	$\mathbf{8390} \pm 25$
TALMA($\ln n$)	11109 ± 34	$\mathbf{8394} \pm 26$
TCUWin(1.05,$\frac{.0001}{2n}$)	13582 ± 35	$\mathbf{9468} \pm 25$
TCUWin(1.05,$\frac{.01}{2n}$)	13579 ± 35	$\mathbf{9459} \pm 25$
TCUWin(1.05,$\frac{.1}{2n}$)	13530 ± 34	$\mathbf{9418} \pm 26$
TCUWin(1.05,$\frac{1}{2n}$)	13238 ± 30	$\mathbf{9133} \pm 28$
TCUWin(1.2,$\frac{.0001}{2n}$)	11888 ± 33	$\mathbf{8347} \pm 27$
TCUWin(1.2,$\frac{.01}{2n}$)	11848 ± 33	$\mathbf{8312} \pm 27$
TCUWin(1.2,$\frac{.1}{2n}$)	11739 ± 32	$\mathbf{8219} \pm 24$
TCUWin(1.2,$\frac{1}{2n}$)	11251 ± 30	$\mathbf{7896} \pm 24$
TCUWin(1.4,$\frac{.0001}{2n}$)	11126 ± 34	$\mathbf{8125} \pm 27$
TCUWin(1.4,$\frac{.01}{2n}$)	11040 ± 32	$\mathbf{8020} \pm 25$
TCUWin(1.4,$\frac{.1}{2n}$)	10817 ± 31	$\mathbf{7844} \pm 25$
TCUWin(1.4,$\frac{1}{2n}$)	10534 ± 31	$\mathbf{7685} \pm 25$
TCUWin(1.7,$\frac{.0001}{2n}$)	10768 ± 32	$\mathbf{8100} \pm 24$
TCUWin(1.7,$\frac{.01}{2n}$)	10525 ± 31	$\mathbf{7901} \pm 26$
TCUWin(1.7,$\frac{.1}{2n}$)	10289 ± 31	$\mathbf{7727} \pm 27$
TCUWin(1.7,$\frac{1}{2n}$)	10338 ± 32	$\mathbf{7721} \pm 26$
TCUWin(2.0,$\frac{.0001}{2n}$)	10647 ± 34	$\mathbf{8118} \pm 29$
TCUWin(2.0,$\frac{.01}{2n}$)	10347 ± 34	$\mathbf{7912} \pm 29$
TCUWin(2.0,$\frac{.1}{2n}$)	10132 ± 33	$\mathbf{7769} \pm 28$
TCUWin(2.0,$\frac{1}{2n}$)	10360 ± 33	$\mathbf{7845} \pm 28$
VR-Combine	8020 ± 27	$\mathbf{6721} \pm 25$

Table 7.8: Total mistakes on basic tracking and recycling algorithms from 15 tracking concepts.

Name	$\hat{\mathrm{M}}$(Name)	$\hat{\mathrm{M}}$(R-Name)
Perceptron	10769 ± 28	$\mathbf{8293} \pm 30$
ALMA(2)	10853 ± 34	$\mathbf{8801} \pm 29$
ALMA($\ln n$)	11545 ± 37	$\mathbf{9286} \pm 32$
Bal(1.05)	10719 ± 27	$\mathbf{8209} \pm 27$
Bal(1.2)	10450 ± 29	$\mathbf{8021} \pm 25$
Bal(1.4)	10485 ± 32	$\mathbf{8133} \pm 28$
Bal(1.7)	10925 ± 32	$\mathbf{8613} \pm 30$
Bal(2.0)	11424 ± 36	$\mathbf{8991} \pm 32$
UWin(1.05)	13172 ± 41	$\mathbf{9165} \pm 32$
UWin(1.2)	12248 ± 33	$\mathbf{8491} \pm 26$
UWin(1.4)	12060 ± 39	$\mathbf{8612} \pm 29$
UWin(1.7)	12599 ± 39	$\mathbf{9245} \pm 32$
UWin(2.0)	13335 ± 39	$\mathbf{9805} \pm 43$
CUWin(1.05)	13581 ± 34	$\mathbf{9435} \pm 28$
CUWin(1.2)	11888 ± 33	$\mathbf{8331} \pm 28$
CUWin(1.4)	11129 ± 33	$\mathbf{8113} \pm 25$
CUWin(1.7)	10813 ± 31	$\mathbf{8421} \pm 57$
CUWin(2.0)	10850 ± 38	$\mathbf{8912} \pm 85$
NWin(1.05,.3)	14179 ± 32	$\mathbf{11047} \pm 26$
NWin(1.2,.3)	11733 ± 30	$\mathbf{8746} \pm 26$
NWin(1.4,.3)	11274 ± 31	$\mathbf{8478} \pm 26$
NWin(1.7,.3)	11498 ± 30	$\mathbf{8769} \pm 32$
NWin(2.0,.3)	11989 ± 33	$\mathbf{9170} \pm 32$
NWin(1.05,.5)	11786 ± 31	$\mathbf{8411} \pm 29$
NWin(1.2,.5)	10916 ± 32	$\mathbf{7935} \pm 25$
NWin(1.4,.5)	10587 ± 31	$\mathbf{7909} \pm 21$
NWin(1.7,.5)	10647 ± 27	$\mathbf{8183} \pm 30$
NWin(2.0,.5)	10967 ± 35	$\mathbf{8519} \pm 29$
NWin(1.05,.7)	12143 ± 36	$\mathbf{9186} \pm 27$
NWin(1.2,.7)	10676 ± 28	$\mathbf{7944} \pm 25$
NWin(1.4,.7)	10238 ± 29	$\mathbf{7768} \pm 26$
NWin(1.7,.7)	10110 ± 34	$\mathbf{7909} \pm 27$
NWin(2.0,.7)	10224 ± 35	$\mathbf{8137} \pm 29$
VR-Combine	7965 ± 26	$\mathbf{6919} \pm 28$

Table 7.9: Total mistakes on basic fixed concept and recycling algorithms from 15 tracking concepts.

Concept	Recycle	VR-Combine3	Bag & Recycle	VR-Combine4
covtype	TCUWin$(1.4,\frac{1}{2n})$ 1315 ± 18	1268 ± 18	TCUWin$(1.4,\frac{1}{2n})$ 1285 ± 18	$\mathbf{1257} \pm 18$
isolet	TCUWin$(2.0,\frac{1}{2n})$ 284 ± 6	261 ± 6	TCUWin$(2.0,\frac{1}{2n})$ 265 ± 5	$\mathbf{255} \pm 5$
letter	TCUWin$(1.7,\frac{1}{2n})$ 319 ± 8	293 ± 7	TCUWin$(2.0,\frac{1}{2n})$ 290 ± 7	$\mathbf{287} \pm 7$
nursery	ALMA(3.0) 572 ± 12	523 ± 11	ALMA(3.0) 530 ± 12	$\mathbf{513} \pm 11$
optdigits	TCUWin$(1.4,\frac{1}{2n})$ 267 ± 5	216 ± 5	TCUWin$(1.7,\frac{1}{2n})$ 212 ± 5	$\mathbf{203} \pm 4$
page-blocks	TCUWin$(2.0,\frac{1}{2n})$ 361 ± 11	337 ± 10	TCUWin$(2.0,\frac{1}{2n})$ 346 ± 10	$\mathbf{333} \pm 10$
pendigits	TCUWin$(1.4,\frac{1}{2n})$ 290 ± 6	259 ± 7	TCUWin$(1.7,\frac{1}{2n})$ 248 ± 6	$\mathbf{242} \pm 6$
sat	TCUWin$(1.4,\frac{1}{2n})$ 545 ± 10	489 ± 10	TCUWin$(1.4,\frac{1}{2n})$ 500 ± 9	$\mathbf{476} \pm 9$
segmentation	TCUWin$(1.4,\frac{1}{2n})$ 460 ± 10	407 ± 10	TCUWin$(1.4,\frac{1}{2n})$ 415 ± 10	$\mathbf{393} \pm 10$
shuttle	TCUWin$(1.4,\frac{1}{2n})$ 349 ± 10	318 ± 9	TCUWin$(1.4,\frac{1}{2n})$ 338 ± 10	$\mathbf{315} \pm 9$
mfeat	TCUWin$(2.0,\frac{1}{2n})$ 153 ± 4	116 ± 3	TCUWin$(2.0,\frac{1}{2n})$ 118 ± 3	$\mathbf{110} \pm 3$
yeast	TCUWin$(1.7,\frac{1}{2n})$ 770 ± 11	724 ± 11	TCUWin$(1.7,\frac{1}{2n})$ 748 ± 11	$\mathbf{717} \pm 11$
news	TCUWin$(1.2,\frac{.0001}{2n})$ 504 ± 8	456 ± 7	TCUWin$(2.0,\frac{1}{2n})$ 487 ± 8	$\mathbf{447} \pm 7$
reuters	ALMA(3.25) 455 ± 9	392 ± 7	ALMA(3.0) 402 ± 8	$\mathbf{379} \pm 7$
web	TUWin$(1.4,\frac{1}{2n})$ 804 ± 10	662 ± 9	ALMA(3.25) 737 ± 11	$\mathbf{638} \pm 9$

Table 7.10: Number of mistakes for best tracking algorithm on 15 tracking problems. Includes instance recycling, bagging and VR-Combine techniques.

tracking concepts. The first column gives the results for the algorithms with recycling. The second column gives the results with VR-Combine3. We will refer to the third and forth column in the next section when we talk about combining bagging and recycling. Comparing with Table 7.6, we see that recycling improves performance over the basic algorithm on every tracking problem. Unsurprisingly, VR-Combine3 makes even fewer mistakes. These trends are repeated in Table 7.11 with the fixed concept algorithms.

Concept	Recycle	VR-Combine3	Bag & Recycle	VR-Combine4
covtype	NWin(1.4,.7) 1334 ± 18	1291 ± 18	NWin(1.4,.7) 1297 ± 18	$\mathbf{1278} \pm 17$
isolet	NWin(2.0,.7) 266 ± 7	243 ± 5	NWin(2.0,.7) 232 ± 5	$\mathbf{228} \pm 5$
letter	NWin(1.7,.7) 333 ± 7	300 ± 6	NWin(1.7,.7) 295 ± 7	$\mathbf{294} \pm 6$
nursery	ALMA(2) 569 ± 11	525 ± 11	ALMA($\ln n$) 524 ± 12	$\mathbf{510} \pm 11$
optdigits	NWin(1.4,.7) 267 ± 6	236 ± 5	NWin(1.4,.7) 220 ± 5	$\mathbf{216} \pm 5$
page-blocks	NWin(1.7,.7) 372 ± 11	341 ± 9	NWin(1.7,.7) 350 ± 10	$\mathbf{337} \pm 10$
pendigits	NWin(1.2,.7) 315 ± 7	279 ± 7	NWin(1.4,.7) 269 ± 6	$\mathbf{262} \pm 6$
sat	NWin(1.2,.5) 537 ± 9	503 ± 9	NWin(1.05,.5) $\mathbf{480} \pm 9$	484 ± 9
segmentation	NWin(1.4,.7) 470 ± 11	427 ± 10	NWin(1.4,.7) 420 ± 9	$\mathbf{409} \pm 9$
shuttle	NWin(1.4,.5) 358 ± 11	320 ± 10	NWin(1.7,.5) 346 ± 11	$\mathbf{315} \pm 9$
mfeat	NWin(1.4,.5) 168 ± 5	131 ± 4	NWin(2.0,.5) 126 ± 4	$\mathbf{121} \pm 4$
yeast	NWin(1.2,.7) 770 ± 11	726 ± 11	NWin(1.2,.7) 746 ± 10	$\mathbf{726} \pm 10$
news	NWin(1.05,.7) 496 ± 8	456 ± 7	Bal(2.0) 484 ± 9	$\mathbf{447} \pm 7$
reuters	Bal(1.4) 452 ± 8	404 ± 8	Bal(1.4) 395 ± 8	$\mathbf{385} \pm 8$
web	Bal(1.2) 837 ± 12	737 ± 11	Bal(1.2) 746 ± 10	$\mathbf{695} \pm 9$

Table 7.11: Number of mistakes for best fixed concept algorithm on 15 tracking problems. Includes instance recycling, bagging and VR-Combine techniques.

7.5 Combining Recycling and Voting

In this section, we consider combining voting and recycling to further improve the performance of on-line algorithms on tracking problems. While we do not expect a decrease in mistakes as dramatic as the individual techniques, the goal is a decrease that is significant given the extra cost involved. We have already seen how to combine voting and instance recycling using the VR-Combine3 algorithm. Combining bagging and instance recycling is also straightforward. The recycling technique is applied to every basic algorithm and these algorithms are used by the bagging technique.

One important criteria for applying these techniques is computational cost. The analysis is complicated by the fact that our Tracking Unnormalized Winnow implementation performs updates in $O(n)$, where n is the number of attributes, while the remaining algorithms perform updates in $O(m)$, where m is the maximum number of non-zero attributes in an instance, and by the fact that we use some parameters twice, once for the tracking problem and once for VR-Combine.

We use the following notation for the algorithms that process instances from the tracking problem. Let v_1 be the number of basic algorithms, let u_1 be the number of times an instance can be used for an update, and let s_1 be the size of the window of old instances. In addition, let M be the maximum number of mistakes made by a basic algorithm and let T be the current trial number. We also need notation for VR-Combine when it is used to perform voting on the tracking problem. For VR-Combine let v_2 be the number of basic algorithms, let u_2 be the number of times an instance can be used for an update, and let s_2 be the size of the window of old instances. For VR-Combine, the maximum size of an instance is v_1 since the predictions of the algorithms used for the tracking problem are used to generate the new instances.

In Table 7.12, we give the cost of various combinations of techniques. We have arranged the order so that the table is organized from least expensive to most expensive based on the experiments used in this chapter. However, the table is slightly misleading in that the first technique, VR-Combine, must be used in combination with one of the other techniques. However, for our experiments, VR-Combine generally only adds a

Algorithm	Without Tracking Winnow	With Tracking Winnow
VR-Combine	$O(v_2u_2s_2v_1T)$	$O(v_2u_2s_2v_1T)$
Basic	$O(v_1mT)$	$O(v_1(mT+nM))$
Basic & Recycling	$O(v_1u_1s_1mT)$	$O(v_1u_1T(s_1m+n))$
Basic & Bagging	$O(hv_1mT)$	$O(hv_1(mT+nM))$
Recycling & Bagging	$O(hv_1u_1s_1mT)$	$O(hv_1u_1T(s_1m+n))$

Table 7.12: Asymptotic cost of running algorithms from this chapter.

minimal cost to the other techniques

Comparing the cost of the basic algorithms with VR-Combine, the factors that do not cancel are $v_2u_2s_2$ for VR-Combine and m for the basic algorithms without Tracking Winnow. While VR-Combine may appear expensive, one must recall that the u_2s_2 factor overestimates the increase in cost from instance recycling. For most practical problems, the algorithms do not come close to to averaging u_2s_2 predictions every trial. See Chapter 4 for more details. Therefore, as long as $v_2u_2s_2$ is not significantly larger than m, VR-Combine should be efficient relative to the basic algorithms. For our experiments $v_2 = 33$, $u_2 = 1$, and $s_2 = 100$ while the average value of m is 588. Things are even better for VR-Combine when taking into account the additional cost associated with Tracking Winnow.

The next most expensive algorithm is the recycling algorithm. While it adds an extra u_1s_1 factor to most basic algorithms, again the increase is generally more modest. On most practical problems, algorithms do not average close to u_1s_1 predictions every trial. The increase with Tracking Winnow is even smaller as the n component of the bound only increases by a u_1 factor. This is because $O(n)$ is need to update a hypothesis, but prediction still takes $O(m)$. See Appendix D for more details.

The addition of bagging causes a large increase in computational cost. A bagging algorithm essentially runs h versions of the same algorithm, and therefore the h factor is an accurate estimate of the increase in cost. The same is true when combining recycling and bagging. The bagging causes a factor of h increase in the cost of the recycling algorithm.

In the remainder of this chapter, we give the results of experiments that combine

the voting and recycling techniques. Based on these results, for tracking problems we recommend using VR-Combine3. We find that bagging is too expensive to justify given its relatively small performance gains. At the end of the chapter, we focus on exploring the effect of the s_1 and u_1 parameters on the recycling algorithms and VR-Combine3.

7.5.1 Recycling and Bagging Experiments

In Table 7.13, we give the total number of mistakes over all the tracking problems of the combination bagging and recycling algorithm. For comparison purposes, we also include the total mistake count for the basic algorithm, the bagging algorithm, and the recycling algorithm. As can be seen, every algorithm has a decrease in the number of mistakes as it progresses from the basic algorithm to the combined bagging and recycling algorithm. The same result can be seen in Table 7.14 with respect to the fixed concept algorithms.

Again the best algorithm is VR-Combine. This algorithm makes the fewest mistakes with only a small extra computational cost with respect to the other algorithms in the same column. Referring back to Table 7.10 and Table 7.11, the third column gives the best algorithm on each of the fifteen tracking problems. The last column gives the number of mistakes made by VR-Combine using the recycling and bagging versions of the basic algorithms. For convenience, we call this algorithm VR-Combine4.

Notice that VR-Combine4 gives the best performance for each tracking problem. This again shows that the VR-Combine algorithm is doing more than just selecting the best algorithm. If possible, it combines the algorithms to perform better than any single input algorithm. However, VR-Combine4 only makes approximately 3% fewer mistakes than VR-Combine3. While this is a statistically significant increase in performance, it is difficult to justify based on the factor of 30 cost increase for bagging. For many problems, this computational effort could be better used by running more basic algorithms for input to VR-Combine.

Name	$\hat{\mathrm{M}}$(Name)	$\hat{\mathrm{M}}$(B-Name)	$\hat{\mathrm{M}}$(R-Name)	$\hat{\mathrm{M}}$(B-R-Name)
TALMA(2.0)	10778 ± 27	9125 ± 27	8320 ± 25	$\mathbf{7674} \pm 27$
TALMA(2.25)	10566 ± 26	8896 ± 29	8130 ± 30	$\mathbf{7493} \pm 27$
TALMA(2.5)	10478 ± 27	8771 ± 28	8052 ± 25	$\mathbf{7408} \pm 25$
TALMA(2.75)	10432 ± 27	8710 ± 27	8018 ± 26	$\mathbf{7364} \pm 25$
TALMA(3.0)	10443 ± 31	8678 ± 28	8003 ± 28	$\mathbf{7345} \pm 25$
TALMA(3.25)	10450 ± 34	8658 ± 26	7990 ± 28	$\mathbf{7340} \pm 25$
TALMA(3.5)	10485 ± 32	8667 ± 28	8010 ± 28	$\mathbf{7343} \pm 25$
TALMA(3.75)	10518 ± 35	8669 ± 25	8033 ± 26	$\mathbf{7365} \pm 23$
TALMA(4.0)	10561 ± 32	8683 ± 30	8061 ± 27	$\mathbf{7384} \pm 24$
TALMA(4.25)	10606 ± 35	8711 ± 28	8098 ± 28	$\mathbf{7408} \pm 23$
TALMA(4.5)	10674 ± 33	8722 ± 29	8113 ± 27	$\mathbf{7436} \pm 23$
TALMA(4.75)	10719 ± 30	8757 ± 27	8158 ± 28	$\mathbf{7461} \pm 24$
TALMA(5.0)	10773 ± 29	8778 ± 29	8195 ± 29	$\mathbf{7497} \pm 24$
TALMA(5.25)	10835 ± 33	8794 ± 27	8221 ± 26	$\mathbf{7513} \pm 24$
TALMA(5.5)	10882 ± 30	8843 ± 28	8253 ± 26	$\mathbf{7554} \pm 24$
TALMA(5.75)	10922 ± 32	8868 ± 27	8296 ± 29	$\mathbf{7573} \pm 24$
TALMA(6.0)	10984 ± 33	8902 ± 27	8325 ± 28	$\mathbf{7603} \pm 24$
TALMA(6.25)	11038 ± 37	8926 ± 30	8355 ± 29	$\mathbf{7636} \pm 22$
TALMA(6.5)	11070 ± 29	8953 ± 30	8390 ± 25	$\mathbf{7660} \pm 24$
TALMA($\ln n$)	11109 ± 34	8940 ± 28	8394 ± 26	$\mathbf{7660} \pm 23$
TCUWin(1.05,$\frac{.0001}{2n}$)	13582 ± 35	12662 ± 33	9468 ± 25	$\mathbf{8992} \pm 26$
TCUWin(1.05,$\frac{.01}{2n}$)	13579 ± 35	12657 ± 31	9459 ± 25	$\mathbf{8979} \pm 26$
TCUWin(1.05,$\frac{.1}{2n}$)	13530 ± 34	12631 ± 32	9418 ± 26	$\mathbf{8931} \pm 28$
TCUWin(1.05,$\frac{1}{2n}$)	13238 ± 30	12205 ± 31	9133 ± 28	$\mathbf{8659} \pm 26$
TCUWin(1.2,$\frac{.0001}{2n}$)	11888 ± 33	10742 ± 31	8347 ± 27	$\mathbf{7875} \pm 25$
TCUWin(1.2,$\frac{.01}{2n}$)	11848 ± 33	10702 ± 33	8312 ± 27	$\mathbf{7840} \pm 24$
TCUWin(1.2,$\frac{.1}{2n}$)	11739 ± 32	10608 ± 28	8219 ± 24	$\mathbf{7749} \pm 25$
TCUWin(1.2,$\frac{1}{2n}$)	11251 ± 30	9987 ± 31	7896 ± 24	$\mathbf{7426} \pm 21$
TCUWin(1.4,$\frac{.0001}{2n}$)	11126 ± 34	9655 ± 29	8125 ± 27	$\mathbf{7593} \pm 23$
TCUWin(1.4,$\frac{.01}{2n}$)	11040 ± 32	9575 ± 29	8020 ± 25	$\mathbf{7507} \pm 24$
TCUWin(1.4,$\frac{.1}{2n}$)	10817 ± 31	9369 ± 29	7844 ± 25	$\mathbf{7349} \pm 21$
TCUWin(1.4,$\frac{1}{2n}$)	10534 ± 31	8954 ± 29	7685 ± 25	$\mathbf{7198} \pm 24$
TCUWin(1.7,$\frac{.0001}{2n}$)	10768 ± 32	9127 ± 28	8100 ± 24	$\mathbf{7544} \pm 24$
TCUWin(1.7,$\frac{.01}{2n}$)	10525 ± 31	8954 ± 27	7901 ± 26	$\mathbf{7366} \pm 24$
TCUWin(1.7,$\frac{.1}{2n}$)	10289 ± 31	8697 ± 29	7727 ± 27	$\mathbf{7212} \pm 25$
TCUWin(1.7,$\frac{1}{2n}$)	10338 ± 32	8569 ± 27	7721 ± 26	$\mathbf{7215} \pm 25$
TCUWin(2.0,$\frac{.0001}{2n}$)	10647 ± 34	8891 ± 27	8118 ± 29	$\mathbf{7548} \pm 28$
TCUWin(2.0,$\frac{.01}{2n}$)	10347 ± 34	8634 ± 28	7912 ± 29	$\mathbf{7355} \pm 26$
TCUWin(2.0,$\frac{.1}{2n}$)	10132 ± 33	8406 ± 25	7769 ± 28	$\mathbf{7248} \pm 26$
TCUWin(2.0,$\frac{1}{2n}$)	10360 ± 33	8482 ± 28	7845 ± 28	$\mathbf{7345} \pm 26$
VR-Combine	8020 ± 27	7511 ± 26	6721 ± 25	$\mathbf{6565} \pm 24$

Table 7.13: Total mistakes on basic tracking algorithms with bagging and recycling.

Name	$\hat{\mathrm{M}}$(Name)	$\hat{\mathrm{M}}$(B-Name)	$\hat{\mathrm{M}}$(R-Name)	$\hat{\mathrm{M}}$(BR-Name)
Perceptron	10769 ± 28	9133 ± 25	8293 ± 30	$\mathbf{7662} \pm 25$
ALMA(2)	10853 ± 34	9423 ± 28	8801 ± 29	$\mathbf{8064} \pm 27$
ALMA($\ln n$)	11545 ± 37	9610 ± 31	9286 ± 32	$\mathbf{8343} \pm 26$
Bal(1.05)	10719 ± 27	9063 ± 28	8209 ± 27	$\mathbf{7577} \pm 27$
Bal(1.2)	10450 ± 29	8819 ± 27	8021 ± 25	$\mathbf{7406} \pm 26$
Bal(1.4)	10485 ± 32	8712 ± 27	8133 ± 28	$\mathbf{7450} \pm 26$
Bal(1.7)	10925 ± 32	8816 ± 26	8613 ± 30	$\mathbf{7805} \pm 27$
Bal(2.0)	11424 ± 36	9012 ± 32	8991 ± 32	$\mathbf{8131} \pm 29$
UWin(1.05)	13172 ± 41	12109 ± 33	9165 ± 32	$\mathbf{8617} \pm 26$
UWin(1.2)	12248 ± 33	10793 ± 29	8491 ± 26	$\mathbf{7908} \pm 23$
UWin(1.4)	12060 ± 39	10137 ± 29	8612 ± 29	$\mathbf{7978} \pm 24$
UWin(1.7)	12599 ± 39	10142 ± 30	9245 ± 32	$\mathbf{8509} \pm 27$
UWin(2.0)	13335 ± 39	10463 ± 30	9805 ± 43	$\mathbf{9089} \pm 31$
CUWin(1.05)	13581 ± 34	12649 ± 29	9435 ± 28	$\mathbf{8954} \pm 25$
CUWin(1.2)	11888 ± 33	10727 ± 30	8331 ± 28	$\mathbf{7850} \pm 24$
CUWin(1.4)	11129 ± 33	9637 ± 30	8113 ± 25	$\mathbf{7592} \pm 23$
CUWin(1.7)	10813 ± 31	9129 ± 26	8421 ± 57	$\mathbf{7670} \pm 30$
CUWin(2.0)	10850 ± 38	8958 ± 31	8912 ± 85	$\mathbf{8061} \pm 63$
NWin(1.05,.3)	14179 ± 32	13680 ± 30	11047 ± 26	$\mathbf{10408} \pm 29$
NWin(1.2,.3)	11733 ± 30	10553 ± 30	8746 ± 26	$\mathbf{8170} \pm 25$
NWin(1.4,.3)	11274 ± 31	9695 ± 28	8478 ± 26	$\mathbf{7829} \pm 24$
NWin(1.7,.3)	11498 ± 30	9407 ± 28	8769 ± 32	$\mathbf{8044} \pm 27$
NWin(2.0,.3)	11989 ± 33	9501 ± 28	9170 ± 32	$\mathbf{8398} \pm 27$
NWin(1.05,.5)	11786 ± 31	10357 ± 28	8411 ± 29	$\mathbf{7866} \pm 24$
NWin(1.2,.5)	10916 ± 32	9365 ± 28	7935 ± 25	$\mathbf{7389} \pm 27$
NWin(1.4,.5)	10587 ± 31	8876 ± 28	7909 ± 21	$\mathbf{7306} \pm 24$
NWin(1.7,.5)	10647 ± 27	8651 ± 30	8183 ± 30	$\mathbf{7470} \pm 24$
NWin(2.0,.5)	10967 ± 35	8690 ± 30	8519 ± 29	$\mathbf{7751} \pm 27$
NWin(1.05,.7)	12143 ± 36	11621 ± 32	9186 ± 27	$\mathbf{8730} \pm 25$
NWin(1.2,.7)	10676 ± 28	9541 ± 28	7944 ± 25	$\mathbf{7471} \pm 25$
NWin(1.4,.7)	10238 ± 29	8830 ± 30	7768 ± 26	$\mathbf{7224} \pm 24$
NWin(1.7,.7)	10110 ± 34	8431 ± 28	7909 ± 27	$\mathbf{7248} \pm 26$
NWin(2.0,.7)	10224 ± 35	8335 ± 27	8137 ± 29	$\mathbf{7402} \pm 25$
VR-Combine	7965 ± 26	7462 ± 28	6919 ± 28	$\mathbf{6705} \pm 23$

Table 7.14: Total mistakes on basic fixed concept algorithms with bagging and recycling.

7.5.2 Instance Recycling Parameters

Next, we explore various parameter values for the recycling algorithm. We do not include the bagging algorithm because, based on previous experiments, its cost is not justified by its performance gains. However, we do include VR-Combine3. When used with the recycled algorithms, it gives a large performance boost for almost no extra cost.

In Table 7.15, we give the total number of mistakes made by the tracking algorithms for various sizes of the instance recycling window. We give the results for window sizes $s = \{0, 10, 50, 100, 150\}$. The same results are presented in Table 7.16 for the fixed concept algorithms. The best two window sizes for both tables are 50 and 100. In Table 7.15, the results are somewhat surprising as $s = 100$ does the best for all the algorithms but $s = 50$ does the best for VR-Combine. However, the difference between VR-Combine with $s = 50$ and VR-Combine with $s = 100$ is small. We assume, just as in Chapter 5, the smaller s values give more diversity in the hypotheses used by VR-Combine. In Table 7.16, we see a much more even mix between $s = 50$ and $s = 100$. Again, the main algorithm, VR-Combine, does best with $s = 50$. Therefore, our default value performs best on these data sets.

In Table 7.17, we give the total number of mistakes made by the tracking algorithms for various values of u, the maximum number of times an instance can be used for an update. We give the results for $u = \{1, 2, 3, 4, 5\}$. The same results are presented in Table 7.18 for the fixed concept algorithms. There appears to be a rough pattern for both ALMA and the Winnow algorithms.

Tracking ALMA generally does best when $u = 3$. While there are five cases where $u = 3$ does not give the best performance, these cases do not appear statistically significant. We only have two parameter values for the normal ALMA algorithm, but the algorithm makes the fewest mistakes on both when $u = 5$. These results are on the edge of statistical relevance if variance reduction techniques are used. The Winnow algorithms tend to prefer a larger u value when they use a smaller multiplier. This is the same behavior we saw in Section 5.2.2. A larger multiplier causes an increase in

Name	$\hat{\mathrm{M}}(\mathrm{R}(0))$	$\hat{\mathrm{M}}(\mathrm{R}(10))$	$\hat{\mathrm{M}}(\mathrm{R}(50))$	$\hat{\mathrm{M}}(\mathrm{R}(100))$	$\hat{\mathrm{M}}(\mathrm{R}(150))$
TALMA(2.0)	10778 ± 27	9011 ± 33	8320 ± 25	$\mathbf{8299} \pm 29$	8397 ± 25
TALMA(2.25)	10566 ± 26	8857 ± 27	8130 ± 30	$\mathbf{8116} \pm 26$	8222 ± 26
TALMA(2.5)	10478 ± 27	8790 ± 27	8052 ± 25	$\mathbf{8022} \pm 26$	8122 ± 25
TALMA(2.75)	10432 ± 27	8758 ± 30	8018 ± 26	$\mathbf{7977} \pm 25$	8070 ± 30
TALMA(3.0)	10443 ± 31	8749 ± 28	8003 ± 28	$\mathbf{7950} \pm 25$	8048 ± 25
TALMA(3.25)	10450 ± 34	8760 ± 32	7990 ± 28	$\mathbf{7948} \pm 26$	8048 ± 26
TALMA(3.5)	10485 ± 32	8798 ± 27	8010 ± 28	$\mathbf{7945} \pm 30$	8044 ± 27
TALMA(3.75)	10518 ± 35	8820 ± 28	8033 ± 26	$\mathbf{7981} \pm 27$	8068 ± 25
TALMA(4.0)	10561 ± 32	8857 ± 32	8061 ± 27	$\mathbf{7993} \pm 26$	8078 ± 24
TALMA(4.25)	10606 ± 35	8892 ± 33	8098 ± 28	$\mathbf{8021} \pm 27$	8105 ± 28
TALMA(4.5)	10674 ± 33	8920 ± 30	8113 ± 27	$\mathbf{8048} \pm 28$	8128 ± 26
TALMA(4.75)	10719 ± 30	8977 ± 36	8158 ± 28	$\mathbf{8080} \pm 25$	8175 ± 25
TALMA(5.0)	10773 ± 29	9016 ± 30	8195 ± 29	$\mathbf{8105} \pm 27$	8201 ± 27
TALMA(5.25)	10835 ± 33	9067 ± 28	8221 ± 26	$\mathbf{8134} \pm 29$	8226 ± 27
TALMA(5.5)	10882 ± 30	9107 ± 31	8253 ± 26	$\mathbf{8173} \pm 26$	8258 ± 28
TALMA(5.75)	10922 ± 32	9144 ± 30	8296 ± 29	$\mathbf{8212} \pm 26$	8290 ± 28
TALMA(6.0)	10984 ± 33	9190 ± 32	8325 ± 28	$\mathbf{8244} \pm 26$	8333 ± 27
TALMA(6.25)	11038 ± 37	9227 ± 33	8355 ± 29	$\mathbf{8269} \pm 26$	8345 ± 27
TALMA(6.5)	11070 ± 29	9276 ± 29	8390 ± 25	$\mathbf{8298} \pm 25$	8387 ± 26
TALMA($\ln n$)	11109 ± 34	9246 ± 30	8394 ± 26	$\mathbf{8290} \pm 24$	8369 ± 28
TCUWin(1.05,$\frac{.0001}{2n}$)	13582 ± 35	10130 ± 29	9468 ± 25	$\mathbf{9452} \pm 26$	9553 ± 25
TCUWin(1.05,$\frac{.01}{2n}$)	13579 ± 35	10120 ± 28	9459 ± 25	$\mathbf{9440} \pm 27$	9543 ± 25
TCUWin(1.05,$\frac{.1}{2n}$)	13530 ± 34	10080 ± 29	9418 ± 26	$\mathbf{9403} \pm 27$	9504 ± 26
TCUWin(1.05,$\frac{1}{2n}$)	13238 ± 30	9795 ± 32	9133 ± 28	$\mathbf{9102} \pm 28$	9192 ± 28
TCUWin(1.2,$\frac{.0001}{2n}$)	11888 ± 33	9038 ± 28	8347 ± 27	$\mathbf{8336} \pm 22$	8447 ± 23
TCUWin(1.2,$\frac{.01}{2n}$)	11848 ± 33	9004 ± 28	8312 ± 27	$\mathbf{8301} \pm 23$	8411 ± 23
TCUWin(1.2,$\frac{.1}{2n}$)	11739 ± 32	8927 ± 28	8219 ± 24	$\mathbf{8191} \pm 24$	8311 ± 25
TCUWin(1.2,$\frac{1}{2n}$)	11251 ± 30	8577 ± 27	7896 ± 24	$\mathbf{7886} \pm 24$	8001 ± 21
TCUWin(1.4,$\frac{.0001}{2n}$)	11126 ± 34	8800 ± 28	8125 ± 27	$\mathbf{8103} \pm 24$	8210 ± 25
TCUWin(1.4,$\frac{.01}{2n}$)	11040 ± 32	8721 ± 27	8020 ± 25	$\mathbf{7979} \pm 24$	8074 ± 24
TCUWin(1.4,$\frac{.1}{2n}$)	10817 ± 31	8559 ± 26	7844 ± 25	$\mathbf{7801} \pm 23$	7909 ± 24
TCUWin(1.4,$\frac{1}{2n}$)	10534 ± 31	8389 ± 27	7685 ± 25	$\mathbf{7658} \pm 24$	7770 ± 21
TCUWin(1.7,$\frac{.0001}{2n}$)	10768 ± 32	8808 ± 27	8100 ± 24	$\mathbf{8054} \pm 28$	8168 ± 23
TCUWin(1.7,$\frac{.01}{2n}$)	10525 ± 31	8597 ± 32	7901 ± 26	$\mathbf{7856} \pm 25$	7956 ± 25
TCUWin(1.7,$\frac{.1}{2n}$)	10289 ± 31	8418 ± 29	7727 ± 27	$\mathbf{7689} \pm 25$	7807 ± 24
TCUWin(1.7,$\frac{1}{2n}$)	10338 ± 32	8437 ± 28	7721 ± 26	$\mathbf{7675} \pm 27$	7766 ± 29
TCUWin(2.0,$\frac{.0001}{2n}$)	10647 ± 34	8852 ± 28	8118 ± 29	$\mathbf{8068} \pm 25$	8158 ± 24
TCUWin(2.0,$\frac{.01}{2n}$)	10347 ± 34	8604 ± 28	7912 ± 29	$\mathbf{7853} \pm 27$	7964 ± 27
TCUWin(2.0,$\frac{.1}{2n}$)	10132 ± 33	8451 ± 26	7769 ± 28	$\mathbf{7715} \pm 27$	7826 ± 25
TCUWin(2.0,$\frac{1}{2n}$)	10360 ± 33	8586 ± 29	7845 ± 28	$\mathbf{7781} \pm 25$	7874 ± 25
VR-Combine	8020 ± 27	7134 ± 25	$\mathbf{6721} \pm 25$	6750 ± 23	6808 ± 24

Table 7.15: Total mistakes with recycling on tracking algorithms using recycling parameters $u = 1$ and $s = \{0, 10, 50, 100, 150\}$.

Name	$\hat{\mathrm{M}}(\mathrm{R}(0))$	$\hat{\mathrm{M}}(\mathrm{R}(10))$	$\hat{\mathrm{M}}(\mathrm{R}(50))$	$\hat{\mathrm{M}}(\mathrm{R}(100))$	$\hat{\mathrm{M}}(\mathrm{R}(150))$
Perceptron	10769 ± 28	8995 ± 27	**8293** ± 30	8309 ± 28	8468 ± 27
ALMA(2)	10853 ± 34	9417 ± 29	**8801** ± 29	8809 ± 29	8945 ± 26
ALMA($\ln n$)	11545 ± 37	10077 ± 35	9286 ± 32	**9246** ± 32	9362 ± 30
Bal(1.05)	10719 ± 27	8927 ± 31	**8209** ± 27	8222 ± 24	8353 ± 24
Bal(1.2)	10450 ± 29	8754 ± 25	8021 ± 25	**8018** ± 28	8160 ± 26
Bal(1.4)	10485 ± 32	8851 ± 30	**8133** ± 28	8139 ± 30	8274 ± 29
Bal(1.7)	10925 ± 32	9321 ± 31	**8613** ± 30	8615 ± 29	8746 ± 31
Bal(2.0)	11424 ± 36	9749 ± 29	8991 ± 32	**8979** ± 37	9109 ± 30
UWin(1.05)	13172 ± 41	9762 ± 33	**9165** ± 32	9189 ± 32	9335 ± 27
UWin(1.2)	12248 ± 33	9218 ± 28	**8491** ± 26	8513 ± 26	8671 ± 24
UWin(1.4)	12060 ± 39	9363 ± 29	**8612** ± 29	8641 ± 26	8812 ± 26
UWin(1.7)	12599 ± 39	9988 ± 32	**9245** ± 32	9308 ± 31	9472 ± 34
UWin(2.0)	13335 ± 39	10590 ± 34	**9805** ± 43	9846 ± 41	9995 ± 43
CUWin(1.05)	13581 ± 34	10081 ± 29	**9435** ± 28	9477 ± 28	9617 ± 26
CUWin(1.2)	11888 ± 33	9007 ± 30	**8331** ± 28	8361 ± 27	8524 ± 25
CUWin(1.4)	11129 ± 33	8787 ± 29	**8113** ± 25	8143 ± 27	8306 ± 24
CUWin(1.7)	10813 ± 31	8975 ± 31	**8421** ± 57	8525 ± 68	8659 ± 66
CUWin(2.0)	10850 ± 38	9242 ± 57	**8912** ± 85	8954 ± 112	9143 ± 108
NWin(1.05,.3)	14179 ± 32	11763 ± 33	11047 ± 26	**10952** ± 25	11009 ± 26
NWin(1.2,.3)	11733 ± 30	9512 ± 26	8746 ± 26	**8688** ± 27	8813 ± 25
NWin(1.4,.3)	11274 ± 31	9218 ± 29	8478 ± 26	**8461** ± 25	8611 ± 28
NWin(1.7,.3)	11498 ± 30	9550 ± 33	**8769** ± 32	8778 ± 28	8914 ± 30
NWin(2.0,.3)	11989 ± 33	9992 ± 33	**9170** ± 32	9180 ± 27	9328 ± 29
NWin(1.05,.5)	11786 ± 31	9036 ± 31	**8411** ± 29	8413 ± 27	8540 ± 27
NWin(1.2,.5)	10916 ± 32	8649 ± 29	**7935** ± 25	7941 ± 24	8080 ± 24
NWin(1.4,.5)	10587 ± 31	8625 ± 30	**7909** ± 21	7919 ± 27	8068 ± 23
NWin(1.7,.5)	10647 ± 27	8883 ± 26	**8183** ± 30	8192 ± 26	8349 ± 26
NWin(2.0,.5)	10967 ± 35	9252 ± 32	**8519** ± 29	8526 ± 30	8682 ± 27
NWin(1.05,.7)	12143 ± 36	9810 ± 31	9186 ± 27	**9151** ± 28	9240 ± 27
NWin(1.2,.7)	10676 ± 28	8594 ± 29	**7944** ± 25	7951 ± 24	8080 ± 24
NWin(1.4,.7)	10238 ± 29	8427 ± 28	**7768** ± 26	**7768** ± 24	7915 ± 26
NWin(1.7,.7)	10110 ± 34	8555 ± 30	7909 ± 27	**7905** ± 24	8038 ± 24
NWin(2.0,.7)	10224 ± 35	8785 ± 32	**8137** ± 29	8144 ± 29	8285 ± 27
VR-Combine	7965 ± 26	7265 ± 27	**6919** ± 28	6963 ± 25	7035 ± 23

Table 7.16: Total mistakes with recycling on fixed concept algorithms using recycling parameters $u = 1$ and $s = \{0, 10, 50, 100, 150\}$.

Name	$\hat{\mathrm{M}}(\mathrm{R}(1))$	$\hat{\mathrm{M}}(\mathrm{R}(2))$	$\hat{\mathrm{M}}(\mathrm{R}(3))$	$\hat{\mathrm{M}}(\mathrm{R}(4))$	$\hat{\mathrm{M}}(\mathrm{R}(5))$
TALMA(2.0)	8320 ± 25	7979 ± 26	7910 ± 26	$\mathbf{7896} \pm 29$	7910 ± 28
TALMA(2.25)	8130 ± 30	7856 ± 24	7822 ± 25	$\mathbf{7821} \pm 28$	$\mathbf{7821} \pm 29$
TALMA(2.5)	8052 ± 25	7805 ± 23	$\mathbf{7769} \pm 25$	7784 ± 27	7796 ± 28
TALMA(2.75)	8018 ± 26	7790 ± 27	$\mathbf{7756} \pm 25$	7759 ± 28	7782 ± 30
TALMA(3.0)	8003 ± 28	7784 ± 24	$\mathbf{7749} \pm 28$	7767 ± 29	7778 ± 28
TALMA(3.25)	7990 ± 28	7797 ± 29	$\mathbf{7770} \pm 29$	7781 ± 28	7799 ± 30
TALMA(3.5)	8010 ± 28	7811 ± 30	$\mathbf{7781} \pm 27$	7807 ± 28	7816 ± 30
TALMA(3.75)	8033 ± 26	7834 ± 25	7823 ± 28	$\mathbf{7822} \pm 29$	7848 ± 30
TALMA(4.0)	8061 ± 27	7844 ± 25	$\mathbf{7835} \pm 28$	7849 ± 29	7862 ± 27
TALMA(4.25)	8098 ± 28	7888 ± 29	$\mathbf{7864} \pm 28$	7875 ± 28	7900 ± 28
TALMA(4.5)	8113 ± 27	7921 ± 26	$\mathbf{7902} \pm 30$	7915 ± 29	7927 ± 32
TALMA(4.75)	8158 ± 28	7953 ± 25	$\mathbf{7928} \pm 26$	7942 ± 29	7946 ± 29
TALMA(5.0)	8195 ± 29	7978 ± 25	$\mathbf{7966} \pm 31$	7968 ± 29	7988 ± 26
TALMA(5.25)	8221 ± 26	8006 ± 30	$\mathbf{8000} \pm 28$	8009 ± 26	8019 ± 29
TALMA(5.5)	8253 ± 26	8036 ± 25	8028 ± 27	$\mathbf{8022} \pm 26$	8043 ± 27
TALMA(5.75)	8296 ± 29	8073 ± 25	$\mathbf{8044} \pm 29$	8071 ± 25	8069 ± 29
TALMA(6.0)	8325 ± 28	8104 ± 26	$\mathbf{8088} \pm 27$	8094 ± 28	8104 ± 27
TALMA(6.25)	8355 ± 29	8132 ± 28	$\mathbf{8099} \pm 27$	8135 ± 31	8137 ± 33
TALMA(6.5)	8390 ± 25	8168 ± 29	8143 ± 24	$\mathbf{8142} \pm 31$	8149 ± 28
TALMA($\ln n$)	8394 ± 26	$\mathbf{8159} \pm 25$	$\mathbf{8159} \pm 27$	8171 ± 28	8197 ± 29
TCUWin(1.05, $\frac{.0001}{2n}$)	9468 ± 25	8604 ± 28	8276 ± 22	8148 ± 26	$\mathbf{8084} \pm 27$
TCUWin(1.05, $\frac{.01}{2n}$)	9459 ± 25	8591 ± 28	8264 ± 23	8137 ± 27	$\mathbf{8067} \pm 27$
TCUWin(1.05, $\frac{.1}{2n}$)	9418 ± 26	8547 ± 27	8218 ± 22	8095 ± 27	$\mathbf{8031} \pm 26$
TCUWin(1.05, $\frac{1}{2n}$)	9133 ± 28	8283 ± 26	8032 ± 28	7936 ± 29	$\mathbf{7887} \pm 27$
TCUWin(1.2, $\frac{.0001}{2n}$)	8347 ± 27	7884 ± 27	$\mathbf{7799} \pm 29$	7806 ± 28	7855 ± 26
TCUWin(1.2, $\frac{.01}{2n}$)	8312 ± 27	7847 ± 26	$\mathbf{7758} \pm 28$	7766 ± 27	7799 ± 28
TCUWin(1.2, $\frac{.1}{2n}$)	8219 ± 24	7758 ± 24	$\mathbf{7675} \pm 26$	7692 ± 26	7727 ± 30
TCUWin(1.2, $\frac{1}{2n}$)	7896 ± 24	7575 ± 28	$\mathbf{7543} \pm 27$	7567 ± 26	7627 ± 28
TCUWin(1.4, $\frac{.0001}{2n}$)	8125 ± 27	$\mathbf{7836} \pm 26$	7845 ± 26	7885 ± 28	7938 ± 28
TCUWin(1.4, $\frac{.01}{2n}$)	8020 ± 25	$\mathbf{7712} \pm 27$	7725 ± 29	7771 ± 29	7830 ± 28
TCUWin(1.4, $\frac{.1}{2n}$)	7844 ± 25	$\mathbf{7577} \pm 26$	7618 ± 25	7675 ± 27	7746 ± 28
TCUWin(1.4, $\frac{1}{2n}$)	7685 ± 25	$\mathbf{7542} \pm 26$	7596 ± 27	7697 ± 28	7781 ± 32
TCUWin(1.7, $\frac{.0001}{2n}$)	8100 ± 24	$\mathbf{7926} \pm 29$	7962 ± 28	8022 ± 29	8099 ± 27
TCUWin(1.7, $\frac{.01}{2n}$)	7901 ± 26	$\mathbf{7751} \pm 29$	7825 ± 24	7912 ± 25	8004 ± 28
TCUWin(1.7, $\frac{.1}{2n}$)	7727 ± 27	$\mathbf{7645} \pm 32$	7741 ± 30	7862 ± 30	7964 ± 30
TCUWin(1.7, $\frac{1}{2n}$)	7721 ± 26	$\mathbf{7712} \pm 26$	7839 ± 31	7973 ± 31	8083 ± 33
TCUWin(2.0, $\frac{.0001}{2n}$)	8118 ± 29	$\mathbf{8004} \pm 28$	8066 ± 27	8145 ± 27	8229 ± 29
TCUWin(2.0, $\frac{.01}{2n}$)	7912 ± 29	$\mathbf{7839} \pm 30$	7955 ± 28	8073 ± 31	8184 ± 30
TCUWin(2.0, $\frac{.1}{2n}$)	$\mathbf{7769} \pm 28$	7771 ± 30	7913 ± 29	8053 ± 28	8175 ± 32
TCUWin(2.0, $\frac{1}{2n}$)	$\mathbf{7845} \pm 28$	7909 ± 30	8075 ± 28	8205 ± 31	8335 ± 35
VR-Combine	6721 ± 25	6546 ± 25	6514 ± 24	$\mathbf{6508} \pm 25$	6517 ± 25

Table 7.17: Total mistakes with recycling on tracking algorithms using recycling parameters $u = \{1, 2, 3, 4, 5\}$ and $s = 50$.

Name	$\hat{\mathrm{M}}(\mathrm{R}(1))$	$\hat{\mathrm{M}}(\mathrm{R}(2))$	$\hat{\mathrm{M}}(\mathrm{R}(3))$	$\hat{\mathrm{M}}(\mathrm{R}(4))$	$\hat{\mathrm{M}}(\mathrm{R}(5))$
Perceptron	8293 ± 30	7961 ± 24	7891 ± 26	**7889** ± 27	7891 ± 25
ALMA(2)	8801 ± 29	8370 ± 31	8258 ± 30	8252 ± 26	**8240** ± 27
ALMA($\ln n$)	9286 ± 32	8962 ± 28	8901 ± 29	8879 ± 30	**8875** ± 32
Bal(1.05)	8209 ± 27	7903 ± 25	7852 ± 25	**7840** ± 30	7857 ± 28
Bal(1.2)	8021 ± 25	7786 ± 27	**7766** ± 24	7805 ± 25	7844 ± 29
Bal(1.4)	8133 ± 28	**8024** ± 27	8065 ± 32	8132 ± 28	8184 ± 28
Bal(1.7)	**8613** ± 30	8620 ± 32	8706 ± 32	8886 ± 52	9044 ± 34
Bal(2.0)	**8991** ± 32	9015 ± 33	9327 ± 34	9344 ± 57	9654 ± 82
UWin(1.05)	9165 ± 32	8456 ± 26	8167 ± 27	8026 ± 26	**7948** ± 26
UWin(1.2)	8491 ± 26	7992 ± 24	7869 ± 26	**7867** ± 29	7886 ± 30
UWin(1.4)	8612 ± 29	**8323** ± 28	8326 ± 29	8393 ± 29	8469 ± 28
UWin(1.7)	9245 ± 32	**9156** ± 38	9356 ± 66	9499 ± 66	9614 ± 67
UWin(2.0)	**9805** ± 43	9811 ± 55	10102 ± 83	10213 ± 97	10375 ± 79
CUWin(1.05)	9435 ± 28	8575 ± 28	8260 ± 26	8124 ± 24	**8051** ± 23
CUWin(1.2)	8331 ± 28	7856 ± 25	**7786** ± 25	7794 ± 26	7843 ± 31
CUWin(1.4)	8113 ± 25	**7881** ± 29	7908 ± 28	7977 ± 25	8063 ± 28
CUWin(1.7)	**8421** ± 57	8819 ± 75	9075 ± 63	9315 ± 80	9404 ± 74
CUWin(2.0)	**8912** ± 85	9436 ± 96	9696 ± 114	9955 ± 101	10128 ± 115
NWin(1.05,.3)	11047 ± 26	9848 ± 27	9415 ± 27	9216 ± 27	**9097** ± 28
NWin(1.2,.3)	8746 ± 26	8282 ± 26	8197 ± 31	**8195** ± 28	8224 ± 27
NWin(1.4,.3)	8478 ± 26	**8265** ± 26	8320 ± 29	8363 ± 27	8436 ± 31
NWin(1.7,.3)	8769 ± 32	**8703** ± 31	8799 ± 31	8880 ± 32	8920 ± 30
NWin(2.0,.3)	**9170** ± 32	**9170** ± 32	9246 ± 28	9287 ± 33	9325 ± 37
NWin(1.05,.5)	8411 ± 29	7981 ± 26	7855 ± 32	**7837** ± 27	7850 ± 28
NWin(1.2,.5)	7935 ± 25	**7707** ± 27	7710 ± 26	7753 ± 26	7812 ± 30
NWin(1.4,.5)	7909 ± 21	**7811** ± 30	7888 ± 33	7995 ± 29	8055 ± 32
NWin(1.7,.5)	**8183** ± 30	8205 ± 31	8302 ± 29	8395 ± 27	8443 ± 30
NWin(2.0,.5)	**8519** ± 29	8562 ± 28	8656 ± 29	8716 ± 29	8777 ± 37
NWin(1.05,.7)	9186 ± 27	8495 ± 26	8265 ± 26	**8183** ± 27	**8138** ± 26
NWin(1.2,.7)	7944 ± 25	7667 ± 23	**7665** ± 29	7705 ± 29	7773 ± 29
NWin(1.4,.7)	7768 ± 26	**7667** ± 27	7749 ± 28	7845 ± 29	7917 ± 27
NWin(1.7,.7)	**7909** ± 27	7939 ± 28	8043 ± 30	8147 ± 32	8217 ± 30
NWin(2.0,.7)	**8137** ± 29	8220 ± 34	8327 ± 33	8426 ± 32	8471 ± 34
VR-Combine	6919 ± 28	6734 ± 26	6695 ± 26	6679 ± 25	**6676** ± 26

Table 7.18: Total mistakes with recycling on fixed concept algorithms using recycling parameters $u = \{1, 2, 3, 4, 5\}$ and $s = 50$.

the effect of noisy instances. The larger u values cause the algorithm to focus on these noisy instances creating more mistakes.

Of course, the most important algorithm is VR-Combine since it always gives the fewest mistakes. As we explained in Section 5.2.2, voting techniques seem to have a preference for larger u values. Presumably, the larger u values increase the hypotheses diversity by causing more updates on the algorithms. We see this both in Table 7.17 where $u = 4$ gives the fewest mistakes, and Table 7.18 where $u = 5$ gives the fewest mistakes. In fact, the fewest mistakes on these learning problems comes from VR-Combine with $s = 50$ and $u = 4$ on the tracking basic algorithms. This algorithm makes only 6508 mistakes. This is an improvement of over 35% on the best basic algorithm. We believe further improvement is possible by using a wider range of basic algorithms along with multiple s and u values, all combined with a single VR-Combine algorithm.

7.6 Summary

In this chapter, we give the results of experiments on tracking problems that are generated by a shifting distribution. We show that the Tracking Unnormalized Winnow algorithm from Chapter 6 gives the fewest mistakes on several realistic data sets when compared to the other basic linear-threshold algorithms used in this thesis. We also give experiments that show our previous techniques of instance recycling and voting can also improve the performance of adversarial on-line algorithms when instances are generated by a shifting distribution. We improve upon the basic algorithms by over 35%. These techniques are efficient and can be applied to a wide range of adversarial on-line learning algorithms.

The recycling technique we use is identical to that found in Chapter 4. Our main voting technique is an application of the VR-Combine algorithm from Chapter 5. For learning tracking concepts, VR-Combine uses an instance based on the output of a set of algorithms. If there are v algorithms then the new instance will have v attributes. This information and the original label are input into the VR-Combine algorithm. The

VR-Combine algorithm attempts to minimize the number of mistakes by finding a hypothesis that combines the predictions of these algorithms.

Chapter 8

Delayed Label Feedback

In this chapter, we consider the problem of label feedback in on-line learning. According to the on-line model, after an algorithm makes a prediction, it is supposed to receive the label from the environment. For many practical problems, the algorithm may have to wait before it receives the label. This chapter builds on the work of [Mes05]. The problem of label delay with on-line learning was first brought to our attention by Davison [Dav01].

Consider spam email filtering. The filtering algorithm often allows the user to train the algorithm using labeled emails [AKC$^+$00]. In between training, many emails may arrive that need to be classified. Anytime successive predictions need to be made without receiving a label, it is a delayed learning problem. Another example is webpage prefetching. This is useful for speeding up the performance of low bandwidth Internet connections. Learning which links to preload is a useful optimization [PM96]; however, the label feedback might be delayed until it is determined that a prefetched webpage will not be used. As a final example, a doctor may want to predict health problems in a patient in order to start treatment as soon as possible. A more definitive test may be prescribed to confirm the diagnosis; this test provides delayed feedback.

To solve this problem, we propose the delayed model of on-line learning. This model is identical to the traditional on-line learning except that the environment can return the label feedback any number of trials after the arrival of the instance. This amounts to changing the last step of on-line learning to receive possibly multiple labels from the current or previous trials. For every on-line learning problem, there is a matching delayed learning problem that receives delayed labels.

In this chapter, we give multiple ways to transform a traditional on-line algorithm

to an algorithm that works with delayed labels. We give upper-bounds on the number of mistakes made by these algorithms, where the bounds are given as a function of the bounds for the original on-line algorithm. Our transformations are relatively simple and generally inexpensive. To allow the transformations to be as general as possible, we allow randomized on-line algorithms to be used in the transformations.

We assume two different techniques for instance generation. First, we assume the instances are generated by an adversary. This is similar to the adversary in Chapter 2 except the adversary is also given some control over the delays of the instances. Second, we assume the instances are generated by a distribution. This includes a shifting distribution as described in Chapter 7. In both cases, the bounds are robust; the bounds allow noisy instances that do not correspond to a target function [Lit89], and the bounds allow tracking a target function that is allowed to change over the trials [HL91, KPR91, Mes02]. We also give lower-bounds on these two instance generation techniques. We show these lower-bounds are close to our upper-bounds.

The choice of which technique to apply for converting an algorithm to the delayed model depends on the type of instance generation. When the instances correspond to a fixed target function, the order of the instance updates does not matter. Therefore, we want to update soon after the label arrives. When the instance corresponds to a shifting target function, we risk increasing the noise if we update with older instances. Therefore we must be careful to keep the updates in the same order as the original sequence of instances. The specific strategy will depend on whether an adversary or a shifting distribution is generating the instances.

The remainder of the chapter is organized as follows. The next section gives some notation needed for the delayed model. Section 8.2 gives algorithms and mistake bounds for the case where instances are generated by an adversary; this includes lower-bounds. In Section 8.3, algorithms and mistake bounds are given for problems with instances generated by a shifting distribution. Again, this includes lower-bounds.

Algorithm B
Initialization
$t \leftarrow 0$ is the trial number.
Initialize algorithm state $s \leftarrow s_0$.
Trials
$t \leftarrow t + 1$.
Instance: $\mathbf{x}_t$.
Prediction: $\hat{y}_t \leftarrow \text{Pred}(s, \mathbf{x}_t)$.
Update: Let y_t be the correct label.
$s \leftarrow \text{Update}(s, \mathbf{x}_t, y_t, \hat{y}_t)$.

Figure 8.1: Pseudo-code for on-line algorithm B.

8.1 Notation

All of our transformations take an existing traditional on-line algorithm and convert it to handle delayed instances. Assume algorithm B is a traditional on-line algorithm with pseudo-code given in Fig. 8.1. On trial t, the algorithm accepts instance $x_t \in X$ and returns a distribution $\hat{y}_t \in [0,1]^{|Y|}$ over the possible output labels. The algorithm predicts by sampling from this distribution.[1] The algorithm then receives feedback on the correct label $y_t \in Y$. It can use this information to update the current state of the algorithm to improve performance on future instances.

We assume that the only randomization in algorithm B comes from the sampling of the label distribution returned by the prediction procedure. The update procedure is deterministic. More information on this assumption can be found in Appendix F.

The transformed algorithms use the same procedures as algorithm B for updates and predictions. The prediction procedure of algorithm B accepts two parameters: the instance $\mathbf{x}_t$ for prediction and the current state of the algorithm, s. The state of the algorithm encodes the value of all the memory used by the algorithm that can have an effect on future predictions. This is different than the hypothesis the algorithm uses to make predictions. The current hypothesis is defined as the label prediction probabilities assigned to all instances. It is possible that the state of the algorithm may change without changing the current hypothesis. The initial state is represented as s_0.

[1] While it is common to just let $\hat{y}_t$ be the predicted label, a distribution is needed to help describe one of the later algorithm transformations.

The prediction procedure returns a probability distribution for the label. We use the notation $\hat{y}_t[i]$ to determine the probability that the predicted label is i on trial t. For a deterministic algorithm, a single label will have value 1.

The update procedure accepts four parameters: the state of the algorithm, the instance, the label returned by the environment, and the predicted label distribution. The update procedure returns two outputs: the new algorithm state and Boolean variable `change_state` that is `TRUE` if the algorithm state has changed because of the update and `FALSE` otherwise. We ignore the `change_state` variable if it is not used by a particular algorithm. Notice that, based on our definition, a traditional on-line algorithm can only change its state after an update.

Next, we need notation for the delayed labels returned by the environment. We use $y_{a,b}$ to refer to the label of an instance where the attributes arrive on trial a and the label arrives right before the start of trial b. Each trial, the algorithm gets a new vector of attributes, but each trial may get zero or more labels. Therefore, when we want to specify an instance, we refer to the trial where the attributes arrived. We define the delay of a particular instance, with label $y_{a,b}$, as $b-a$. Let k be the maximum delay over all instances. In traditional on-line learning, all instances have a delay of 1 and have labels of the form $y_{t,t+1}$. In delayed on-line learning, each instance has an arbitrary positive integer delay.

In the rest of the chapter, we use the following notation. A sequence of instances is a potentially infinite sequence where each item in the sequence is a tuple with two elements: a sequence of attributes and a label. Let $E[\text{Mist}(B, s)]$ be the expected number of mistakes algorithm B makes on s, a sequence of instances. Let $E[\text{C1}(B, s)]$ be the expected number of times B changes its current hypothesis on sequence s and let $E[\text{C2}(B, s)]$ be the expected number of times B changes its state on sequence s. The expectation is taken with respect to any randomization used by the algorithm. If the instances are generated by an adversary, let $E[\text{Mist}(B)]$ be the maximum expected mistakes made by algorithm B over a set of instance sequences. The particular set of sequences should be clear from context. If the instances are generated by a distribution, let $E[\text{Mist}(B)]$ be the expected number of mistakes based on sampling from the

distribution. In the case of a distribution, the expectation is taken with respect to the generation of instances and any randomization of the algorithm.

When giving lower-bounds, we often need to refer to the optimal algorithm in the traditional on-line setting. Let Opt_D be the algorithm that minimizes $E[\text{Mist}(B)]$ over all traditional deterministic algorithms B. Let Opt_R be the algorithm that minimizes $E[\text{Mist}(B)]$ over all traditional randomized algorithms B. Notice that the optimal algorithm can change depending on whether instances are generated by an adversary or a distribution.

8.2 Instances Generated by an Adversary

In this section, we give algorithms and bounds on mistakes when instances are generated by an adversary. First, we need to define what we mean by an adversary generating instances. Later we will allow the adversary to set delays for these instances.

Let S be a set of instance sequences. An adversary generates a sequence of instances by selecting any sequence from S. Typically this is a sequence that maximizes the number of mistakes for the particular learning algorithm being used. Therefore, we need to make assumptions about the types of sequences found in S in order to give a mistake bound for an algorithm. For our purposes, most of the assumptions about S will come from the traditional on-line algorithm that we are converting to the delayed setting. Therefore these assumptions are implicit in our analysis.

For example, S might contain all possible sequences that can be generated by any disjunction of at most k literals. One can even model noise by allowing S to contain any sequence that is correctly labeled by a disjunction with at most k literals after at most $2N$ of the binary attributes are changed. Notice that any sequence from S has an upper-bound on the number of mistakes using the algorithms and analysis from Chapter 2. In a similar way, we can generate sequence sets based on the shifting linear-threshold functions from Chapter 6.

However, for the purposes of this chapter, we make few explicit restrictions on the instance sequences contained in S. Therefore, these techniques apply to a wide range of

learning problems. The only assumption we need to make about S depends on whether the learning algorithm applies to fixed or shifting learning problems. Before we give these assumptions, we start with some useful definitions.

Definition 8.1 *A set of sequences S is closed under permutation if, for every sequence $s \in S$, every size of permutation of s is also contained in S.*

Definition 8.2 *An set of sequences S is closed under subsequence if, for every sequence $s \in S$, every subsequence of s is also contained in S.*

We call an adversary that is closed under permutation a permutation adversary. A permutation adversary can take any $s \in S$ and generate any sized permutation of s. This is a natural assumption for a fixed target function. For a particular sequence $s \in S$, all the instances correspond to a single target function, so an adversary should be able to mix the instances to maximize the difficulty of learning.

We call an adversary that is closed under subsequence a subsequence adversary. A subsequence adversary can take any $s \in S$ and generate every subsequence of s. This is a weaker assumption than the permutation adversary. It is needed when dealing with shifting target functions. For a shifting target function, a permutation would mix up the target functions. A subsequence adversary preserves the target function shifts but allows the adversary to skip over some of the instances in s to form a new sequence. Notice that every permutation adversary is a subsequence adversary.

The analysis in Chapter 2 applies to a permutation adversary. The analysis in Chapter 6 applies to a subsequence adversary. Given a problem where the adversary S is not a permutation or subsequence adversary, it is easy to convert the adversary by adding the necessary sequences to form S'. Since we are adding sequences any upper-bound on mistakes for S' also applies to S.

It is an open question as to what types of adversaries are useful for modeling learning problems. One purpose of an adversarial analysis is to show that an algorithm performs well even given unrealistic worst-case assumptions. Since adding sequences to create a permutation or subsequence adversary increases the mistake bound, these additions can be seen as continuing this tradition of worst-case analysis.

For delayed on-line learning, we need to let the adversary delay the label feedback. To make this as general as possible, we let the adversary pick the delay for an instance from an infinite sized multi-set $\mathcal{D}$ of positive numbers. For any sequence of instances $s \in S$, the adversary assigns a single element of $\mathcal{D}$ to each instance in s. Each element in $\mathcal{D}$ can only be used for a single instance in each sequence. Therefore, delayed adversary S maximizes the number of mistakes by considering all possible delays from $\mathcal{D}$ assigned to all sequences in S. For example, $\mathcal{D}$ may contain the number 5 an infinite number of times and the number 20 ten times. In this case, for every sequence $s \in S$ the adversary can give at most 10 instances a delay of 20; the remaining instances all have a delay of 5.

Let $d_i \in \mathbf{R} \cup \{\infty\}$ be the number of delays of length i in $\mathcal{D}$. Our bound is based on values of the various d_i; this allows us to model a wide range of problems. For example, in the medical problem explained earlier, each patient may take a different amount of time to get the lab test needed for the label; a few patients may never take the lab test and therefore have an infinite delay.

Throughout this chapter, we refer to both delayed and traditional sequences of instances. To make the distinction clear, a traditional sequence of instances is defined as a sequence of instances where each instance has a delay of 1. A delayed sequence of instances has delays set by the adversary from the multi-set $\mathcal{D}$. Often the type of sequence will be clear from context because a delayed sequence can not typically be used with a traditional on-line algorithm.

Before we give our main bound, we need a lemma to help us work with the delay multi-set $\mathcal{D}$. We want to place as many elements from $\mathcal{D}$ into a list L with the restriction that the first c numbers must be at least 1, the next c numbers must be at least 2, and so on where the mth block of c numbers must be at least m. We call this the ordered class selection problem.

Another way to view the ordered class selection problem is to create c lists where the value of each element in each list is at least equal to the index of the element in the list, and we want to maximize the sum of the sizes of all the lists.. For example, the list $[1, 2, 5, 5]$ would be valid, but the list $[1, 2, 2, 5, 5]$ would be invalid because the

third element is smaller than three.

Notice that if the adversary uses the values $[5, 5, 2, 1]$ as delays assigned to consecutive instances then the algorithm would not receive any labels for these instances until after predictions have been made on all four instances. Therefore, if the adversary knows the algorithm is likely to make a mistake on a particular instance and the algorithm repeats this instance for these four trials the adversary can cause several mistakes while only revealing information about a single instance.

Next we give a simple way to compute the maximum value of the ordered class selection problem. If one uses the greedy algorithm of always placing the smallest number remaining in $\mathcal{D}$ into the next position in the list then the total number of elements of value i that are in this greedy list is $r_i = \min\left(d_i, ic - \sum_{j=1}^{i-1} r_j\right)$.

Lemma 8.3 *Let $F(\mathcal{D}, c)$ be the maximum number of elements that can be placed from $\mathcal{D}$ in the ordered class selection problem. This maximum is obtained by the greedy algorithm, and $F(\mathcal{D}, c) = \sum_{j=1}^{\infty} r_j$.*

Proof We break the proof into two cases. First, assume that $F(\mathcal{D}, c) = \infty$. Based on the definition of the ordered class problem, for any number a, there must be an infinite number of elements in $\mathcal{D}$ greater than a. Therefore, the greedy algorithm will also generate an infinite list.

Second, assume that $F(\mathcal{D}, c)$ is finite. Let l_o be a list of elements that satisfy the ordered class selection problem with the number of elements in l_o equal to $F(\mathcal{D}, c)$. Let l_g be the list generated by the greedy algorithm. We will compare each element of l_g with l_o and show that the lists must have the same length.

Start at the beginning of each list and compare elements. If $l_o(0) = l_g(0)$ then go to the next element. If $l_o(0) > l_g(0)$ then find the next index, $i > 0$, in list l_o such that $l_o(i) = l_g(0)$. If index i exists then, in list l_o, swap values $l_o(0)$ and $l_o(i)$. This still gives a legal list. If there is no such index then the number of elements with value $l_g(0)$ used in list l_o must be less then $d_{l_g(0)}$. Therefore we can just assign $l_o(0)$ to value $l_g(0)$. The new l_o list is still a valid list and still has length $F(\mathcal{D}, c)$.

We can repeat this procedure for each pair of elements from list l_o and l_g. Let i_e

be the last element in list l_g. At this point both lists are identical up to index i_e. Any additional elements in l_o must have a value of at least $l_g(i_e)$ since otherwise the l_g list would not be greedy. However, if the additional elements have values of at least $l_g(i_e)$ then l_g would not end at index i_e. Therefore the new l_o list and l_g list must be the same length. Since the length of the modified l_o has not changed from the original length, the length of l_g is $F(\mathcal{D}, c)$. Based on the greedy algorithm, the length of l_g is equal to $\sum_{j=1}^{\infty} r_j$. This proves the lemma. ■

The value of $F(\mathcal{D}, c)$ is important for our adversarial mistake bounds. As previously mentioned, one can view the $F(\mathcal{D}, c)$ list as built from c separate lists where each list item must have a value greater than its index. These lists can be used as delays that are assigned to consecutive instances such that no label for these instances is returned until after all the instances have been received for prediction. As we will see, the adversary tries to force a mistake on all the instances in a list while only allowing one of the instances to help lower the mistake bound.

Another way of viewing the $F(\mathcal{D}, c)$ function is to consider the average size of a list. Let $\bar{k}(c) = F(\mathcal{D}, c)/c$. The $\bar{k}$ function gives the average size of the c delay lists. As we will see, the $\bar{k}$ function is, in some sense, the effective delay of the on-line learning problem.

Note that the $F(\mathcal{D}, c)$ function is monotonically increasing with c while $\bar{k}(c)$ is monotonically decreasing with c. Also notice that adding a few large outlier delays does not have much of an influence on $F(\mathcal{D}, c)$. No matter how large the delay, each outlier can increase $F(\mathcal{D}, c)$ by at most one.

8.2.1 Fixed Target Function

Assume B is a traditional on-line algorithm. Our first transformation creates the delayed on-line algorithm $OD2$-B. This algorithm just updates any instance as soon as the label becomes available. The pseudo-code for $OD2$-B is given in Fig. 8.2.

The computational cost of the $OD2$-B algorithm is similar to the cost of the B

Algorithm *OD2-B*
Initialization
$t \leftarrow 0$ is the trial number.
Initialize empty hash table H that stores new instances.
Initialize algorithm to state $s \leftarrow s_0$.
Trials
$t \leftarrow t + 1$.
Instance: Store $\mathbf{x}_t$ in H with key t.
Prediction: $\hat{y}_t \leftarrow \text{Pred}(s, \mathbf{x}_t)$.
Update:
For all returned labels $y_{a,t}$
Remove x_a from H.
$\hat{y} \leftarrow \text{Pred}(s, x_a)$.
$s \leftarrow \text{Update}(s, x_a, y_{a,t}, \hat{y})$.

Figure 8.2: Pseudo-code for delayed on-line algorithm *OD2-B*.

algorithm. The number of updates is at most the same as algorithm B and the number of predictions is at most double. The *OD2-B* algorithm needs extra storage for at most $F(\mathcal{D}, 1)$ instances since this is the maximum number of instances that have arrived for prediction but have not yet received their labels.

The next lemma gives a mistake bound and proof for the *OD2-B* algorithm. Recall that $\text{C1}(B, s)$ is a random variable for the number of times algorithm B changes its hypothesis on instance sequence s.

Lemma 8.4 *Assume B is a traditional on-line algorithm, and s is a sequence of delayed instances generated by a permutation adversary. There exists a traditional instance sequence u_s which is a permutation of s such that the expected number of mistakes of the OD2-B algorithm on sequence s is at most $E[\mathit{Mist}(B, u_s)] + E[F(\mathcal{D}, \mathit{C1}(B, u_s))] - E[\mathit{C1}(B, u_s)]$ in the delayed on-line model.*

Proof Define an internal mistake as an incorrect prediction made during an update. These are not real mistakes as they only occur during the update procedure. However, we can bound the expected number of internal mistakes and use this to help bound the number of real mistakes.

Let u_s be the sequence of instances from s arranged in order of the updates in algorithm *OD2-B* with all the delays set to 1. If these instances are passed to algorithm

B, the expected number of mistakes is $E[\text{Mist}(B, u_s)]$. Notice that any mistakes made by B on u_s corresponds to an internal mistake on sequence s for algorithm $OD2\text{-}B$. Therefore the expected number of internal mistakes is $E[\text{Mist}(B, u_s)]$.

Next, it is useful to partition the sequence s into two sets. Let Q_1 be the set of instances $\mathbf{x}_t$ such that, when $OD2\text{-}B$ updates the label $y_{t,t+k}$, the hypothesis at trial $t+k$ has not changed since trial t. Let Q_2 be the set of all other instances.

For any instance $\mathbf{x}_t$ from Q_1, the probability of a mistake on trial t is the same as the probability of an internal mistake when the label arrives. Since the expected number of internal mistakes is $E[\text{Mist}(B, u_s)]$, the expected number of mistakes on instances from Q_1 is at most $E[\text{Mist}(B, u_s)]$.

Next consider Q_2. There is a limit on the number of instances in Q_2 based on the number of times the hypothesis changes and the number of instances with specific delay values. This number is primarily determined by the solution to the multi-set problem in Lemma 8.3. However, for each hypothesis change at least one of the delayed instances from the multi-set solution must cause the update that changes the hypothesis. Therefore the expected number of elements in Q_2 is at most $E[F(\mathcal{D}, \text{C1}(B, u_s))] - E[\text{C1}(B, u_s))]$. Since each instance in Q_2 can cause at most one mistake, this proves the theorem. ■

In order to get a good bound, we need to use a B algorithm that makes a small number of mistakes and changes its hypothesis infrequently. Fortunately, deterministic mistake-driven algorithms fulfill these criteria. A deterministic mistake-driven algorithm is an algorithm that only updates its state when it makes a mistake [Lit88]. In Appendix F, we define a transformation that takes a traditional on-line algorithm B and converts it into a mistake-driven algorithm $MD\text{-}B$. We show that $MD\text{-}B$ does not increase the mistake bound of algorithm B when instances are generated by a subsequence adversary.

To handle randomized algorithms, we use the fact that any randomized learning

algorithm can be converted to a deterministic learning algorithm with a similar mistake bound. On every trial, this new deterministic algorithm just predicts the highest probability label from the randomized algorithm. The deterministic algorithm makes at most double the expected number of mistakes of the randomized algorithm [AW95]. Given a learning algorithm B, we call *DR-B* the derandomized learning algorithm. See Appendix F for more details.

Theorem 8.5 *Assume B is an on-line algorithm and that instances are generated by a permutation adversary. The number of mistakes made by the OD2-MD-DR-B algorithm is at most $Mist(B)\bar{k}(Mist(B))$ when B is deterministic and $2E[Mist(B)]\bar{k}(2E[Mist(B)])$ when B is randomized.*

Proof Assume B is deterministic. In this case, algorithm *DR-B* is identical to algorithm B. Based on Theorem F.2, $\text{Mist}(MD\text{-}B) = \text{Mist}(B)$. Because *MD-B* is mistake-driven, it only changes its hypothesis on a mistake. Therefore, using Lemma 8.4, the maximum number of mistakes made by *OD2-MD-DR-B* is $\text{Mist}(B)\bar{k}(\text{Mist}(B))$.

Assume B is randomized and $E[\text{Mist}(B)] = M$. Using the derandomized algorithm, we get $\text{Mist}(DR\text{-}B) \leq 2M$. Theorem F.2 shows that $\text{Mist}(MD\text{-}DR\text{-}B) \leq 2M$. Since algorithm *MD-DR-B* is mistake-driven it can only change its hypothesis on mistakes. Therefore, using Lemma 8.4, the maximum number of mistakes made by *OD2-MD-DR-B* is $2E[\text{Mist}(B)]\bar{k}(2E[\text{Mist}(B)])$. ■

This theorem shows that the number of mistakes made by algorithm *OD2-MD-DR-B* versus B could grow by a factor of at most $\bar{k}(\text{Mist}(B))$ when B is deterministic and $2\bar{k}(2E[\text{Mist}(B)])$ when B is randomized. Using this technique, randomization does not help because the technique removes any randomization. Unfortunately, while algorithm *OD2-MD-B* or *OD2-B* may improve the mistake bound, we do not currently have a useful way to analyze these algorithms. The principle problem is that we need a way to bound the number of times *OD2-MD-B* and *OD2-B* change their hypotheses. We discuss this issue more in Section 8.2.3

Algorithm *OD*1-*B*

Initialization

$t \leftarrow 0$ is the trial number.
`last` $\leftarrow 0$ is the last instance used for an update.
Initialize empty hash table H that stores new instances.
Initialize empty stack U that stores instances ready for updates.
Initialize algorithm to state $s \leftarrow s_0$.

Trials

$t \leftarrow t + 1$.
Instance: Store $\mathbf{x}_t$ in H with key t.
Prediction: $\hat{y}_t \leftarrow \text{Pred}(s, \mathbf{x}_t)$.
Update:
For all returned labels $y_{a,t}$
Remove x_a from H.
If $a >$ `last` add instance $(a, x_a, y_{a,t})$ to U in sorted order based on a.
For $i = 1$ to $|U|$
$(a, x_a, y_{a,t}) \leftarrow \text{pop}(U)$.
$\hat{y} \leftarrow \text{Pred}(s, x_a)$.
$(s,$ `new_state`$) \leftarrow \text{Update}(s, x_a, y_{a,t}, \hat{y})$.
If `new_state` $= 1$ or $\hat{y}[y_{a,t}] \neq 1$ then
`last` $\leftarrow a$.
Periodically remove all instances from H older than `last`.

Figure 8.3: Pseudo-code for delayed on-line algorithm *OD*1-*B*.

8.2.2 Shifting Target Function

For problems that have shifting target functions, we cannot assume a permutation adversary. In this case, we assume the instances are generated by a subsequence adversary. For solving these problems, we covert a traditional on-line algorithm B into *OD*1-*B* using our second transformation.[2] This algorithm is similar to algorithm B except that it potentially skips some of the updates. The pseudo-code for *OD*1-*B* is in Fig. 8.3.

*OD*1-*B* keeps track of a `last` trial and only performs updates using instances that are more recent than `last`. After the algorithm performs an update, if the update either changes the state of the algorithm, or if the update is based on an instance that could have caused an internal mistake[3] then the algorithm increases `last` to the trial

[2]The out of order naming is based on previous publications [Mes05].

[3]An internal mistake is not a real mistake since it does not occur during the original prediction with the instance.

of the instance used for the update. This ensures that the changes to the algorithm occur in the same order as the instance arrival times. Changes occurring out of order can cause problems when the concept is shifting.

The computational cost of the $OD1$-B algorithm is similar to the cost of algorithm B. The number of updates is at most the same as algorithm B and the number of predictions is at most double. There is an extra cost based on sorting the instances in U. This cost depends on the number of labels returned per trial. Let γ be the maximum number of label returned during a trial. Using merge sort gives an amortized cost of at most $O(\ln(\gamma))$ per trial. Because $F(\mathcal{D}, 1)$ is the maximum number of instances that can be simultaneously waiting for their labels, we have that $\gamma \leq F(\mathcal{D}, 1) \leq k$.

The $OD1$-B algorithm needs extra storage for at most $F(\mathcal{D}, 1)$ instances. By keeping space for $2F(\mathcal{D}, 1)$ instances, the algorithm can periodically remove any old instances that were skipped for updates with only a constant amortized increase in the cost per trial.

The next lemma gives an upper-bound on the number of mistakes for the $OD1$-B algorithm. It is similar to the result in Lemma 8.4. Recall that $\mathrm{C2}(B, s)$ is a random variable for the number of times algorithm B changes its state on instance sequence s.

Lemma 8.6 *Assume B is a traditional on-line algorithm, and assume s is a delayed sequence of instances generated by a subsequence adversary. There exists an instance sequence u_s which is a subsequence of s such that the expected number of mistakes of the OD1-B algorithm is at most $E[Mist(B, u_s)] + E[F(\mathcal{D}, C2(B, u_s))] - E[C2(B, u_s)]$ in the delayed on-line model.*

Proof Consider all the instances that change the variable `last` in algorithm $OD1$-B. Let u_s be this subsequence of instances with all the delays set to 1. These instances are in trial order. Notice that the algorithm states for $OD1$-B on s are related to the states used by algorithm B on u_s. Every update for $OD1$-B on s maps to an identical update by algorithm B on u_s.

It is useful to partition the sequence s into two sets. Let Q_1 be the set of instances $\mathbf{x}_t$ such that, when $OD1$-B updates the label $y_{t,t+k}$, the state at trial $t + k$ has not

changed since trial t. Let Q_2 be all other instances. We can divide the instances from Q_1 into two groups. Let g_1 be the instances from Q_1 that are in u_s. Let g_2 be all other instances in Q_1. Notice that an instance in g_2 did not cause a mistake and did not change the state of $OD1$-B.

Assume that x is an instance from g_1. The probability of a mistake by $OD1$-B on x must equal the probability of a mistake in the equivalent instance from u_s with algorithm B since the state when the label arrived is the same as the state when the instance arrived. For the instances in g_2, the $OD1$-B algorithm must have a zero probability of making a mistake otherwise the instance would be in u_s. Therefore the expected number of mistakes by algorithm $OD1$-B for instances from Q_1 must be at most $E[\text{Mist}(B, u_s)]$.

Next consider Q_2. There is a limit on the number of instances in Q_2 based on the number of times the state changes and the number of instances with specific delay values. This number is primarily determined by the solution to the multi-set problem in lemma 8.3. However, for each state change at least one of the delayed instances from the multi-set solution must cause the update that changes the state. Therefore the expected number of elements in Q_2 is at most $E[F(\mathcal{D}, \text{C2}(B, u_s))] - E[\text{C2}(B, u_s))]$. Since each instance in Q_2 can cause at most one mistake, this proves the theorem. ■

Next, we use the same technique as we used for algorithm $OD2$-B. We derandomize the B algorithm and convert it to a mistake-driven form.

Theorem 8.7 *Assume B is an on-line algorithm and that instances are generated by a subsequence adversary. The number of mistakes made by the OD1-MD-DR-B algorithm is less than or equal to $Mist(B)\bar{k}(Mist(B))$ when B is deterministic and $2E[Mist(B)]\bar{k}(2E[Mist(B)])$ when B is randomized.*

Proof Assume B is deterministic. In this case, algorithm DR-B is identical to algorithm B. Based on Theorem F.2, $\text{Mist}(MD\text{-}B) = \text{Mist}(B)$. Because MD-B is mistake-driven, it only changes its state on a mistake. Therefore, using Lemma 8.6, the maximum number of mistakes made by $OD2$-MD-DR-B is $\text{Mist}(B)\bar{k}(\text{Mist}(B))$.

Assume B is randomized and $E[\text{Mist}(B)] = M$. Using the derandomized algorithm, $\text{Mist}(DR\text{-}B) \leq 2M$. Theorem F.2 shows that $\text{Mist}(MD\text{-}DR\text{-}B) \leq 2M$. Since algorithm *MD-DR-B* is mistake-driven, it can only change its state on mistakes. Therefore, using Lemma 8.6, the maximum number of mistakes made by *OD*2-*MD-DR-B* is $2E[\text{Mist}(B)]\bar{k}(2E[\text{Mist}(B)])$. ■

Notice that the *OD*1 transformation is more general than *OD*2 as it can handle both fixed and shifting target concepts and that it has the same mistake bound. Still the *OD*2 transformation is useful. When dealing with something more realistic than a worst-case adversary, *OD*2-*B* might lower the number of mistakes since it attempts to update on all of its instances instead of skipping updates. These extra updates may improve performance in practice.

8.2.3 Lower-bounds

A natural question is whether a different transformation can make fewer mistakes. To help answer this question, we define another type of adversary. This new type of adversary implicitly removes noisy instances from a set of sequences. We start with another definition on sequences.

Definition 8.8 *A set of sequences S is closed under repetition if for every element $s = (z_1, z_2, \ldots) \in S$ and every legal index j of s, the sequence $s' = (z_1, z_2, \ldots, z_j, z_j, z_{j+1}, \ldots)$ is also contained in S.*

We call an adversary that is closed under repetition a repetition adversary. As an example, consider the set of sequences $S = \{12, 3\}$. To convert this into a repetition adversary, add all sequences of the form 1^+2^+ and 3^+, where the $+$ operator is a standard operator for repetition in regular expressions.

A repetition adversary can be combined with the previous adversary definitions. The previous example can be made into a repetition, permutation adversary by adding all sequences of the form 1^*2^*, 2^*1^*, and 3^*. The Kleene star operator allows zero or more repetitions and is commonly used in regular expressions.

For a fixed target function, if there are instances that cannot be represented by a legal target function then a repetition adversary must have an infinite mistake bound. In order to generate useful mistake bounds, the repetition adversary can not have noisy instances. For a shifting target function, the noisy instances can always be represented by using more shifts of the target function. Therefore, with a repetition adversary, any instances inconsistent with the current value of the target function can be incorporated into a new shift of the target function. This removes any noisy instances.

We want our lower-bounds to allow some noisy instances, but we need to precisely control how much noise is allowed by the adversary. The easiest way is to give a procedure to add a specified amount of noise to an adversary and then use this procedure on a repetition adversary. To create label noise in adversary S, copy any sequence in S and change at most A labels. This new sequence is then added to S. We formalize this in the following definition.

Definition 8.9 *Assume S is an adversary and let $s \in S$. For every subsequence s' of s where $|s'| = A$, S_A contains copies of s using all possible labels for the instances in subsequence s'.*

Our lower-bound proof depends on being able to use a delayed on-line learning algorithm to solve a traditional on-line learning problem. The essence of the proof is to show that the delayed on-line learning algorithm can not be used to create a traditional on-line algorithm that is better than optimal. We start with an algorithm transformation that converts a delayed on-line learning algorithm C into a traditional on-line algorithm $DO\text{-}C(k)$. We use the k notation because we have a different transformation for each value of k.

The $DO\text{-}C(k)$ algorithm solves a traditional on-line problem. The $DO\text{-}C(k)$ algorithm receives instance $\mathbf{x}_1$ and creates k copies of the instance, $(\mathbf{x}'_1 = \mathbf{x}_1, \ldots, \mathbf{x}'_k = \mathbf{x}_1)$. These k copies are used as the first k instances of algorithm C. When the label y_1 is received, it is copied to k labels for instances x'_1 to x'_k. The labels must be spaced to make sure that no label arrives before algorithm C has made predictions on all k instances. This continues for every trial of $DO\text{-}C(k)$ algorithm, creating k instances and

labels for input into the C algorithm. The prediction for $\mathbf{x}_t$ by the $DO\text{-}C(k)$ algorithm is just the random majority prediction of the C algorithm over the k identical instances. By random, we mean the $DO\text{-}C(k)$ algorithm predicts label y with a probability equal to the ratio of the k identical instances that predict label y with algorithm C.

Recall that $\bar{k}(c) = F(\mathcal{D}, c)/c$. If c is infinite, we define $\bar{k}(c)$ in terms of the limit as c approaches infinity. This is always defined as long as there is an upper-bound on the size of the delays in $\mathcal{D}$ and corresponds to the first delay value with an infinite number of entries in the delay multi-set. We use the $\bar{k}(c)$ function in the following lemma.

Lemma 8.10 *Let C be a delayed on-line learning algorithm. Assume s is a traditional sequence of instances generated by a repetition adversary and that $\mathcal{D}$ is the multi-set of delays for the delayed version of this learning problem. Assume $\hat{k} \leq \lfloor \bar{k}(|s|) \rfloor$. When running algorithm $DO\text{-}C(\hat{k})$ on sequence s, $E[Mist(DO\text{-}C(\hat{k}), s)] \leq E[Mist(C)]/\hat{k}$.*

Proof Create a new delayed sequence s' from s that can be used by the C algorithm to duplicate the behavior of the $DO\text{-}C(\hat{k})$ algorithm. The delays in D are sufficient to create this sequence based on the definition of $\hat{k}$. The repeated instances are allowed since the instances are generated by a repetition adversary.

Running algorithm $DO\text{-}C(\hat{k})$ on s is related to running algorithm C on s'. Every instance from s' that is predicted incorrectly increases the probability that $DO\text{-}C(\hat{k})$ will make a mistake on that instance by $1/\hat{k}$ because of the random majority algorithm. Therefore, for all sequences s generated by the adversary, $E[\text{Mist}(DO\text{-}C(\hat{k}), s)] = E[\text{Mist}(C, s')]/\hat{k} \leq E[\text{Mist}(C)]/\hat{k}$. ■

Lemma 8.10 implies how the bound for a delayed on-line learning algorithm must grow with $\hat{k}$. We flesh out that implication with two lower-bound theorems. The first theorem deals with the an arbitrary multi-set D. The second theorem gives a slightly better bound for the a multi-set that has an infinite number of delays greater than a particular value.

Theorem 8.11 *Assume $Mist(Opt_D) = M$ for a traditional on-line learning problem with instances generated by a repetition adversary S. Let S_A be the repetition adversary*

S with A added noise. For any delayed learning algorithm C using instance from S_A and delays from multi-set $\mathcal{D}$, $E[Mist(C)] \geq \lfloor \bar{k}(M) \rfloor M/2 + A$.

Proof First we consider a bound based on adversary S. Let $\hat{k} = \lfloor \bar{k}(M) \rfloor$. First we convert algorithm C to the traditional on-line algorithm $DO\text{-}C(\hat{k})$. Based on Lemma F.4, there must exist a sequence s of length M such that $\text{Mist}(DO\text{-}C(\hat{k}), s) \geq M/2$. Based on Lemma 8.10, $E[\text{Mist}(DO\text{-}C(\hat{k}), s)] \leq E[\text{Mist}(C)]/\hat{k}$. Combining these, we conclude that $E[\text{Mist}(C)] \geq \hat{k}M/2$.

Using adversary S_A, we can add new instances to the end of sequence s. We add $2A+1$ instances where each instance has the same attributes. This instance is identical to one of the instances that has already appeared in sequence s. Call this instance x. The correct label for x has already been set in sequence s. Next, we randomly select A of these $2A + 1$ instances and change their labels. There is no way for algorithm C to determine which of the $2A + 1$ instances have the noisy label. To minimize the number of mistakes algorithm C must predict with the original label on all $2A + 1$ instances. This adds an extra A mistakes. ■

Theorem 8.12 *Assume $Mist(Opt_R) = M$ for a traditional on-line learning problem with instances generated by a repetition adversary S. Let S_A be the repetition adversary S with A added noise. For any delayed learning algorithm C using instance from S_A, when $\mathcal{D}$ has an infinite number of delays of value $\hat{k}$ or greater, $E[Mist(C)] \geq \hat{k}M + A$.*

Proof First, we consider a bound based on adversary S. We use Lemma 8.10 to create a traditional on-line algorithm $DO\text{-}C(\hat{k})$ where for all sequences s, $E[\text{Mist}(DO\text{-}C(\hat{k}), s)] \leq E[\text{Mist}(C)]/\hat{k}$. Since there must exist an s where $M \leq E[\text{Mist}(DO\text{-}C(\hat{k}), s)]$, we conclude that $M \leq E[\text{Mist}(C)]/\hat{k}$.

Using adversary S_A, we can follow the same strategy as Theorem 8.11. After we receive the correct label for some instance x generated by adversary S, we create $2A+1$ copies of this instance to force A mistakes due to noise. ■

These two bounds are similar. Based on the technique of derandomizing an algorithm covered in Appendix F, $E[\text{Mist}(Opt_R)] \leq \text{Mist}(Opt_D) \leq 2E[\text{Mist}(Opt_R)]$. Therefore, Theorem 8.11 is at most a factor of two lower than the specialized case in Theorem 8.12. Also, the algorithms $OD1\text{-}MD\text{-}Opt_D$ and $OD2\text{-}MD\text{-}Opt_D$ give upper-bounds on mistakes that are at most a factor of two worse than Theorem 8.11 and Theorem 8.12 when the label noise is zero.

There are learning problems that show Theorem 8.11 and Theorem 8.12 are tight. Consider a problem that has l instances, and assume the learning problem allows all 2^l possible binary target functions. The adversary is not allowed to shift the target function and there is no noise, so every instance must be correctly classified by the target function. Call this learning problem $H(l)$.

Define algorithm L_R as follows. The algorithm keeps a label count for each instance. Every time L_R receives an instance, the algorithm predicts according to the majority label for that instance. If there is a tie then the algorithm predicts -1 with probability 1/2 and 1 with probability 1/2. Define algorithm L_D in a similar way. The only difference is that if the label count is tied, algorithm L_D always predicts 1. Both the L_D and L_R algorithms only change their state when they receive a new label.

The L_R algorithm is an optimal randomized algorithm for the traditional on-line $H(l)$ problem because for each instance the adversary can choose either label. If the algorithm predicts one label with higher probability, the adversary will always choose the other label. Once all the instances have been seen, the target function has been determined. Therefore the expected number of mistakes for L_R is $l/2$. The L_D algorithm is optimal based on the same reasoning for the deterministic case. It makes at most l mistakes.

Assume the delayed version of problem $H(l)$ has a maximum delay of k and that an infinite number of instances have delay k. Both Theorem 8.11 and Theorem 8.12 imply that any algorithm must make at least $kl/2$ mistakes on some sequence of instances. Algorithm $OD2\text{-}L_R$ learns the correct label of each instance within k trials. Therefore, because the algorithm has a 1/2 probability of getting an unknown instance wrong, $OD2\text{-}L_R$ expects to make at most $kl/2$ mistakes. This shows the lower-bounds are

tight. The upper-bound in Lemma 8.4 gives a bound of $kl - l/2$ for algorithm $OD2\text{-}L_R$. This shows that are upper-bounds are off by a factor of close to two even when we do not use derandomization.

Another issue with our lower-bounds is the effect of label noise. Both lower-bounds only increase the number of mistakes by A where A is the number of label flips allowed on the instances by an adversary. However, in Appendix F we show that, on the previous $H(l)$ problem with A label flips, the $OD2\text{-}L_D$ algorithm makes at most $kl+2A$ mistakes. This shows our lower-bound is close to tight.

Unfortunately, the upper-bounds in Theorem 8.5 and Theorem 8.7 give a much weaker bound with respect to noise. If the upper-bound of a traditional deterministic algorithm B increases from M to $M + 2A$ with the addition of A label flips then the upper-bound of $OD1\text{-}MD\text{-}DR\text{-}B$ increases from $M\bar{k}(M)$ to $(M + 2A)\bar{k}(M + 2A)$. The tightness of this upper-bound may seem questionable given that the $OD2\text{-}L_D$ algorithm only increases its mistake bound by $2A$.

In Appendix F, we show that there exists a sequence of instances that causes algorithm $OD2\text{-}MD\text{-}L_D$ to make $kl + (k + 1)A$ mistakes. Therefore, our upper-bound is close to tight for some algorithm/problem combinations. This example also shows that, unlike the traditional on-line model, the mistake-driven modification is not always optimal for delayed learning with subsequence adversaries. Therefore, given a traditional algorithm B, it may be beneficial to use algorithms $OD1\text{-}B$ and $OD2\text{-}B$ with delayed learning problems when dealing with noise.

8.3 Instances Generated by a Distribution

In this section, we give algorithm transformations for delayed on-line learning when the instances are generated by a shifting distribution. This shifting includes both the target function and the probability of a particular instance. A shifting distribution is a realistic model for many on-line learning problems. Often the learning environment is not trying to maximize the number of mistakes; instead, the instances are generated by a distribution that is infrequently or slowly changing. For example, in our hypothetical

medical learning problem, the population may be slowly changing dietary habits which could effect the target function.

Let D be a distribution over $X \times Y$. Let $\mathcal{W}$ be a sequence, $(D_1, D_2, \ldots)$, of these distributions. An instance is generated at trial t by sampling an instance from distribution D_t. Let Υ be a set of sequences of these distributions. The goal of the learning algorithm is to minimize the expected number of mistakes for a sequence of distributions selected from Υ. Our bounds will depend on how much the distribution changes over the selected sequence. We call this the adversarial distribution instance generation model.

To make our techniques work, we need to make a similar assumption as Section 8.2.

Definition 8.13 *A distribution adversary Υ is a subsequence distribution adversary if Υ is closed under subsequence.*

This definition says that a subsequence distribution adversary can pick any subsequence of distributions to generate the instances. This is a worst-case assumption, as any adversary can be converted to a subsequence distribution adversary by adding the missing sequences and possibly increasing the upper bound on the number of mistakes.

For the delayed on-line model, the environment selects the delays of the instances from a multi-set $\mathcal{D}$. We assume the environment must select a delay before picking an instance from the distribution. Allowing the environment to select the delays is again a worst-case assumption. It is possible to refine the analysis to allow a distribution to generate the delays, but this does not have a large effect on the mistake bound.

A key component of the bound is the total amount the distribution changes over the trials. We use variational distance to measure the change between two distributions [DGL91]. Given discrete distributions D_1 and D_2 over sample space H, let the probability of an element x be $p_1(x)$ for D_1 and $p_2(x)$ for D_2. The variational distance is $V(D_1, D_2) = \frac{1}{2} \sum_{x \in X} |p_1(x) - p_2(x)|$. Using a sequence of distributions $\mathcal{W}$, the total variational distance over all trials is $\Psi(\mathcal{W}) = \sum_{i=1}^{\infty} V(D_{i+1}, D_i)$. We define $\Psi = \max_{\mathcal{W} \in \Upsilon} \Psi(\mathcal{W})$. These definitions generalize to arbitrary probability measures.

Algorithm $\overline{OD2}$-B

Initialization

$t \leftarrow 0$ is the trial number.
Initialize empty hash table H that stores new instances.
Initialize empty stack U that stores instances ready for updates.
Initialize algorithm to state $s \leftarrow s_0$.

Trials

$t \leftarrow t + 1$.
Instance: Store $\mathbf{x}_t$ in H with key t.
Prediction: $\hat{y}_t \leftarrow \text{Pred}(s, \mathbf{x}_t)$.
Update:
For all returned labels $y_{a,t}$
Remove x_a from H.
Add instance $(x_a, y_{a,t})$ to U.
If $|U| > 0$ then
$(x_a, y_{a,t}) \leftarrow \text{pop}(U)$.
$\hat{y} \leftarrow \text{Pred}(s, x_a)$.
$s \leftarrow \text{Update}(s, x_a, y_{a,t}, \hat{y})$.

Figure 8.4: Pseudo-code for delayed on-line algorithm $\overline{OD2}$-B.

8.3.1 Fixed Distribution

In this section, we transform a traditional on-line algorithm B to perform well in the delayed on-line model when instances are generated by a fixed distribution. We call the transformed algorithm $\overline{OD2}$-B. It is related to algorithm $OD2$-B except that it uses at most one instance per trial for an update. If multiple labels arrive at the start of the trial, only one can be used for the update. The remaining labels must wait for another trial to perform their update. The pseudo-code for $\overline{OD2}$-B is given in Fig. 8.4.

The computational cost of the $\overline{OD2}$-B algorithm is the same as $OD2$-B. The number of updates is at most the same as algorithm B and the number of predictions is at most double. The $\overline{OD2}$-B algorithm needs extra storage for at most $F(\mathcal{D}, 1)$ instances since this is the maximum number of instances that have arrived for prediction but have not yet received their labels.

Before we give the upper mistake bound for the $\overline{OD2}$-B algorithm, we give a lemma that upper-bounds the number of trials where no update occurs for algorithm $\overline{OD2}$-B.

These are trials where no attribute label pair is available for an update.

Lemma 8.14 *There are at most $F(\mathcal{D}, 1)$ trials where no update occurs for algorithm $\overline{OD2}$-B.*

Proof Assume that trial t does not perform an update and that it is the $F(\mathcal{D}, 1) + 1$ trial that does not perform an update. Therefore, a total of $t - (F(\mathcal{D}, 1) + 1)$ labels have been used for updates. Because trial t does not perform an update, there are no labels waiting to be used in the stack. This gives a total of $F(\mathcal{D}, 1) + 1$ labels that have not yet been received by trial t. The maximum number of instances that have arrived but not yet received their label is $F(\mathcal{D}, 1)$ based on the definition of $F(\mathcal{D}, 1)$. This is a contradiction. ■

Theorem 8.15 *Assume B is a traditional on-line algorithm where $E[Mist(B)] = M$ when instances are generated from a fixed distribution D. The $\overline{OD2}$-B algorithm expects to make at most $F(\mathcal{D}, 1) - 1 + M$ mistakes when instances are generated by a fixed distribution in the delayed distribution model.*

Proof Let s be the sequence of instances seen by algorithm $\overline{OD2}$-B. Let u_s be the subsequence of instances that cause updates in algorithm $\overline{OD2}$-B based on the order of update. If we use sequence u_s on algorithm B with a delay of 1 given to each instance then we expect to make at most M mistakes. This is because sequence u_s is based on the same distribution that generated the original sequence.

For algorithm B on sequence u_s, let h_i be the hypothesis that is used for prediction in trial i, and let X_i be a random variable that is 1 if algorithm B makes a mistake on trial i and 0 otherwise. Let Y_i be a random variable that is 1 if algorithm $\overline{OD2}$-B makes a mistake on trial i and 0 otherwise.

The hypotheses used by $\overline{OD2}$-B on sequence s are the same as the hypotheses used by algorithm B on u_s. The only difference is that sequence s does not always perform an update and may have to use a hypothesis on more than one trial. The total number

of trials with repeated hypotheses is at most $F(\mathcal{D}, 1) - 1$ based on Lemma 8.14 and the fact that the first trial does not cause an update but does use a new hypothesis.

Since the hypotheses are the same for $\overline{OD2}$-B and algorithm B, their expected accuracies will be the same since they both are used for predictions on instances drawn from distribution D. We already know that algorithm B makes at most M mistakes, therefore algorithm $\overline{OD2}$-B makes at most $M + F(\mathcal{D}, 1) - 1$ mistakes since at worst it can make a mistake on every repeated hypothesis. ■

The above result can be improved slightly by predicting with an unbiased coin flip every time the algorithm is forced to use a hypothesis that missed a chance at an update. In this case, the bound becomes $(F(\mathcal{D}, 1) - 1)/2 + M$ expected mistakes. However, if the repeated hypothesis has better than 1/2 accuracy, it is better to use the repeated hypothesis for prediction. Heuristically, if the algorithm has already been learning for many trials, the repeated hypothesis is probably better than a random coin flip. This is especially likely if the techniques from Chapter 5 are used to increase the accuracy of the predictions.

Along the same lines, one can apply algorithm $OD2$-B to solve the fixed distribution problem. Remember that $OD2$-B updates as soon as it gets a label instead of updating at most once a trial. This has the potential advantage of using the label information as quickly as possible to improve the current hypothesis. The main disadvantage is that we can not bound the accuracy of the subset of hypotheses used for prediction by $OD2$-B in terms of the hypotheses used by algorithm B. A perverse algorithm B could predict with a coin flip every 10 trials while otherwise using the optimal hypothesis. The adversary could use the delays to force $OD2$-B to always predict with a coin flip.

One possible assumption that allows Theorem 8.15 to apply to algorithm $OD2$-B is if we assume algorithm B predicts with hypotheses that are *expected* to monotonically increase in accuracy. This accuracy is measured over the distribution of hypotheses generated by selecting instances from distribution D. This is a desirable property for any learning algorithm that uses instances generated by fixed distributions. We are

unaware of any proofs that show this to be true for the linear-threshold algorithms used in this thesis; however it must be true for optimal Bayesian algorithms such as Naive Bayes [DH73].

8.3.2 Shifting Distribution

Our next algorithm works with the general case of a shifting distribution. Given a traditional on-line algorithm B, we call the transformed algorithm $\overline{OD3}$-$B(k')$. The $\overline{OD3}$-$B(k')$ algorithm does an update with every instance, and it does the updates in trial order, the same order as B. In other words, $\overline{OD3}$-$B(k')$ can only update the instance from trial $t+1$ after the update for trial t occurs. Also only a single update is allowed per trial, so if multiple labels arrive at the start of the trial, only one can be used for the update. The remaining labels must wait for another trial to perform their update. Therefore, $\overline{OD3}$-$B(k')$ computes the same hypotheses as algorithm B but uses them in different trials.

There is an exception to the above scheme based on the single parameter k'. This parameter controls the maximum delay $\overline{OD3}$-$B(k')$ allows for any instance. If an instance has not received its label after $k'+1$ trials then the algorithm pretends the instance does not exist for the purpose of updates. For some problems, this technique is important since otherwise a single early instance with an infinite delay can prevent all later updates. The pseudo-code for $\overline{OD3}$-$B(k')$ is given in Fig. 8.5. We use a heap U to store the instances that are ready for updates. Also we have designed the $\overline{OD3}$-$B(k')$ algorithm so that the k' parameter can be increased while the algorithm is running. This allows use to build off the $\overline{OD3}$-$B(k')$ algorithm to create an algorithm that learns a good k' parameter.

The computational cost of the $\overline{OD3}$-$B(k')$ algorithm is similar to the cost of the B and $OD1$-B algorithms. The number of updates is at most the same as algorithm B and the number of predictions is at most double. There is an extra cost based on using the heap. The cost to insert and remove instances from the heap adds a cost of $O(\ln(\min(k', k)))$ per trial. The $\overline{OD3}$-$B(k')$ algorithm needs space for at most $\min(k', k)$ instances.

Algorithm $\overline{OD3}$-$B(k')$
Initialization
$t \leftarrow 0$ is the trial number.
`current` $\leftarrow 1$ is the next instance for an update.
`skipped` $\leftarrow 0$ is the number of instances skipped.
Initialize empty hash table H that stores new instances.
Initialize empty stack U that stores instances ready for updates.
Initialize algorithm to state $s \leftarrow s_0$.
Trials
$t \leftarrow t + 1$.
Instance: Store $\mathbf{x}_t$ in H with key t.
Prediction: $\hat{y}_t \leftarrow \text{Pred}(s, \mathbf{x}_t)$.
Update:
For all returned labels $y_{a,t}$
Remove x_a from H.
If $a \geq$ `current` then insert instance $(a, x_a, y_{a,t})$ to U.
$(a, x_a, y_{a,t}) \leftarrow \min(U)$.
If $a =$ `current`
$(a, x_a, y_{a,t}) \leftarrow$ extract-min(U)
$\hat{y} \leftarrow \text{Pred}(s, x_a)$
$s \leftarrow \text{Update}(s, x_a, y_{a,t}, \hat{y})$
`current` $\leftarrow$ `current` $+ 1$.
Else if $t -$ `current` $= k'$ then
Remove $x_{\texttt{current}}$ from H.
`current` $\leftarrow$ `current` $+ 1$.
`skipped` $\leftarrow$ `skipped` $+ 1$.

Figure 8.5: Pseudo-code for delayed on-line algorithm $\overline{OD3}$-$B(k')$.

Before we give the upper mistake bound for the $\overline{OD3}$-$B(k')$ algorithm, we prove a lemma that bounds the number of trials that do not perform an update. We need the following notation; let μ be the number of instances that are skipped by $\overline{OD3}$-$B(k')$ because their delay is too large. This is stored in the variable `skipped`.

Lemma 8.16 *When running the $\overline{OD3}$-$B(k')$ algorithm, there are at most $\mu + \min(k', k)$ trials that do not perform an update.*

Proof Assume that trial t is the $\min(k', k) + \mu + 1$ trial that does not perform an update. Therefore only $t - \min(k', k) - \mu - 1$ labels have been used for updates. Looking at instance x_1 to instance $x_{t-\min(k',k)}$, any of these instances that have not been skipped must have already returned their labels by trial t. Since at most μ of these instances have been skipped, the minimum number of labels from these instances that have been returned is $t - \min(k', k) - \mu$. This means there must be at least one label from this sequence of instances that has not been used for an update. By trial t, this

label has been placed on the heap for an update. This is a contradiction since trial t does not perform an update. ■

Theorem 8.17 *Assume the instance are generated by a subsequence distribution adversary Υ. Assume the expected number of mistakes of a traditional on-line algorithm B is at most M when instances are generated by a sequence of distributions from Υ. The expected number mistakes made by the $\overline{OD3}$-$B(k')$ algorithm is at most $M + (\mu + \min(k', k) - 1)(\Psi + 1)$ in the delayed distribution model.*

Proof Let s be a sequence of instances generated by $\mathcal{W} \in \Upsilon$ with delay set $\mathcal{D}$. Let u_s be the sequence of instances that cause updates in algorithm $\overline{OD3}$-$B(k')$. If we use sequence u_s on algorithm B with a delay of 1 given to each instance then we expect to make at most M mistakes. This is because Υ is a subsequence distribution adversary.

For algorithm B on sequence u_s, let h_i be the hypothesis that is used for prediction in trial i, and let X_i be a random variable that is 1 if algorithm B makes a mistake on trial i and 0 otherwise. Let Y_i be a random variable that is 1 if algorithm $\overline{OD3}$-$B(k')$ makes a mistake on trial i and 0 otherwise.

The hypotheses used by $\overline{OD3}$-$B(k')$ on s are the same as the hypotheses used by algorithm B on u_s. The only difference is that hypotheses of $\overline{OD3}$-$B(k')$ are shifted a certain positive number of trials since the instances have to wait for their labels. This will force the $\overline{OD3}$-$B(k')$ algorithm to occasionally use the same hypothesis for multiple trials as the algorithm waits for a label. Based on Lemma 8.16, the maximum number of trials that do not perform an update for algorithm $\overline{OD3}$-$B(k')$ is $\mu + \min(k', k)$. Since the first trial never performs an update, this means that $\mu + \min(k', k) - 1$ of the hypotheses from algorithm B are reused. Let $r = \mu + \min(k', k) - 1$.

Consider the shifted hypothesis of algorithm $\overline{OD3}$-$B(k')$. When the same hypothesis is used on two related distributions, the accuracies will be similar. The difference in accuracy comes from the amount the distribution changes between the trials. Let $v_t = V(D_{t+1}, D_t)$. Based on this metric, the error-rate of hypothesis h_t may, in the worst case, increase by v_t if used during trial $t + 1$. Since each hypothesis is shifted

Algorithm $\overline{OD3}$-B

Initialization

$t \leftarrow 0$ is the trial number.

`skipped` is the number of instances that have been skipped in $\overline{OD3}$-$B(k')$.

Trials

Run algorithm $\overline{OD3}$-$B(k')$ always setting $k' \leftarrow$ `skipped+1`

Figure 8.6: Pseudo-code for delayed on-line algorithm $\overline{OD3}$-B.

by at most r trials, the error-rate of hypothesis h_t can increase by at most $\sum_{i=t}^{t+r-1} v_i$. We can use this to bound the expected number of mistakes. Assuming mistakes on repeated hypotheses and taking into account the number of trials the hypothesis from B are shifted,

$$E[Mist(\overline{OD3}\text{-}B(k'))] \leq r + \sum_{i=1}^{\infty} E[Y_i] \ .$$

The above is

$$\leq r + \sum_{i=1}^{\infty} \left(E[X_i] + \sum_{j=i}^{i+r-1} v_j \right) \leq r + E\left[\sum_{i=1}^{\infty} X_i\right] + r\sum_{i=i}^{\infty} v_i \leq r + M + r\Psi \ .$$

This proves the result. ■

A major disadvantage of the $\overline{OD3}$-$B(k')$ algorithm is specifying the k' parameter. A simple modification that gives close to optimal performance is to set $k' = 1 +$ `skipped`. We call this algorithm $\overline{OD3}$-B. Algorithm $\overline{OD3}$-B has a similar cost as $\overline{OD3}$-$b(k')$. Let k_2 be the maximum value of k' used in $\overline{OD3}$-B. The computation cost of $\overline{OD3}$-B is at most the cost of $\overline{OD3}$-$B(k_2)$ since the cost per trial of $\overline{OD3}$-$B(k')$ monotonically increases with k'.

Next, we want to relate the mistake bound of $\overline{OD3}$-B with $\overline{OD3}$-$B(k')$ when k' is chosen to have the optimal fixed value. First, we need a better understanding of the optimal k' for algorithm $\overline{OD3}$-$B(k')$. Because the number of skipped instances is at most $\sum_{i>k'} d_i$, the bound from Theorem 8.17 is at most

$$M + \left(\left(\sum_{i>k'} d_i \right) + k' - 1 \right) (\Psi + 1)$$

To minimize this bound, we want to choose a value of k' that minimizes $k' + \sum_{i>k'} d_i$. Call one of these values k_1.

Corollary 8.18 *Assume the expected number of mistakes of traditional on-line algorithm B is at most M when instances are generated from Υ. The expected number mistakes made by the $\overline{OD3}$-B algorithm is at most $M + (2k_1 - 1 + 2\sum_{i>k_1} d_i)(\Psi + 1)$ in the delayed distribution model.*

Proof For any value $a \in N$, algorithm $\overline{OD3}$-B skips at most $a + \sum_{i>a} d_i$ instances because each time an instance is skipped, k' increases by 1. Therefore algorithm $\overline{OD3}$-B can skip at most $k_1 + \sum_{i>k_1} d_i$ instances. This gives a final k' value of at most $k_1 + \sum_{i>k_1} d_i$. Plugging these value into Theorem 8.17 proves the corollary. ■

The previous theorem shows that, in the worst-case, we do roughly a factor of two worse than the bound for an optimal setting of the k' parameter. This bound is tight as it is straightforward to generate the delays that cause this increase in the mistake bound.

A possible modification to algorithm $\overline{OD3}$-$B(k')$ or $\overline{OD3}$-B is to predict with a random coin flip on any repeated hypothesis. This will lower the upper-bound on mistakes for $\overline{OD3}$-$B(k')$ to $M + (\mu + \min(k', k) - 1)(\Psi + 1/2)$. In practice, one may want to restrict a coin flip prediction to repeated hypotheses near the start of the trials since later repeated hypotheses may have a high accuracy.

Also, just as with the $\overline{OD2}$-B algorithm, it could be beneficial to update with multiple instance on a single trial. Again, this has the advantage of using the label information as soon as it is available, but requires the B algorithm to have hypotheses where the expected accuracy is monotonically increasing in order to preserve the mistake bound.

8.3.3 Lower-bounds

In this section, we consider lower-bounds on the number of mistakes when instances are generated in the adversarial distribution model with delayed instances. These lower-bounds are based on the optimal randomized algorithm, Opt_R, in the traditional on-line setting. For the most part, we will consider a fixed distribution generating the instances.

For a fixed distribution, a trivial lower bound for the delayed learning problem is the bound of the optimal algorithm on the traditional on-line learning problem. We can refine this bound slightly by noticing that the adversary can select delays so that no label is received for the beginning on-line trials. At first glance, it seems the best the algorithm can do is to predict with a coin flip on these initial instances because no labels are available. There are two difficulties with this assumption.

First, the algorithm might be able to select an initial hypothesis that guarantees better than random performance during the initial trials. Whether this is possible depends on Υ, but for many sets of learning problems, it is not possible. Many sets of learning problems have a symmetry such that for every problem and starting hypothesis that performs better than chance, there is another problem of equal difficulty that forces the same starting hypothesis to perform worse than chance. For example, if the problem has a bias for a particular label, there is generally a matching problem that has a bias for another label.

The second difficulty is that the algorithm might be able to perform semi-supervised learning techniques to get information on the target function using the instances without the labels. For example, if instances from each label are known to come from different normal distributions then this information could help the learning algorithm without the need for labels [DH73].

To improve our lower-bound, we assume that the adversary has the symmetry needed to prevent the initial hypothesis from performing better than chance and that the algorithm does not update its state based on attributes without label information. This is true for all the algorithms considered in this dissertation. Given these assumptions, the best the algorithm can do is to predict over the labels uniformly until the

environment starts returning labels.

Therefore, our lower-bound depends on the number of instances that can be received before a label arrives. This is just determined by $F(\mathcal{D}, 1)$ since this is the maximum number of instances that arrived but have not yet returned their labels. For binary labels, this gives a lower-bound of $\text{Mist}(Opt_R) + (F(\mathcal{D}, 1) - 1)/2$. The $-1/2$ term comes from the fact that the first hypothesis needs to make a best guess for both the traditional setting and the delayed setting, and therefore it is already counted in the bound for Opt_R. This is identical to the bound for $\overline{OD1}\text{-}Opt_R$ when $\overline{OD1}\text{-}Opt_R$ uses coin flips to predict for repeated hypothesis.

For a shifting adversary, we do not have a strong lower-bound. We can use the bound from the fixed distribution to lower-bound the shifting adversary since the shifting adversary is more general. This lower-bound of $\text{Mist}(Opt_R) + (F(\mathcal{D}, 1) - 1)/2$ will be good when there is not much shifting in the distribution. Using algorithm $\overline{OD3}\text{-}Opt_R(k')$ gives a bound of $\text{Mist}(Opt_R) + (\mu + \min(k', k) - 1)(\Psi + 1)$. This compares well to the lower bound when $(\mu + \min(k', k) - 1)(\Psi + 1)$ is small. Of course, this bound is progressively worse as Ψ is allowed to grow. However, remember that a shifting distribution can duplicate an adversary by using distributions that place all the weight on particular instances. Therefore, the bounds for shifting distributions also covers our previous adversaries. However, when dealing with a problem that is more adversarial, this distribution based bound will be quite poor. The distribution bound is most relevant for problems where Ψ is small.

8.4 Summary

In this chapter, we give algorithms and mistake bounds for delayed on-line learning. In general, when dealing with an adversary generating the instances, the new bounds can be poor. If the instances all have a delay of k trials then the mistake bound can grow by a factor of $2k$ over the normal on-line learning bounds. We show this bound is within a factor of 2 from optimal in the no-noise case.

We also give a more general analysis for problems where the adversary must select

the delay of instances from a multi-set $\mathcal{D}$. This upper-bound depends on a particular combinatorial property of $\mathcal{D}$ and gives insight to how the algorithm behaves with a range of instance delays.

Things are more hopeful when dealing with a fixed distribution generating the instances. In this case, if all the instances have a delay k then the expected mistake bound only increases by $k-1$. Even with a slightly shifting distribution, we give a transformation that performs well with a small increase in mistakes based on the amount of shifting.

Chapter 9

Conclusion and Future Work

The goal of this dissertation is to give several techniques to improve the performance of on-line learning for specific types of practical problems. While further work is needed, we feel that this dissertation is an important step in making on-line learning a more practical tool for machine learning applications.

9.1 Contributions

Chapter 2 is a preliminary chapter that gives information about several linear-threshold algorithms. These are the main algorithms that we use and modify in this dissertation. For all of these algorithms, we give bounds on the number of mistakes when instances are generated by an adversary. While all of these algorithms have been previously published, the upper-bounds on mistakes are either new or refinements of previous bounds. In addition, we express all the bounds in a uniform notation. This simplifies comparison of the algorithms.

Chapter 3 gives a technique that modifies on-line algorithms to improve their performance on problems where instances are generated by sampling from a distribution. Our technique modifies existing on-line algorithm to generate a new algorithm that combines several hypotheses to make a more accurate prediction. While similar voting techniques have been studied [Lit95], we build off this previous work and show with real world data sets that our voting technique improves performance at a similar computational cost. In addition, our voting technique gives an efficient method for parameter and algorithm selection with on-line learning.

Chapter 4 gives another technique to improve the performance of adversarial on-line algorithms. We modify an existing algorithm to recycle over saved instances as

if they were new trials. However, each instance can only be used once in an update that changes the algorithm's hypothesis. Any mistakes on these saved instances can potentially decrease the number of mistakes on new trials. This technique was originally created by Littlestone [Lit96]. We modify his technique to allow only a recent window of saved instances in order to keep the computational cost low, and we allow the algorithm to use each instance u times, where u is set by the user of the algorithm. We show that instance recycling works on the same data sets used with voting, and Chapter 5 shows that combining instance recycling and voting improves performance more than either technique in isolation.

In Chapter 6, we give an upper-bound on the number of mistakes for a version of the Unnormalized Winnow algorithm that tracks linear-threshold functions. The bound shows that the concept tracking version has many of the same advantages as traditional fixed concept Winnow algorithm. We also show how the performance of this algorithm does not depend on the number of attributes and instead depends on $\max_{t\in\{1,\ldots,T\}} \ln\left(\|X_t\|_1\right)$. In Appendix A, we prove that the original Winnow algorithm also has this property if its parameters are set to an appropriate value. Chapter 7 continues the work on tracking by giving experiments with the tracking version of Winnow and a tracking version of ALMA [KSW02]. We also perform experiments to improve the tracking algorithms using the voting and instance recycling techniques of Chapter 3 and Chapter 4.

Our final chapter introduces a modification of the on-line model. The on-line model traditionally reports the label for each previous instances before the algorithm needs to make a prediction on the next instance. Here we extend the model and allow the label to be returned during any future trial. We give general techniques to transform traditional on-line algorithms to this setting and give upper and lower bounds on these transformed algorithms for both adversarial instance generation and distribution instance generation. Given an optimal algorithm for the traditional on-line setting, our transformations generate algorithms for the delayed setting that are close to optimal.

9.2 Future Work

We have several areas of research we wish to explore in the future. Part of this research is based on the results discovered in the course of completing this thesis. Other parts are based on related ideas that did not fit into the thesis based on time and space constraints.

9.2.1 Extensions of Thesis

Normalized Winnow often performed the best in our experiments with fixed and shifting distributions. However, for some experiments, particularly with sparse instances, Balanced Winnow gave the best performance. In the future, we would like to do experiments with Balanced Winnow using the threshold parameter of Normalized Winnow. The threshold parameter of Normalized Winnow has a large effect on its performance, and it is likely that Balanced Winnow would also benefit. This could create a new set of Balanced Winnow algorithms with even better performance on sparse instances.

Another interesting topic is to expand the set of algorithms with good mistake bounds for concept tracking. In particular, we would like to research variants of Normalized Winnow and Balanced Winnow that allow the algorithms to track concepts. Part of this research should focus on the somewhat surprising result that most of the fixed concept algorithms already perform well on several tracking problems. Do there exist distributions that cause these fixed concept algorithms to perform poorly on tracking concept problem or are adversaries necessary?

On the same topic, we would like to find a way to increase the efficiency of tracking Unnormalized Winnow algorithm. The complemented attributes combined with the minimum weight value make it difficult to efficiently implement the algorithm when updating with sparse instances. This also gives more incentive to modify Balanced Winnow to handle concept tracking. Because Balanced Winnow does not need complemented attributes, it can efficiently work with sparse instances even with the minimum weight modification used by Tracking Unnormalized Winnow.

Another area for future research is to consider more complicated techniques to improve the performance of adversarial on-line algorithms for shifting distributions. In Chapter 7, we used slight modifications to the techniques developed for fixed distributions. It should be possible to specialize these techniques for even greater improvement.

9.2.2 Promising Avenues for On-line Learning

Next we consider some areas of on-line research that we feel are important, but we have not had a chance to significantly explore in this dissertation. One omission is multi-class learning. Here the target function can have more than just two output labels. We have done some preliminary work in this area [Mes00, Mes01]. Also our work in Chapter 8 on delayed learning covers multi-class problems. However, a more extensive evaluation of the wide range of research on multi-class learning is needed to understand what works best for on-line learning.

Another interesting area of on-line learning research is the creation of new features. The main advantage of on-line learning is a potentially infinite stream of label feedback. On-line learning algorithms need to find ways to exploit this information to improve performance. One possibility is the constant exploration of new features that may give a better representation of the target function. This is partially explored in [FG03] using boosting like techniques on the problem of CPU branch prediction. While this is a promising initial step, we feel other techniques should be considered along with a more comprehensive set of experiments to help evaluate performance.

On problem with applying the techniques of [FG03] is that the Arc-x4 boosting algorithm [Bre98] requires some instances to be given more influence on learning. This is commonly called cost-sensitive learning [ZLA03]. Previous work on cost-sensitive, on-line learning includes [HLL00b] and [HLL00a]. Unfortunately, this work is highly theoretical, and the algorithms have a high computational cost. Part of the problem is that the techniques attempt to give good bounds even when dealing with an adversary. Cost sensitive learning is an important problem in its own right, and it would be interesting to see if efficient on-line algorithms exist that give good performance guarantees. Another alternative is to optimize cost-sensitive learning for on-line problems where

instances are generated by something weaker than an adversary such as a distribution.

The very same advantage of on-line learning is also one of its greatest weaknesses. The constant need for label feedback restricts the applicability of the model. Many learning problems can not provide label feedback for most of the instances because of the cost involved in getting accurate labels. An important solution to this problem is to consider co-training techniques for on-line learning [BM98]. These techniques use multiple ways to solve the problem as a technique to generate labels. While the labels might initially have low accuracy, the technique bootstraps itself to improve performance. We feel that some of our work on improving the accuracy of on-line algorithm hypotheses when dealing with distributions might be important tools for creating solutions for co-training problems using efficient on-line algorithms.

Appendix A

Unnormalized Winnow

In this appendix, we prove mistake bounds for Unnormalized Winnow. First, we give a slightly refined analysis of the algorithm which lowers previously published mistake-bounds by a constant. The analysis also shows how noisy instances affect the mistake-bound. The noise analysis is similar to that presented in [Lit89], but uses the noise notation of Chapter 2. Second, we use an alternative initial value of the weights to improve the mistake bound when the maximum one-norm of the instances is small. Last, we show how noisy instances effect the algorithm as the multiplier α approaches 1. To clarify the results, we restate the algorithm in Figure A.1. This is identical to the algorithm found in Section 2.3.1. For the following proofs define $0 \ln 0 = 0$.

Our first theorem is a slight modification of the result found in [Lit91]. We improve some of the approximations used, which decreases the upper-bound by a constant factor, and we change the notation to a form more compatible with the other results in this dissertation.

First, we need to specify the noise function that controls the generation of the instances. Let $\mathbf{u} \in [0, \infty)^n$ be the target weights. Let $\delta > 0$ be the margin. The noise on instance $\mathbf{x_t}$ is defined as $\nu_t = \max(0, \delta - y_t(\mathbf{u} \cdot \mathbf{x_t} - 1))$ and the total noise, up to trial T, is $N = \sum_{t=1}^{T} \nu_t$. For more information on the noise function see Section 2.2.

Before we begin the proof, we give a helpful lemma. This lemma gives upper and lower bounds for $\ln(1 + x)$ using rational functions.

Lemma A.1 *For all $x \geq 0$,*

$$\frac{2x}{2+x} \leq \ln(1+x) \leq \frac{2x(1+x^2/10)}{2+x}.$$

Proof Let $f(x) = \frac{2x(1+x^2/10)}{2+x} - \ln(1+x)$. It is sufficient to show that $f(0) = 0$ and

Unnormalized Winnow(α, σ)
Parameters
$\alpha > 1$ is the update multiplier.
$\sigma > 0$ is the initial weight value.
Initialization
$t \leftarrow 1$ is the current trial.
$\forall i \in \{1, \ldots, n\}$ $w_{i,1} = \sigma$ are the weights.
Trials
Instance: $\mathbf{x}_t \in [0,1]^n$.
Prediction: If $\mathbf{w_t} \cdot \mathbf{x_t} \geq 1$
Predict $\hat{y}_t = 1$ else predict $\hat{y}_t = -1$.
Update: Let $y_t \in \{-1, 1\}$ be the correct label.
If $y_t(\mathbf{w_t} \cdot \mathbf{x_t}) \leq 0$ then
$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = \alpha^{y_t x_{i,t}} w_{i,t}$.
Else
$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = w_{i,t}$.
$t \leftarrow t + 1$.

Figure A.1: Pseudo-code for the Unnormalized Winnow algorithm.

$f'(x) \geq 0$ for all $x \geq 0$. Clearly $f(0) = 0$. The derivative $f'(x) = \frac{(2x^2+8x+1)x^2}{5(1+x)(2+x)^2} \geq 0$ for all $x > 0$.

The same technique works for the lower-bound. Let $f(x) = \ln(1+x) - \frac{2x}{2+x}$. Again $f(0) = 0$ and $f'(x) = \frac{x^2}{(1+x)(2+x)^2} \geq 0$ for all $x > 0$. ■

The derivation of a mistake-bound is based on a progress function. This progress function increases on every mistake by at least a fixed amount and has a maximum value. The only exception occurs for noisy instances, which can cause a decrease in the progress function. However, since we assume a upper-bound on the noise, the net effect is to increase the total amount of progress we must make until we we reach the maximum value of the progress function.

Let the progress function be $Q(t) = \sum_{i=1}^{n} u_i \ln w_{i,t} - \sum_{i=1}^{n} w_{i,t}$ where $\mathbf{u}$ are the weights from the target function, and $\mathbf{w}_t$ are the weights from the Unnormalized Winnow algorithm at the start of trial t [Lit91]. The progress function can only change after a mistake because that is the only time Unnormalized Winnow changes its weights.

Lemma A.2 *After a mistake on trial t, if $\alpha = 1 + \delta$ then*

$$Q(t+1) - Q(t) > \frac{\delta^2}{2+\delta} - \frac{\delta(2+\delta^2/5)}{2+\delta}\nu_t.$$

Proof After a mistake,

$$\begin{aligned} Q(t+1) - Q(t) &= \sum_{i=1}^{n} u_i \ln\left(\alpha^{y_t x_{i,t}} w_{i,t}\right) - \sum_{i=1}^{n} \alpha^{y_t x_{i,t}} w_{i,t} - \left(\sum_{i=1}^{n} u_i \ln w_{i,t} - \sum_{i=1}^{n} w_{i,t}\right) \\ &= \sum_{i=1}^{n} w_{i,t} + y_t \ln(\alpha) \sum_{i=1}^{n} u_i x_{i,t} - \sum_{i=1}^{n} \alpha^{y_t x_{i,t}} w_{i,t}. \end{aligned}$$

Based on the convexity of α^x, $\alpha^{y_t x_{i,t}} \leq 1 + (\alpha^{y_t} - 1)x_{i,t}$ for all $x_{i,t} \in [0,1]$ and $y_t \in \{-1, 1\}$. Using this fact, the preceding is greater than or equal to

$$\begin{aligned} &\sum_{i=1}^{n} w_{i,t} + y_t \ln(\alpha) \sum_{i=1}^{n} u_i x_{i,t} - \sum_{i=1}^{n} w_{i,t} - (\alpha^{y_t} - 1) \sum_{i=1}^{n} w_{i,t} x_{i,t} \\ &= y_t \ln(\alpha) \sum_{i=1}^{n} u_i x_{i,t} - (\alpha^{y_t} - 1) \sum_{i=1}^{n} w_{i,t} x_{i,t}. \end{aligned}$$

We break the rest of the proof into two cases. For the first case, assume $y_t = -1$. Therefore

$$Q(t+1) - Q(t) \geq -\ln(\alpha) \sum_{i=1}^{n} u_i x_{i,t} - (1/\alpha - 1) \sum_{i=1}^{n} w_{i,t} x_{i,t}.$$

Because $\hat{y}_t = 1$, $\sum_{i=1}^{n} w_{i,t} x_i \geq 1$, and $\sum_{i=1}^{n} u_i x_{i,t} \leq 1 - \delta + \nu_t$, the last formula is greater than or equal to

$$-\ln(\alpha)(1 - \delta + \nu_t) - (1/\alpha - 1) = 1 - 1/\alpha - \ln(\alpha)(1 - \delta) - \ln(\alpha)\nu_t \tag{A.1}$$

Using our assumption that $\alpha = 1 + \delta$ and the upper-bound from Lemma A.1,

$$Q(t+1) - Q(t) \geq \frac{\delta^2}{2+\delta}\frac{5 + 9\delta + \delta^3}{5 + 5\delta} - \frac{2\delta(1 + \delta^2/10)}{2+\delta}\nu_t > \frac{\delta^2}{2+\delta} - \frac{\delta(2+\delta^2/5)}{2+\delta}\nu_t$$

The second case is similar. Assume $y_t = 1$. Therefore

$$Q(t+1) - Q(t) \geq \ln(\alpha) \sum_{i=1}^{n} u_i x_{i,t} - (\alpha - 1) \sum_{i=1}^{n} w_{i,t} x_{i,t}.$$

Because $\hat{y}_t = -1$, $\sum_{i=1}^{n} w_{i,t} x_i \leq 1$, and $\sum_{i=1}^{n} u_i x_{i,t} \geq 1 + \delta - \nu_t$, the last formula is greater than or equal to

$$\ln(\alpha)(1 + \delta - \nu_t) - (\alpha - 1) = \ln(\alpha)(1 + \delta) - \alpha + 1 - \ln(\alpha)\nu_t \tag{A.2}$$

Using our assumption that $\alpha = 1 + \delta$ and the bounds from Lemma A.1, we get

$$Q(t+1) - Q(t) \geq \frac{\delta^2}{2+\delta} - \frac{\delta(2+\delta^2/5)}{2+\delta}\nu_t.$$

■

The main theorem follows by computing the starting value and maximum value of the progress function, Q.

Theorem A.3 *The number of mistakes made by Unnormalized Winnow when $\alpha = 1+\delta$ is at most*

$$\frac{(2+\delta)\left(\sigma n + \sum_{i=1}^n u_i \ln u_i + \ln(1/\sigma)\sum_{i=1}^n u_i - \sum_{i=1}^n u_i\right)}{\delta^2} + \frac{(2+\delta^2/5)N}{\delta}.$$

Proof The maximum value of $Q(t)$ can be determined by taking its derivative with respect to the algorithm weights and setting these equations to zero. The maximum is achieved when $\mathbf{w}_t = \mathbf{u}$. Therefore,

$$\sum_{i=1}^n u_i \ln u_i - \sum_{i=1}^n u_i - \left(\sum_{i=1}^n u_i \ln \sigma - \sum_{i=1}^n \sigma\right) \geq Q(T+1) - Q(1)$$

$$= \sum_{t=1}^T Q(t+1) - Q(t) = \sum_{t \in M} Q(t+1) - Q(t) \geq \sum_{t \in M}\left(\frac{\delta^2}{2+\delta} - \frac{\delta(2+\delta^2/5)}{2+\delta}\nu_t\right)$$

$$= \frac{\delta^2}{2+\delta}|M| - \frac{\delta(2+\delta^2/5)}{2+\delta}\sum_{t \in M}\nu_t \geq \frac{\delta^2}{2+\delta}|M| - \frac{\delta(2+\delta^2/5)}{2+\delta}N.$$

Rearranging the inequality gives,

$$|M| \leq \frac{(2+\delta)\left(\sigma n + \sum_{i=1}^n u_i \ln u_i + \ln(1/\sigma)\sum_{i=1}^n u_i - \sum_{i=1}^n u_i\right)}{\delta^2} + \frac{(2+\delta^2/5)N}{\delta}.$$

Notice that this proof works for any value of T, therefore it is an upper-bound on the number of mistakes. ■

Given Theorem A.3, the optimal setting of the initial weights is $\sigma = \sum_{i=1}^n u_i/n$. Unfortunately, for most problems the user of the algorithm does not know the value of $\sum_{i=1}^n u_i$. However, as can be seen in the previous bound, $\sigma = 1/n$ is close to optimal as long as $\sum_{i=1}^n u_i << n$. Typically this is true when only a few of the n attributes are relevant.

Corollary A.4 *The number of mistakes made by Unnormalized Winnow when* $\alpha = 1+\delta$ *and* $\sigma = 1/n$ *is at most*

$$\frac{(2+\delta)\left(1+\sum_{i=1}^{n} u_i \ln u_i + \ln(n)\sum_{i=1}^{n} u_i - \sum_{i=1}^{n} u_i\right)}{\delta^2} + \frac{(2+\delta^2/5)N}{\delta}.$$

Proof Just use Theorem A.3 with $\sigma = 1/n$. ■

The previous result can be improved if we impose further restrictions on the adversary. For example, assume the adversary can only generate instances with a small one-norm. Let λ be an upper-bound on the one norm of the instances.

Theorem A.5 *Let* $\alpha = 1 + 0.98\delta$ *and* $\sigma = \frac{\delta}{50\lambda}$. *Let* k *be the number of target weights that have* $u_i > \sigma$. *Without loss of generality, assume the first* k *attributes have* $u_i > \sigma$. *The number of mistakes made by Unnormalized Winnow is less than*

$$\frac{(2.09+1.03\delta)\left(\sum_{i=1}^{k} u_i \ln u_i + \ln\left(\frac{50\lambda}{\delta}\right)\sum_{i=1}^{k} u_i\right)}{\delta^2} + \frac{(2.05+\delta^2/5)N}{\delta}.$$

Proof For every target weight value u_i that is less than σ, shift its value to σ. Every other target weight keeps the same value. Let $\hat{u}_i$ be the new target weights and let k be the number of new target weights that do not equal σ.

Since we have increased the target weight values, the value of $\mathbf{u}\cdot\mathbf{x_t}$ may increase. We have increased each weight by at most σ, and λ is an upper-bound for the one-norm of an instance, therefore

$$\sum_{i=1}^{n} u_i x_{i,t} \le \sum_{i=1}^{n} \hat{u}_i x_{i,t} \le \sum_{i=1}^{n} u_i x_{i,t} + \sigma\lambda \ .$$

Let $\hat{\delta} = \delta - \sigma\lambda = \delta(1-1/50)$ and $\hat{\nu_t} = \max(0, \hat{\delta} - y_t(\hat{\mathbf{u}}\cdot\mathbf{x_t} - 1))$. When $y = -1$,

$$\hat{\nu_t} = \max(0, \delta - \sigma\lambda + (\hat{\mathbf{u}}\cdot\mathbf{x_t} - 1)) \le \max(0, \delta - \sigma\lambda + (\mathbf{u}\cdot\mathbf{x_t} + \sigma\lambda - 1)) = \nu_t \ .$$

When $y = 1$,

$$\hat{\nu_t} = \max(0, \delta - \sigma\lambda - (\hat{\mathbf{u}}\cdot\mathbf{x_t} - 1)) \le \max(0, \delta - \sigma\lambda - (\mathbf{u}\cdot\mathbf{x_t} - 1)) \le \nu_t \ .$$

Therefore $\hat{N} = \sum_{t=1}^{T} \hat{\nu_t} \le \sum_{t=1}^{T} \nu_t = N$.

Without loss of generality, assume the first k attributes have $\hat{u}_i > \sigma$. Substituting the new target function into the bound in Theorem A.3 gives

$$\frac{(2+\hat{\delta})\left(\sigma n + \sum_{i=1}^{n} \hat{u}_i \ln \hat{u}_i + \ln(1/\sigma) \sum_{i=1}^{n} \hat{u}_i - \sum_{i=1}^{n} \hat{u}_i\right)}{\hat{\delta}^2} + \frac{(2+\hat{\delta}^2/5)\hat{N}}{\hat{\delta}}.$$

$$= \frac{(2+\hat{\delta})\left(\sigma n + \sum_{i=1}^{k} u_i \ln u_i + \ln(1/\sigma) \sum_{i=1}^{k} u_i - \sum_{i=1}^{k} u_i - \sigma(n-k)\right)}{\hat{\delta}^2} + \frac{(2+\hat{\delta}^2/5)\hat{N}}{\hat{\delta}}.$$

$$< \frac{(2.09+1.03\delta)\left(\sum_{i=1}^{k} u_i \ln u_i + \ln\left(\frac{50\lambda}{\delta}\right) \sum_{i=1}^{k} u_i\right)}{\delta^2} + \frac{(2.05+\delta^2/5)N}{\delta}.$$

■

The upper-bound in Theorem A.5 is similar to the bound in Corollary A.4. The main difference is the $\ln(50\lambda/\delta)$ instead of $\ln n$ term. As $50\lambda/\delta$ gets smaller than n, the Theorem A.5 bound becomes advantageous. The advantage of using a large σ value with sparse instances was first mentioned in [GR96], however, the result was justified empirically without proof.

Inside the proof of Lemma A.2, we identified two equations. Equation A.1 and Equation A.2 show how the progress changes as a function of α. We can use these equations to show that, as we decrease α, the noise term tends to N/δ. As explained in Section 2.2, this is optimal. Let D be the set of trials where a mistake occurs and $y_t = -1$. Let P be the set of trials where a mistake occurs and $y_t = 1$.

Theorem A.6 *As α approaches 1, the upper-bound on the number of mistakes made by Unnormalized Winnow due to the noise term approaches N/δ.*

Proof Assume the change in progress on a mistake is upper-bounded by $a + b\nu$ when the label is -1 and $c + d\nu$ when the label is 1. Following the proof of Theorem A.3, the number of mistakes at trial T is at most

$$\frac{Q(T) - Q(1)}{\min(a,c)} + \sum_{t \in D} \frac{b}{a}\nu_t + \sum_{t \in P} \frac{d}{c}\nu_t.$$

Substituting in the values of a, b, c, and d from Equation A.1 and Equation A.2 gives a number of mistakes less than

$$\frac{Q(T) - Q(1)}{\min(a,c)} + \sum_{t \in D} \frac{\ln \alpha}{\frac{\alpha-1}{\alpha} - (1-\delta)\ln \alpha}\nu_t + \sum_{t \in P} \frac{\ln \alpha}{(1+\delta)\ln \alpha - \alpha + 1}\nu_t.$$

$$= \frac{Q(T) - Q(1)}{\min(a, c)} + \sum_{t \in D} \frac{1}{\frac{\alpha-1}{\alpha \ln \alpha} - (1 - \delta)} \nu_t + \sum_{t \in P} \frac{1}{(1 + \delta) - \frac{\alpha+1}{\ln \alpha}} \nu_t.$$

Because both $\frac{\alpha-1}{\alpha \ln \alpha}$ and $\frac{\alpha+1}{\ln \alpha}$ tend to 1 as α approaches 1, the sum of the noise terms tends towards $\sum_{t \in M} \nu_t / \delta$. ■

For problems, where the noise is bounded, these ideas can be used to set the multiplier to minimize the trade-off between the mistakes made by learning the concept and mistakes made by noisy instances [CBFH$^+$97].

In this appendix, our main result is a proof on the maximum number of mistakes made by Unnormalized Winnow. This bound gives a constant factor improvement on previous bounds found in [Lit91]. We also show how to set the parameters of Unnormalized Winnow to improve performance when the maximum one-norm of the instances is small. Last, we show how noisy instances effect Unnormalized as the multiplier α approaches 1.

Appendix B

Normalized Winnow

In this appendix, we prove an upper-bound on the number of mistakes made by the Normalized Winnow algorithm. This version of the algorithm allows a threshold parameter that significantly improves the performance for some learning problems. We also show that the algorithm exhibits optimal behavior for noisy instances as the weight multiplier α approaches 1. To clarify these results, we restate the algorithm in Figure B.1. This is identical to the algorithm found in Chapter 2. For the following proofs define $0 \ln 0 = 0$.

The addition of a threshold parameter to Normalized Winnow was made by Nick Littlestone [Lit94], but his original proof did not clarify the improvements of this algorithm over using Normalized Winnow with $\theta = 1/2$. The proof in this thesis is based on unpublished work done with Nick Littlestone to refine that proof. Normalized Winnow with a threshold has since appeared in other publications. In [GW99], Normalized Winnow is presented, and mistake bounds are given for learning disjunctions. Here we give more general mistake bounds for linear-threshold functions.

First, we need some information about the target function. Let $\mathbf{u}$ be a vector of n target weights where each $u_i \geq 0$ and $\sum_{i=1}^{n} u_i = 1$. Let $\tau = \min(\theta, 1-\theta)$ and let δ be the margin for the target function where $0 < \delta \leq \tau$. The noise for instance $\mathbf{x}_t$ is defined as $\nu_t = \max(0, \delta - y_t(\mathbf{u} \cdot \mathbf{x_t} - \theta))$, and the total noise is $N = \sum_{t=1}^{T} \nu_t$. For more information on target functions see Section 2.2.

The derivation of a mistake-bound is based on a progress function. This progress function increases on every mistake by at least a fixed amount and has a maximum value. The only exception is for noisy instances, which can cause a decrease in the progress function. However, since we assume an upper-bound on the noise, the net

Normalized Winnow(α, θ)
Parameters
$\alpha > 1$ is the update multiplier.
$0 < \theta < 1$ is the threshold.
Initialization
$t \leftarrow 1$ is the current trial.
$\forall i \in \{1, \ldots, n\}$ $w_{i,1} = 1$ are the weights.
Trials
Instance: $\mathbf{x}_t \in [0,1]^n$.
Prediction: If $\mathbf{w_t} \cdot \mathbf{x_t} \geq \theta \sum_{i=1}^{n} w_{i,t}$
Predict $\hat{y}_t = 1$ else predict $\hat{y}_t = -1$.
Update: Let $y_t \in \{-1, 1\}$ be the correct label.
If $y_t(\mathbf{w_t} \cdot \mathbf{x_t}) \leq 0$ then
$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = \alpha^{y_t x_{i,t}} w_{i,t}$.
Else
$\forall i \in \{1, \ldots, n\}$ $w_{i,t+1} = w_{i,t}$.
$t \leftarrow t + 1$.

Figure B.1: Pseudo-code for the Normalized Winnow algorithm.

effect is to increase the total amount of progress we must make until we we reach the maximum value of the progress function.

Let the progress function be $Q(t) = \sum_{i=1}^{n} u_i \ln w_{i,t} - \ln\left(\sum_{i=1}^{n} w_{i,t}\right)$ where $\mathbf{u}$ are the weights from the target function, and $\mathbf{w}_t$ are the weights from the Normalized Winnow algorithm at the start of trial t [Lit89]. The progress function can only change after a mistake because that is the only time Normalized Winnow changes its weights.

Lemma B.1 *If there is a mistake on trial t and the correct label is $y = -1$ then*

$$Q(t+1) - Q(t) \geq \ln(\alpha)(1 - \theta + \delta) - \ln(1 + (\alpha - 1)(1 - \theta)) - \ln(\alpha)\nu_t$$

If there is a mistake on trial t and the correct label is $y = 1$ then

$$Q(t+1) - Q(t) \geq \ln(\alpha)(\theta + \delta) - \ln(1 + (\alpha - 1)(\theta)) - \ln(\alpha)\nu_t$$

Proof Using the definition of $Q(t) = \sum_{i=1}^{n} u_i \ln w_i - \ln\left(\sum_{i=1}^{n} w_i\right)$, we derive that on a mistake

$$Q(t+1) - Q(t) = \sum_{i=1}^{n} u_i \ln w_{i,t+1} - \sum_{i=1}^{n} u_i \ln w_{i,t} - \ln\left(\sum_{i=1}^{n} w_{i,t+1}\right) + \ln\left(\sum_{i=1}^{n} w_{i,t}\right)$$

$$= \sum_{i=1}^{n} u_i \ln\left(\frac{w_{i,t+1}}{w_{i,t}}\right) - \ln\left(\frac{\sum_{i=1}^{n} w_{i,t+1}}{\sum_{i=1}^{n} w_{i,t}}\right) = \sum_{i=1}^{n} u_i \ln\left(\alpha^{y_t x_{i,t}}\right) - \ln\left(\frac{\sum_{i=1}^{n} \alpha^{y_t x_{i,t}} w_{i,t}}{\sum_{i=1}^{n} w_{i,t}}\right)$$

Based on the convexity of α^x, $\alpha^{y_t x_{i,t}} \leq 1 + (\alpha^{y_t} - 1)x_{i,t}$ for $x_{i,t} \in [0,1]$ and $y_t \in \{-1, 1\}$, therefore

$$Q(t+1) - Q(t) \geq y_t \ln(\alpha) \sum_{i=1}^{n} u_i x_{i,t} - \ln\left(1 + (\alpha^{y_t} - 1)\frac{\sum_{i=1}^{n} w_{i,t} x_{i,t}}{\sum_{i=1}^{n} w_{i,t}}\right)$$

We break the rest of the proof into two cases. First assume that $y_t = -1$. Therefore, based on the definition of noise, $\sum_{i=1}^{n} u_i x_{i,t} \leq \theta - \delta + \nu_t$. Also since the algorithm made a mistake, $\sum_{i=1}^{n} w_i x_{i,t} / \sum_{i=1}^{n} w_{i,t} \geq \theta$. This gives us

$$\begin{aligned}
Q(t+1) - Q(t) &\geq -\ln(\alpha)(\theta - \delta + \nu_t) - \ln\left(1 + (\alpha^{-1} - 1)\theta\right) \\
&= -\ln(\alpha)(\theta - \delta + \nu_t) - \ln\left(\frac{\alpha + \theta - \alpha\theta}{\alpha}\right) \\
&= \ln(\alpha)(\delta - \theta) - \ln(\alpha + \theta - \alpha\theta) + \ln(\alpha) - \ln(\alpha)\nu_t \\
&= \ln(\alpha)(1 - \theta + \delta) - \ln(1 + (\alpha - 1)(1 - \theta)) - \ln(\alpha)\nu_t.
\end{aligned}$$

Assume $y = 1$. Therefore, $\sum_{i=1}^{n} u_i x_{i,t} \geq \theta + \delta - \nu_t$ and $\sum_{i=1}^{n} w_i x_{i,t} / \sum_{i=1}^{n} w_{i,t} < \theta$. This gives us

$$\begin{aligned}
Q(t+1) - Q(t) &\geq \ln(\alpha)(\theta + \delta - \nu_t) - \ln(1 + (\alpha - 1)\theta) \\
&= \ln(\alpha)(\theta + \delta) - \ln(1 + (\alpha - 1)\theta) - \ln(\alpha)\nu_t.
\end{aligned}$$

This completes the proof. ■

The purpose of our next lemma is to help understand the lower-bounds that we just derived in Lemma B.1. Let $P_{-1}(\theta) = \ln(\alpha)(1 - \theta + \delta) - \ln(1 + (\alpha - 1)(1 - \theta)) - \ln(\alpha)\nu_t$ and let $P_1(\theta) = \ln(\alpha)(\theta + \delta) - \ln(1 + (\alpha - 1)(\theta)) - \ln(\alpha)\nu_t$. Our goal is to show that $P_1(\theta)$ is a lower-bound on the change in progress when $\theta \in [0, 1/2]$.

Lemma B.2 *If $\theta \in [0, 1/2]$ then $P_1(\theta) \leq P_{-1}(\theta)$.*

Proof Let

$$G(\theta) = P_{-1}(\theta) - P_1(\theta) = \ln(\alpha)(1 - 2\theta) + \ln(1 + (\alpha - 1)\theta) - \ln(1 + (\alpha - 1)(1 - \theta)).$$

Notice that $G(0)$ and $G(1/2)$ are both zero. We prove that $G(\theta)$ is non-negative for $\theta \in [0, 1/2]$ by showing that $G'(0) > 0$ and that $G(\theta)$ has one critical point in $[0, 1/2]$.

$$G'(\theta) = \frac{\alpha - 1}{1 + (\alpha - 1)\theta} + \frac{\alpha - 1}{1 + (\alpha - 1)(1 - \theta)} - 2\ln(\alpha).$$

Let

$$f(\alpha) = G'(0) = \alpha - \frac{1}{\alpha} - 2\ln\alpha.$$

Using Taylor's Theorem around $\alpha = 1$ gives $f(\alpha) = 0 + 0 + \frac{z-1}{z^3}(\alpha - 1)^2$ for $z \in (1, \alpha)$. Therefore, because $\alpha > 1$, $G'(0) > 0$.

Notice that $G'(\theta) = 0$ is quadratic in θ and that

$$\frac{1}{2} + \frac{\sqrt{\ln(\alpha)\left(\alpha^2 \ln\alpha + 2\alpha\ln\alpha + \ln\alpha - 2\alpha^2 + 2\right)}}{2(\alpha - 1)\ln\alpha} > \frac{1}{2}$$

is one of the critical points. Therefore there is only one critical point in $[0, 1/2]$. ■

This next lemma gives us a single lower-bound on the change in progress. It uses the fact that $P_{-1}(\theta)$ and $P_1(\theta)$ are mirror images. It also uses the previous definition that $\tau = \min(\theta, 1 - \theta)$.

Lemma B.3 *If there is a mistake on trial t then*

$$Q(t+1) - Q(t) \geq \ln(\alpha)(\tau + \delta) - \ln(1 + (\alpha - 1)\tau) - \nu_t \ln(\alpha).$$

Proof Recall the definitions for $P_{-1}(\theta)$ and $P_1(\theta)$. These functions mirror each other around $\theta = 1/2$. In other words, $P_{-1}(\theta) = P_1(1 - \theta)$ for $\theta \in [0, 1]$.

Based on Lemma B.1 and Lemma B.2, for $\theta \in [0, 1/2]$, $Q(t+1) - Q(t) \geq P_1(\theta)$. When $\theta \in [1/2, 1]$, we can combine the mirror nature of the functions with Lemma B.2 to show that $P_{-1}(\theta) \leq P_1(\theta)$. Therefore when $\theta \in [1/2, 1]$, $Q(t+1) - Q(t) \geq P_{-1}(\theta) = P_1(1-\theta)$. This proves the lemma. ■

Next we use the previous lemma to give the mistake-bound for Normalized Winnow. Recall that we define M as the set of trials where a mistake occurs.

Theorem B.4 *If* $\ln(\alpha)(\tau+\delta) - \ln(1+(\alpha-1)\tau) > 0$ *then the number of mistakes made by Normalized Winnow is at most*

$$\frac{\ln n + \sum_{i=1}^{n} u_i \ln u_i}{\ln(\alpha)(\tau+\delta) - \ln(1+(\alpha-1)\tau)} + \frac{\ln(\alpha) N}{\ln(\alpha)(\tau+\delta) - \ln(1+(\alpha-1)\tau)}.$$

Proof The maximum value of $Q(t)$ can be determined by taking it's derivative with respect to the algorithm weights and setting these equations to zero. The maximum is achieved when $w_{i,t}/(\sum_{i=1}^{n} w_{i,t}) = u_i$. This gives a maximum $Q(t)$ value of $\sum_{i=1}^{n} u_i \ln u_i$. Therefore, for any trial T,

$$\sum_{i=1}^{n} u_i \ln u_i + \ln n \geq Q(T+1) - Q(1) = \sum_{t=1}^{T} Q(t+1) - Q(t) = \sum_{t \in M} Q(t+1) - Q(t).$$

Based on Lemma B.3,

$$\begin{aligned}
\sum_{t \in M} Q(t+1) - Q(t) &\geq \sum_{t \in M} (\ln(\alpha)(\tau+\delta) - \ln(1+(\alpha-1)\tau) - \ln(\alpha)\nu_t) \\
&= (\ln(\alpha)(\tau+\delta) - \ln(1+(\alpha-1)\tau))|M| - \ln(\alpha) \sum_{t \in M} \nu_t \\
&\geq (\ln(\alpha)(\tau+\delta) - \ln(1+(\alpha-1)\tau))|M| - \ln(\alpha) N.
\end{aligned}$$

Therefore,

$$\sum_{i=1}^{n} u_i \ln u_i + \ln n \geq (\ln(\alpha)(\tau+\delta) - \ln(1+(\alpha-1)\tau))|M| - \ln(\alpha) N.$$

Rearranging this inequality proves the theorem. ■

At this stage, the upper-bound is not very intuitive because we do not know how the denominator, $\ln(\alpha)(\tau+\delta) - \ln(1+(\alpha-1)\tau)$, behaves. However, in this form, we can say some interesting things about how α effects the bounds. In particular, we can show how the noise term behaves as α gets close to 1.

Corollary B.5 *As* α *approaches* 1*, the number of mistakes made by Normalized Winnow tends to at most*

$$\frac{\ln n + \sum_{i=1}^{n} u_i \ln u_i}{\ln(\alpha)(\tau+\delta) - \ln(1+(\alpha-1)\tau)} + \frac{N}{\delta}.$$

Proof The only part of the bound from Theorem B.4, we are concerned with is the noise term,

$$\frac{\ln(\alpha)\,N}{\ln(\alpha)\,(\tau+\delta)-\ln(1+(\alpha-1)\tau)}=\frac{N}{\tau+\delta-\frac{\ln(1+(\alpha-1)\tau)}{\ln\alpha}}$$

Using L'Hôpital's rule,

$$\lim_{\alpha\to 1^+}\frac{\ln(1+(\alpha-1)\tau)}{\ln\alpha}=\tau.$$

Plugging this into the noise term from above gives

$$\lim_{\alpha\to 1^+}\frac{\ln(\alpha)\,N}{\ln(\alpha)\,(\tau+\delta)-\ln(1+(\alpha-1)\tau)}=\frac{N}{\tau+\delta-\tau}.$$

■

In the previous corollary, we were only concerned with the noise term. As α goes to 1, the first part of the bound, that deals with learning the target function, goes to infinity. A trade-off is needed. The value of α that minimizes the target function portion of the bound is $1+\frac{\delta}{\tau(1-\tau-\delta)}$. However, this goes to infinity as τ and δ approach $1/2$ and causes the constant on the noise term to approach infinity. Therefore, we must restrict the α value in order to control the effect of the noise.

For purposes of the proof, we set $\alpha=\min\left(6,1+\frac{\delta}{\tau(1-\tau-\delta)}\right)$. This cutoff on the value of the multiplier is sufficient to control the effects of noise on the mistake bound. This is not to say that one should never use a multiplier greater than 6. Even against an adversary, a larger multiplier may make fewer mistakes if the noise on the problem is small. However, a maximum value of 6 is convenient for our proof technique.

The remainder of the appendix is focused on restating Theorem B.4 in a more intuitive form. In its present form, the mistake-bound is difficult to interpret. Unfortunately, this requires some effort given the number of variables involved and the cutoff used in setting α. We start with two lemmas that help us determine when $\alpha=6$.

Lemma B.6 *If* $1+\frac{\delta}{\tau(1-\tau\delta)}\geq 6$ *then* $\tau\geq 2/5$.

Proof Let $f(\delta)=1+\frac{\delta}{\tau(1-\tau-\delta)}$. The function $f(\delta)$ strictly increases with δ. Solving $f(\delta)=6$ gives $\delta=\frac{5\tau(1-\tau)}{5\tau+1}$. Therefore, if $f(\delta)\geq 6$ then $\delta\geq\frac{5\tau(1-\tau)}{5\tau+1}$. Based on the definition of the learning problem, $\tau\geq\delta$. Therefore, $\delta\geq\frac{5\tau(1-\tau)}{5\tau+1}$ implies that $\tau\geq\frac{5\tau(1-\tau)}{5\tau+1}$.

Rearranging this inequality shows that $\tau \geq 2/5$. ■

Lemma B.7 *If* $1 + \frac{\delta}{\tau(1-\tau\delta)} \geq 6$ *then* $\delta \geq 5/14$.

Proof Let $f(\delta) = 1 + \frac{\delta}{\tau(1-\tau-\delta)}$. The function $f(\delta)$ strictly increases with δ. Solving $f(\delta) = 6$ gives $\delta = \frac{5\tau(1-\tau)}{5\tau+1}$. Therefore, if $f(\delta) \geq 6$ then $\delta \geq \frac{5\tau(1-\tau)}{5\tau+1}$. Using Lemma B.6, we know that $\tau \geq 2/5$. Therefore $\delta \geq \min_{\tau\in[2/5,1/2]} \frac{5\tau(1-\tau)}{5\tau+1} = 5/14$. ■

Our next lemma approximates the change in progress for a limited range of parameters and concepts. This includes $\theta = 1/2$. This is the default parameter setting used for Normalized Winnow before it was generalized to handle $\theta \in (0, 1)$ [Lit89].

Lemma B.8 *If* $\tau + \delta \geq 1/2$ *and* $\alpha = \min(6, 1 + \frac{\delta}{\tau(1-\tau-\delta)})$ *then*

$$\ln(\alpha)(\tau + \delta) - \ln(1 + (\alpha - 1)\tau) \geq 2\delta^2.$$

Proof Let $f(\delta) = \ln(\alpha)(\tau+\delta) - \ln(1 + (\alpha - 1)\tau)$. We break the proof into two cases. For the first case assume $\alpha < 6$. Substituting α into $f(\delta)$ gives

$$f(\delta) = \ln\left(1 + \frac{\delta}{\tau(1-\tau-\delta)}\right)(\tau + \delta) - \ln\left(1 + \frac{\delta}{1-\tau-\delta}\right).$$

Using Taylor's Theorem around $\delta = 0$ gives

$$f(\delta) = 0 + 0 + \frac{\delta^2}{2(\tau + z)(1 - \tau - z)} \text{ for } z \in (0, \delta).$$

The minimum value is obtained by maximizing $(\tau + z)(1 - \tau - z)$ which is achieved at $z = 1/2 - \tau$. Therefore $f(\delta) \geq 2\delta^2$.

For the second case, assume $\alpha = 6$. Therefore,

$$f(\delta) = \ln(6)(\tau + \delta) - \ln(1 + 5\tau).$$

Based on Lemma B.6, $\tau \geq 2/5$. The value of $\tau \in [2/5, 1/2]$ that minimizes the previous equation is $2/5$. Therefore,

$$\ln(6)(\tau + \delta) - \ln(1 + 5\tau) \geq \ln(6)\,\delta + \frac{2}{5}\ln(6) - \ln(3).$$

For $\delta \in [5/14, 1/2]$ the previous formula is greater than $2\delta^2$. Based on Lemma B.7, this proves the result. ■

The originally published mistake-bounds on Normalized Winnow also show the progress function changes by $2\delta^2$ [Lit89]. In that paper, a multiplier of $(1+2\delta)/(1-2\delta)$ is used. This is almost the same as our multiplier when $\theta = 1/2$. Our multiplier adds protection against noise by cutting off the value of α at 6. This cut-off lowers the change in progress, but the $2\delta^2$ approximation is still valid.

Our next lemma is slightly unconventional, in that we prove part of the approximation with a computer based proof. The proof was executed with the Maple$^{\text{TM}}$ program [MGH$^+$05].

Lemma B.9 *If* $\tau + \delta \geq 1/2$ *then*

$$\frac{\ln\left(\min\left(6, 1 + \frac{\delta}{\tau(1-\tau-\delta)}\right)\right)}{2\delta} \leq 2.53.$$

Proof We break the proof into two cases. First assume $\delta \leq 1/16$. Based on Lemma B.7,

$$\min\left(6, 1 + \frac{\delta}{\tau(1-\tau-\delta)}\right) = 1 + \frac{\delta}{\tau(1-\tau-\delta)}.$$

Using the fact that $\ln(1+x) \leq x$, we derive

$$\frac{\ln\left(\min\left(6, 1 + \frac{\delta}{\tau(1-\tau-\delta)}\right)\right)}{2\delta} = \frac{\ln\left(1 + \frac{\delta}{\tau(1-\tau-\delta)}\right)}{2\delta} \leq \frac{1}{2\tau(1-\tau-\delta)} \ .$$

Because $\tau + \delta \geq 1/2$, we know that $\tau \in [7/16, 1/2]$. With these constraints, the maximum of the last formula is achieved when $\tau = 1/2$ and $\delta = 1/16$. This gives us

$$\frac{\ln\left(\min\left(6, 1 + \frac{\delta}{\tau(1-\tau-\delta)}\right)\right)}{2\delta} \leq 16/7 \leq 2.29 \ .$$

For the second case assume that $\delta \geq 1/16$. We create a grid that covers $\delta \in [1/16, 1/2]$ and $\tau \in [1/2-\delta, 1/2]$. For each element of the grid we compute the maximum value of

$$\frac{\ln\left(\min\left(6, 1 + \frac{\delta}{\tau(1-\tau-\delta)}\right)\right)}{2\delta}.$$

```
highest:=0;
step:=1/256:
delta:=1/16:
f:=1+x/y/(1-y-x):
while delta<=1/2-step do
  tau:=1/2-delta:
  while tau<=1/2-step do
    d_min:=delta:
    d_max:=delta+step:
    t_min:=tau:
    t_max:=tau+step:
    a_max:=min(6,maximize(subs(x=d_max,f),y=t_min..t_max)):
    val:=log(a_max)/(2*d_min):
    real_val:=evalf(val):
    if real_val>highest then
      highest:=real_val:
    end if:
    tau:=tau+step:
  end do:
  delta:=delta+step:
end do:
print(highest):
```

Figure B.2: MAPLE code for Lemma B.9.

The MAPLE code for this case is in Figure B.2. The computation is exact until we compare with the current maximum value. At this stage, we convert to a floating point representation. However, the conversion to floating point only introduces a small round-off error because the exact value is of the form $a \ln b$ where a and b are fractions. This round-off error is accounted for in our final answer. The algorithm returns an upper-bound of 2.528929873 which we round to 2.53. ■

Next, we combine the previous results to give an approximation of maximum number of mistakes made by Normalized Winnow when $\tau + \delta \geq 1/2$.

Theorem B.10 *If* $\tau + \delta \geq 1/2$ *and* $\alpha = \min(6, 1 + \frac{\delta}{\tau(1-\tau-\delta)})$ *then the number of mistakes made by Normalized Winnow is at most*

$$\frac{\ln n + \sum_{i=1}^{n} u_i \ln u_i}{2\delta^2} + \frac{2.53N}{\delta}.$$

Proof Based on Theorem B.4 and Lemma B.8, the number of mistakes is at most

$$\frac{\ln n + \sum_{i=1}^{n} u_i \ln u_i}{2\delta^2} + \frac{\ln(\alpha) N}{2\delta^2}.$$

Using Lemma B.9 proves the result. ■

Next, we turn to the more interesting case where $\tau+\delta \leq 1/2$. We start with another lemma that is partially based on a computer proof. Again this helps us approximate the noise term in the mistake-bound.

Lemma B.11 *If* $\tau + \delta \leq 1/2$ *then*

$$\frac{2(\tau+\delta)(1-\tau-\delta)\ln\left(1+\frac{\delta}{\tau(1-\tau-\delta)}\right)}{\delta} \leq \min(2+2\delta/\tau, 2.8)$$

Proof We break the proof up into two cases. First assume $\delta/\tau \leq 1/4$. Using the fact that $\ln(1+x) \leq x$,

$$\frac{2(\tau+\delta)(1-\tau-\delta)\ln\left(1+\frac{\delta}{\tau(1-\tau-\delta)}\right)}{\delta} \leq \frac{2(\tau+\delta)}{\tau} = 2+2\delta/\tau \leq 2.5 \ .$$

Next assume that $\delta/\tau \geq 1/4$. We use the Maple$^{\text{TM}}$ program in Figure B.3 to generate an upper-bound. We create a grid looping over all $\tau \in [0, 1/4]$ and $\delta/\tau \in [1/4, 1]$. For each element of the grid we compute the maximum value of

$$\frac{2(1+\delta/\tau)(1-\tau-\delta)\ln\left(1+\frac{\delta}{\tau(1-\tau-\delta)}\right)}{\delta/\tau}.$$

The computation is exact until we compare with the current maximum value. At this stage, we convert to a floating point representation. However, the conversion to floating point only introduces a small round-off error because the exact value is of the form $a \ln b$ where a and b are fractions. This round-off error is accounted for in our final answer. The algorithm returns an upper-bound of 2.793773101 which we round to 2.8. ■

Theorem B.12 *If* $\tau + \delta \leq 1/2$ *and* $\alpha = \min(6, 1 + \frac{\delta}{\tau(1-\tau-\delta)})$ *then the number of mistakes made by Normalized Winnow is at most*

$$\frac{2(\tau+\delta)(1-\tau-\delta)(\ln n + \sum_{i=1}^{n} u_i \ln u_i)}{\delta^2} + \frac{\min(2+2\delta/\tau, 2.8)N}{\delta}.$$

```
highest:=0;
step:=1/256:
tau:=0:
while tau<=1/4-step do
  ratio:=1/4:
  while ratio<=1-step do:
    r_min:=ratio:
    r_max:=ratio+step:
    t_min:=tau:
    t_max:=tau+step:
    d_min:=r_min*t_min:
    d_max:=r_max*t_max:
    a_max:=1+r_max/(1-d_max-t_max):
    val:=2*(1+1/r_min)*(1-t_min-d_min)*log(a_max):
    real_val:=evalf(val):
    if real_val>highest then
      highest:=real_val:
    end if:
    ratio:=ratio+step:
  end do:
  tau:=tau+step:
end do:
print(highest):
```

Figure B.3: MAPLE code for Theorem B.11.

Proof Let $f(\delta) = \ln(\alpha)(\tau + \delta) - \ln(1 + (\alpha - 1)\tau)$. Recall that this is the progress made during a mistake from Lemma B.3 without the noise term. Based on Lemma B.7, $\alpha = 1 + \frac{\delta}{\tau(1-\tau-\delta)}$. Substituting this value into $f(\delta)$ we get

$$f(\delta) = \ln\left(1 + \frac{\delta}{\tau(1-\tau-\delta)}\right)(\tau + \delta) - \ln\left(1 + \frac{\delta}{1-\tau-\delta}\right).$$

Using Taylor's Theorem around $\delta = 0$ gives

$$f(\delta) = 0 + 0 + \frac{\delta^2}{2(\tau + z)(1 - \tau - z)} \text{ for } z \in (0, \delta).$$

The minimum value is obtained by maximizing $(\tau + z)(1 - \tau - z)$. Given the constraint that $\tau + \delta \leq 1/2$, the maximum of $(\tau + z)(1 - \tau - z)$ is achieved at $z = \delta$. Therefore,

$$f(\delta) \geq \frac{\delta^2}{2(\tau + \delta)(1 - \tau - \delta)}.$$

Combining this result with Theorem B.4, the number of mistakes made by Normalized Winnow is at most

$$\frac{2(\tau + \delta)(1 - \tau - \delta)(\ln n + \sum_{i=1}^{n} u_i \ln u_i)}{\delta^2} + \frac{2(\tau + \delta)(1 - \tau - \delta)\ln\left(1 + \frac{\delta}{\tau(1-\tau-\delta)}\right)N}{\delta^2}.$$

Using Lemma B.11 completes the proof. ■

The previous bound is significantly better than the mistake-bound given in Theorem B.10 as τ gets close to zero. While it appears that the noise term gets slightly worse in Theorem B.12 because of the 2.8 factor, this is an artifact of our approximation. In addition, smaller α values can be used to reduce the effects of noise.

Finally, we want to give a single bound to cover the two cases: $\tau + \delta \leq 1/2$ and $\tau + \delta \geq 1/2$. It is not as tight as the other bounds, but as we have explained, the purpose of these approximations is to make the bound more intuitive, and a single bound is easier to interpret.

Corollary B.13 *If* $\alpha = \min(6, 1 + \frac{\delta}{\tau(1-\tau-\delta)})$ *then Normalized Winnow makes at most*

$$\frac{2(\theta(1-\theta) + \delta|1-2\theta|)(\ln n + \sum_{i=1}^{n} u_i \ln u_i)}{\delta^2} + \frac{2.8N}{\delta}.$$

Proof The bound in Theorem B.12 handles the case $\tau + \delta \leq 1/2$. The bound is

$$\frac{2(\tau+\delta)(1-\tau-\delta)(\ln n + \sum_{i=1}^{n} u_i \ln u_i)}{\delta^2} + \frac{\min(2 + 2\delta/\tau, 2.8)N}{\delta}.$$

The factor $2(\tau + \delta)(1 - \tau - \delta)$ in the target function term equals

$$2\tau(1-\tau) + 2\delta(1-2\tau) - 2\delta^2 = 2\theta(1-\theta) + 2\delta|1-2\theta| - 2\delta^2 \leq 2\theta(1-\theta) + 2\delta|1-2\theta|.$$

The bound in Theorem B.10 handles the case $\tau + \delta \geq 1/2$. The bound is

$$\frac{\ln n + \sum_{i=1}^{n} u_i \ln u_i}{2\delta^2} + \frac{2.53N}{\delta}.$$

Next, we prove that $2\theta(1-\theta) + 2\delta|1-2\theta| \geq 1/2$ when $\tau + \delta \geq 1/2$. This means that this term can be used in place of the $1/2$ factor in the target function term. We break the proof into two cases. First assume $\theta \in (0, 1/2]$. Therefore, because δ is at least $1/2 - \tau = 1/2 - \theta$,

$$2\theta(1-\theta) + 2\delta|1-2\theta| \geq 2\theta(1-\theta) + 2(1/2-\theta)(1-2\theta) = 2\theta^2 - 2\theta + 1 \geq 1/2$$

Second, assume $\theta \in [1/2, 1)$. Therefore, because δ is at least $1/2 - \tau = 1/2 - (1-\theta) = \theta - 1/2$,

$$2\theta(1-\theta) + 2\delta|1-2\theta| \geq 2\theta(1-\theta) + 2(\theta-1/2)(2\theta-1) = 2\theta^2 - 2\theta + 1 \geq 1/2.$$

The noise term is based on the maximum noise term from Theorem B.10 and Theorem B.12. Combined with the target function term, this proves the corollary. ■

In this appendix, our main result is an upper-bound on the number of mistakes made by the Normalized Winnow algorithm. This version of the algorithm allows a threshold parameter that significantly improves the performance for some learning problems. We also show that the algorithm exhibits optimal behavior for noisy instances as the weight multiplier α approaches 1.

Appendix C

Balanced Winnow

In this appendix, we prove an upper-bound on the number of mistakes made by the Balanced Winnow algorithm. In addition, we show that the bound can not always be tight and give some examples of where the algorithm may do better than predicted by the mistake bound. We also show that the algorithm exhibits optimal behavior for noisy instances as the weight multiplier, α approaches 1. To clarify these results, we restate the algorithm in Figure C.1. This is identical to the algorithm found in Chapter 2. For the following proofs define $0 \ln 0 = 0$.

The proof for the mistake bound is similar to the result presented by Littlestone in [Lit89]. The main difference is that we modify the noise analysis used in [Lit89] to a form based on the hinge-loss [GW99]. All other changes are purely notational.

First, we need to specify the noise function that controls the generation of the instances. Let $\mathbf{u}^+$ and $\mathbf{u}^-$ be the weights for the target function where $u_i^+ \geq 0$, $u_i^- \geq 0$, and $\sum_{i=1}^{n} u_i^+ + u_i^- = 1$. The noise on instance x_t is defined as $\nu_t = \max(0, \delta - y_t(\mathbf{u}^+ \cdot \mathbf{x_t} - \mathbf{u}^- \cdot \mathbf{x_t}))$ where $\delta > 0$ is the margin. The total noise up to trial T is $N = \sum_{t=1}^{T} \nu_t$.

The derivation of the mistake-bound is based on a progress function. This progress function increases on every mistake by at least a fixed amount and has a maximum value. The only exception is for noisy instances, which can cause a decrease in the progress function. However, since we assume a upper-bound on the noise, the net effect is to increase the total amount of progress we must make until we we reach the maximum value of the progress function.

The progress function is $Q(t) = \sum_{i=1}^{n}(u_i^+ \ln w_{i,t}^+ + u_i^- \ln w_{i,t}^-) - \ln\left(\sum_{i=1}^{n} w_{i,t}^+ + w_{i,t}^-\right)$ where $\mathbf{u}^+$ and $\mathbf{u}^-$ are the weights from the target function; $\mathbf{w}_t^+$ and $\mathbf{w}_t^-$ are the weights

Balanced Winnow(α)
Parameters
$\alpha > 1$ is the update multiplier.
Initialization
$t \leftarrow 1$ is the current trial.
$\forall i \in \{1, \ldots, n\}$ $w^+_{i,1} = 1$ are the positive weights.
$\forall i \in \{1, \ldots, n\}$ $w^-_{i,1} = 1$ are the negative weights.
Trials
Instance: $\mathbf{x}_t \in [0,1]^n$.
Prediction: If $\mathbf{w}^+_\mathbf{t} \cdot \mathbf{x_t} \geq \mathbf{w}^-_\mathbf{t} \cdot \mathbf{x_t}$
Predict $\hat{y}_t = 1$ else predict $\hat{y}_t = -1$.
Update: Let $y_t \in \{-1, 1\}$ be the correct label.
If $y_t(\mathbf{w}^+_\mathbf{t} \cdot \mathbf{x_t} - \mathbf{w}^-_\mathbf{t} \cdot \mathbf{x_t}) \leq 0$
$\forall i \in \{1, \ldots, n\}$ $w^+_{i,t+1} = \alpha^{y_t x_{i,t}} w^+_{i,t}$ and $w^-_{i,t+1} = \alpha^{-y_t x_{i,t}} w^-_{i,t}$.
Else
$\forall i \in \{1, \ldots, n\}$ $w^+_{i,t+1} = w^+_{i,t}$ and $w^-_{i,t+1} = w^-_{i,t}$.
$t \leftarrow t + 1$.

Figure C.1: Pseudo-code for the Balanced Winnow algorithm.

from the algorithm at the start of trial t [Lit89]. The progress function can only change after a mistake because that is the only time the Balanced Winnow algorithm changes its weights. Our first lemma bounds this change in progress.

Lemma C.1 *After a mistake on trial t,*

$$Q(t+1) - Q(t) \geq \delta \ln(\alpha) - \ln\left(\frac{\alpha^2 + 1}{2\alpha}\right) - \ln(\alpha)\,\nu_t.$$

Proof Based on the definition of $Q(t)$, after a mistake

$$Q(t+1) - Q(t) = \sum_{i=1}^{n} \left[u_i^+ \ln\left(\frac{w^+_{i,t+1}}{w^+_{i,t}}\right) + u_i^- \ln\left(\frac{w^-_{i,t+1}}{w^-_{i,t}}\right)\right] - \ln\left(\frac{\sum_{i=1}^n (w^+_{i,t+1} + w^-_{i,t+1})}{\sum_{i=1}^n (w^+_{i,t} + w^-_{i,t})}\right)$$

$$= \sum_{i=1}^{n} \left[u_i^+ \ln \alpha^{y_t x_{i,t}} + u_i^- \ln \alpha^{-y_t x_{i,t}}\right] - \ln\left(\frac{\sum_{i=1}^n (w^+_{i,t}\alpha^{y_t x_{i,t}} + w^-_{i,t}\alpha^{-y_t x_{i,t}})}{\sum_{i=1}^n (w^+_{i,t} + w^-_{i,t})}\right).$$

Based on the convexity of α^x, $\alpha^{y_t x_{i,t}} \leq 1 + (\alpha^{y_t} - 1)x_{i,t}$ for all $x_{i,t} \in [0,1]$ and $y_t \in \{-1, 1\}$. Using this fact, the preceding is greater than or equal to

$$y_t \ln(\alpha) \sum_{i=1}^{n} (u^+_{i,t} - u^-_{i,t}) x_{i,t} - \ln\left(\frac{\sum_{i=1}^n [w^+_{i,t}(1 + (\alpha^{y_t} - 1)x_{i,t}) + w^-_{i,t}(1 + (\alpha^{-y_t} - 1)x_{i,t})]}{\sum_{i=1}^n (w^+_{i,t} + w^-_{i,t})}\right)$$

$$= y_t \ln(\alpha) \sum_{i=1}^{n} (u^+_{i,t} - u^-_{i,t}) x_{i,t} - \ln\left(1 + \frac{(\alpha^{y_t} - 1)\sum_{i=1}^n w^+_{i,t} x_{i,t} + (\alpha^{-y_t} - 1)\sum_{i=1}^n w^-_{i,t} x_{i,t}}{\sum_{i=1}^n (w^+_{i,t} + w^-_{i,t})}\right).$$

We can break the rest of the proof into two cases. First assume $y_t = -1$. In this case, $\sum_{i=1}^{n}(u_i^+ x_{i,t} - u_i^- x_{i,t}) \leq -\delta + \nu_t$ and $\sum_{i=1}^{n} w_{i,t}^- x_{i,t} \leq \sum_{i=1}^{n} w_{i,t}^+ x_{i,t}$. Plugging these into the previous formula gives

$$-\ln(\alpha)(\nu_t - \delta) - \ln\left(1 + \frac{(\alpha^{-1} - 1 + \alpha - 1)\sum_{i=1}^{n} w_{i,t}^- x_{i,t}}{\sum_{i=1}^{n}(w_{i,t}^+ + w_{i,t}^-)}\right). \tag{C.1}$$

Notice that $\frac{\sum_{i=1}^{n} w_{i,t}^- x_{i,t}}{\sum_{i=1}^{n}(w_{i,t}^+ + w_{i,t}^-)} \leq 1/2$ because the maximum value of $\sum_{i=1}^{n}(w_{i,t}^+ x_{i,t} + w_{i,t}^- x_{i,t})$ is $\sum_{i=1}^{n}(w_{i,t}^+ + w_{i,t}^-)$ and $\sum_{i=1}^{n} w_{i,t}^- x_{i,t} \leq \sum_{i=1}^{n} w_{i,t}^+ x_{i,t}$. Therefore the previous equation is greater than or equal to

$$-\ln(\alpha)(\nu_t - \delta) - \ln\left(1 + \frac{(\alpha^{-1} - 1 + \alpha - 1)}{2}\right) = \delta\ln(\alpha) - \ln\left(\frac{\alpha^2 + 1}{2\alpha}\right) - \ln(\alpha)\nu_t.$$

The case for $y_t = 1$ is symmetrical. In this case, $\sum_{i=1}^{n}(u_i^+ x_{i,t} - u_i^- x_{i,t}) \geq \delta - \nu_t$ and $\sum_{i=1}^{n} w_{i,t}^- x_{i,t} \geq \sum_{i=1}^{n} w_{i,t}^+ x_{i,t}$. Plugging these into the lower-bound from above gives

$$\ln(\alpha)(\delta - \nu_t) - \ln\left(1 + \frac{(\alpha - 1 + \alpha^{-1} - 1)\sum_{i=1}^{n} w_{i,t}^+ x_{i,t}}{\sum_{i=1}^{n}(w_{i,t}^+ + w_{i,t}^-)}\right). \tag{C.2}$$

Using the fact that $\frac{\sum_{i=1}^{n} w_{i,t}^+ x_{i,t}}{\sum_{i=1}^{n}(w_{i,t}^+ + w_{i,t}^-)} \leq 1/2$, we lower-bound the previous formula with

$$\ln(\alpha)(\delta - \nu_t) - \ln\left(1 + \frac{(\alpha - 1 + \alpha^{-1} - 1)}{2}\right) = \delta\ln(\alpha) - \ln\left(\frac{\alpha^2 + 1}{2\alpha}\right) - \ln(\alpha)\nu_t.$$

■

In the previous lemma, we use the fact that $\frac{\sum_{i=1}^{n} w_{i,t}^- x_{i,t}}{\sum_{i=1}^{n}(w_{i,t}^+ + w_{i,t}^-)} \leq 1/2$ in equation C.1 when $y = -1$, and the fact that $\frac{\sum_{i=1}^{n} w_{i,t}^+ x_{i,t}}{\sum_{i=1}^{n}(w_{i,t}^+ + w_{i,t}^-)} \leq 1/2$ in equation C.2 when $y = 1$. It is impossible for the adversary to make these conditions tight for many problems. The only way an adversary can make these conditions tight is to make every $w_{i,t} > 0$ have a corresponding $x_{i,t} = 1$.

For example, if the problem consists of sparse instances where $\|X_t\|_1$ is small compared to n then during the starting trials both of the previous formulas must be significantly less than $1/2$. This is because the initial weight of each attribute is 1. Later, if the target function is a disjunction then these conditions still can not be made tight. As the algorithm gets close to learning the disjunction, most of the weight must be distributed on the relevant variables in the disjunction. However, for this very reason the

adversary can not make $x_i = 1$ for all of these relevant attributes since the algorithm would not make a mistake.

For many problems, the lower-bound on progress in Lemma C.1 is not tight. The Balanced Winnow algorithm will make more progress when $\frac{\sum_{i=1}^{n} w_{i,t}^{+} x_{i,t} + \sum_{i=1}^{n} w_{i,t}^{-} x_{i,t}}{\sum_{i=1}^{n} (w_{i,t}^{+} + w_{i,t}^{-})} < 1$. The amount of extra progress depends on the setting of α and can be computed using Equation C.1 and C.2 from Lemma C.1.

Next we give a proof of the mistake bound. At this point, we do not substitute in a value for α. Later, this will allow us to analyze the behavior of the algorithm as α approaches 1. Recall that we defined M as the set of trials where the algorithm makes a mistake.

Theorem C.2 *If* $\delta \ln(\alpha) - \ln\left(\frac{\alpha^2+1}{2\alpha}\right) > 0$ *then the number of mistakes made by Balanced Winnow is at most*

$$\frac{\ln(2n) + \sum_{i=1}^{n} (u_i^{+} \ln u_i^{+} + u_i^{-} \ln u_i^{-})}{\delta \ln(\alpha) - \ln\left(\frac{\alpha^2+1}{2\alpha}\right)} + \frac{\ln(\alpha)\, N}{\delta \ln(\alpha) - \ln\left(\frac{\alpha^2+1}{2\alpha}\right)}.$$

Proof The maximum value of $Q(t) = \sum_{i=1}^{n} (u_i^{+} \ln w_{i,t}^{+} + u_i^{-} \ln w_{i,t}^{-}) - \ln\left(\sum_{i=1}^{n} w_{i,t}^{+} + w_{i,t}^{-}\right)$ can be determined by taking it's derivative with respect to the algorithm weights and setting these equations to zero. The maximum is achieved when $\frac{w_{i,t}^{+}}{\sum_{i=1}^{n} (w_{i,t}^{+} + w_{i,t}^{-})} = u_i^{+}$ and $\frac{w_{i,t}^{-}}{\sum_{i=1}^{n} (w_{i,t}^{+} + w_{i,t}^{-})} = u_i^{-}$. This gives a maximum $Q(t)$ value of $\sum_{i=1}^{n} (u_i^{+} \ln u_i^{+} + u_i^{-} \ln u_i^{-})$. Therefore, for any trial T,

$$\sum_{i=1}^{n} (u_i^{+} \ln u_i^{+} + u_i^{-} \ln u_i^{-}) + \ln 2n \geq Q(T+1) - Q(1)$$

$$= \sum_{t=1}^{T} Q(t+1) - Q(t) = \sum_{t \in M} Q(t+1) - Q(t).$$

Based on Lemma C.1,

$$\sum_{t \in M} Q(t+1) - Q(t) \geq \sum_{t \in M} \left[\delta \ln(\alpha) - \ln\left(\frac{\alpha^2+1}{2\alpha}\right) - \ln(\alpha)\, \nu_t \right].$$

$$= \left[\delta \ln(\alpha) - \ln\left(\frac{\alpha^2+1}{2\alpha}\right) \right] |M| - \ln(\alpha) \sum_{t \in M} \nu_t$$

$$\geq \left[\delta \ln(\alpha) - \ln\left(\frac{\alpha^2+1}{2\alpha}\right) \right] |M| - \ln(\alpha)\, N.$$

Therefore,

$$\sum_{i=1}^{n}(u_i^+ \ln u_i^+ + u_i^- \ln u_i^-) + \ln 2n \geq \left[\delta \ln(\alpha) - \ln\left(\frac{\alpha^2+1}{2\alpha}\right)\right]|M| - \ln(\alpha)\, N.$$

Rearranging this inequality proves the theorem. ■

A good choice for α is $\sqrt{\frac{1+\delta}{1-\delta}}$. With this choice, we get the following upper-bound on the number of mistakes.

Theorem C.3 *If $\alpha = \sqrt{\frac{1+\delta}{1-\delta}}$ then the number of mistakes made by Balanced Winnow is at most*

$$\frac{2\ln 2n + 2\sum_{i=1}^{n}(u_i^+ \ln u_i^+ + u_i^- \ln u_i^-)}{\delta^2} + \frac{2\left(1 + \frac{2\delta^2}{5(1-\delta)^2}\right)}{\delta} N.$$

Proof The bound in Theorem C.2 has two terms. The first term is

$$\frac{\sum_{i=1}^{n}(u_i^+ \ln u_i^+ + u_i^- \ln u_i^-) + \ln 2n}{\delta \ln(\alpha) - \ln\left(\frac{\alpha^2+1}{2\alpha}\right)} \tag{C.3}$$

and deals with learning the concept. To simplify this term substitute $\alpha = \sqrt{\frac{1+\delta}{1-\delta}}$ and let

$$f(\delta) = \delta \ln(\alpha) - \ln\left(\frac{\alpha^2+1}{2\alpha}\right) = \frac{\delta}{2}\ln\left(\frac{1+\delta}{1-\delta}\right) + \frac{\ln\left(1-\delta^2\right)}{2}.$$

Using Taylor's Theorem around $\delta = 0$, we get

$$f(\delta) = 0 + 0 + \frac{\delta^2}{2(1-z^2)} \text{ for } z \in (0, \delta).$$

Therefore $f(\delta) > \delta^2/2$. Substituting this into the equation C.3 gives the first term of the bound. This also shows that $\delta \ln(\alpha) - \ln\left(\frac{\alpha^2+1}{2\alpha}\right) > 0$ which is a requirement of Theorem C.2.

The second term

$$\frac{\ln(\alpha)\, N}{\delta \ln(\alpha) - \ln\left(\frac{\alpha^2+1}{2\alpha}\right)}. \tag{C.4}$$

deals with the noise. Using the result from the previous term, let

$$g(\delta) = \frac{\ln(\alpha)}{\delta \ln(\alpha) - \ln\left(\frac{\alpha^2+1}{2\alpha}\right)} = \frac{\frac{1}{2}\ln\left(\frac{1+\delta}{1-\delta}\right)}{f(\delta)} \leq \frac{\frac{1}{2}\ln\left(\frac{1+\delta}{1-\delta}\right)}{\frac{\delta^2}{2}} = \frac{\ln\left(\frac{1+\delta}{1-\delta}\right)}{\delta^2}.$$

Next, we use the result that $\ln(1+x) \leq \frac{2x(1+x^2/10)}{2+x}$ from Lemma A.1. This show that

$$\frac{\ln\left(\frac{1+\delta}{1-\delta}\right)}{\delta^2} = \frac{\ln\left(1+\frac{2\delta}{1-\delta}\right)}{\delta^2} \leq \frac{\frac{4\delta}{1-\delta}\left(1+\frac{4\delta^2}{10(1-\delta)^2}\right)}{\left(2+\frac{2\delta}{1-\delta}\right)\delta^2} = \frac{2\left(1+\frac{2\delta^2}{5(1-\delta)^2}\right)}{\delta}.$$

This completes the theorem. ■

The choice of $\alpha = \sqrt{\frac{1+\delta}{1-\delta}}$ is optimal for the target function term in the previous mistake bound, but it is not necessarily optimal when noise is considered. As α approaches 1 the noise term becomes optimal as explained in Section 2.2.

Corollary C.4 *As α approaches 1, the number of mistakes made by Balanced Winnow is at most*

$$\frac{\ln(2n) + \sum_{i=1}^{n}(u_i^+ \ln u_i^+ + u_i^- \ln u_i^-)}{\delta \ln(\alpha) - \ln\left(\frac{\alpha^2+1}{2\alpha}\right)} + \frac{N}{\delta}.$$

Proof In order for Theorem C.2 to apply, we first show that $\delta \ln(\alpha) - \ln\left(\frac{\alpha^2+1}{2\alpha}\right) > 0$. Using the fact that $x/(1+x) \leq \ln(1+x) \leq x$,

$$\delta \ln(\alpha) - \ln\left(\frac{\alpha^2+1}{2\alpha}\right) \geq \frac{\delta(\alpha-1)}{\alpha} - \frac{(\alpha-1)^2}{2\alpha} = \frac{\alpha-1}{\alpha}\left(\delta - \frac{\alpha-1}{2}\right).$$

This is positive as long as $\alpha > 1$ and $\alpha < 1 + 2\delta$.

Next, we deal with the noise term from Theorem C.2. The noise term is

$$\frac{\ln(\alpha)\, N}{\delta \ln(\alpha) - \ln\left(\frac{\alpha^2+1}{2\alpha}\right)} = \frac{N}{\delta - \frac{\ln\left(\frac{\alpha^2+1}{2\alpha}\right)}{\ln\alpha}}.$$

Using L'Hôpital's rule,

$$\lim_{\alpha\to 1^+} \frac{\ln\left(\frac{\alpha^2+1}{2\alpha}\right)}{\ln\alpha} = \lim_{\alpha\to 1^+} \frac{\alpha^2-1}{\alpha^2+1} = 0.$$

Therefore, the noise term approaches N/δ. ■

In the previous corollary, we were only concerned with the noise term. As α goes to 1, the first part of the bound, that deals with learning the target function, goes to infinity. A trade-off is needed. In Theorem C.3, because the maximum value of δ is

1/2, the noise term can be at most $2.8N/\delta$. In truth, this approximation is slightly misleading, and the maximum of the noise term is only $2.2N/\delta$. Therefore, even the optimal value of α causes only a 2.2 factor in the effect of the noise term. However, for problems with a large amount of noise, this can be significant. In practice, one should use several α values to find the one that gives the best mistake bound. See Section 2.5 for more details on parameter selection.

In this appendix, we prove an upper-bound on the number of mistakes made by the Balanced Winnow algorithm. This is essentially the same proof and bound given in [Lit89] with the addition of a noise analysis based on the hinge loss. However, we use this proof to give some new insight into the Balanced Winnow algorithm. We show that the bound can not always be tight and give some examples of where the algorithm may do better than predicted by the mistake bound. We also show that the algorithm exhibits optimal behavior for noisy instances as the weight multiplier, α approaches 1.

Appendix D

Sparse Instances

All the algorithms we covered in Chapter 2 have a straightforward implementation that performs predictions and updates in $O(n)$. In this appendix, we give implementations for linear-threshold learning algorithms that are more efficient when most of the attributes of the instances have a value of zero. We call these sparse instance problems.

An example of a sparse instance problem is the bag of words representation for a text problem. With the bag of words representation, an instance can have at most the word length of the document non-zero attributes, but the size of the instance can be on the order of the number of words in the dictionary. Therefore, for any given instance, most of the attributes have value zero.

The main idea behind these implementations is to exploit the fact that attributes with an identical value have a similar effect when predicting and updating. Instead of manipulating each weight, we keep track of their aggregate effect and only explicitly deal with the attributes that deviate from the fixed value. These implementations are based on a suggestion by Nick Littlestone that the Unnormalized Winnow algorithm could be implemented in a form that is efficient on problems where few attributes are non-zero even with complemented attributes [Lit97].

In order to speed up the implementation of a linear-threshold algorithm, we need to compress the size of the instances. If the instance representation is $O(n)$ then the time spent reading the instance may dominate any improvement in prediction and updating. Given our assumption that many of the attributes have a value of zero, we represent an instance using only the attributes that are non-zero. We call these the active attributes.

Let m_t be the number of active attributes on trial t. For each active attribute, the instance encodes the position of the attribute in the attribute vector and the value of

the attribute. This gives a total storage requirement of $O(m_t \log n)$. In practice, the required storage can be approximated as $O(m_t)$ because the largest practical values of n are small enough to be represented by an integer in a modern computer architecture. Therefore, we make the simplifying assumption that an instance only takes $O(m_t)$ storage.

For some problems, the algorithm may need to spend $\omega(n)$ time to generate the instances. For these types of problems, the sparse representation is less beneficial. For example, if the attributes are computed based on a digitized picture then many of the attributes might be zero for a problem such as optical character recognition. However, most algorithms still need to perform work proportional to the number of pixels. The algorithm might even perform complex computations on the raw attributes to calculate new attributes. In this dissertation, we do not consider the cost of generating the instances, but in practice it may effect the overall efficiency of learning

D.1 Perceptron and Balanced Winnow

Perceptron and Balanced Winnow require little modification to perform predictions and updates in $O(m_t)$ time. The Perceptron algorithm is explained in Section 2.2, and the Balanced Winnow algorithm is explained in Section 2.3.4. Both algorithms make predictions using the dot product of the weight and instance vectors. Using the sparse instance representation, this can be computed in $O(m_t)$ time. Both algorithms only change the weights that correspond to attributes that are non-zero. Again using the sparse instance representation, this can be computed in $O(m_t)$ time. The sparsity advantage of these algorithms was originally mentioned in [GR96].

D.2 Unnormalized Winnow

Information on Unnormalized Winnow can be found in Section 2.3.1. Unnormalized Winnow has the same sparsity properties as Perceptron and Balanced Winnow. Therefore, the algorithm performs predictions and updates in $O(m_t)$ time. Unfortunately, this form of the algorithm does not allow negative weights. To allow negative weights, the

algorithm must add attribute $\bar{x}_i = 1 - x_i$ for every attribute x_i. We call $\bar{\mathbf{x}}$ the complemented attributes [Lit89] and call $\mathbf{x}$ the normal attributes. Fortunately, Unnormalized Winnow using both the normal and complemented attributes can be implemented to perform predictions and updates in $O(m_t)$ time [Lit97].

The complemented attributes destroy the natural sparsity of a problem. With complemented attributes every instance has at least n attributes that are non-zero. However, these complemented attributes have a large amount of structure. They are completely determined by the values of the normal attributes. Therefore, we do not encode the complemented attributes into the instance representation. Their value is implied by the values of the normal attributes. Whenever we talk about an active attribute, we are only referring to the normal attributes; when $x_i = 0$, $\bar{x}_i = 1$, but we do not call the complemented attributes active.

In Figure D.1, we give pseudo-code for Unnormalized Winnow with complemented attributes that performs predictions and updates in $O(m_t)$. Notice that we do not explicitly keep track of all the weight values since at least n values change every update. Instead, we compute the weight values as needed using s_i and U. The value of s_i keeps track of the effect of updates on active attributes. The value of U keeps track of the updates on inactive attributes. The value of w_i is $\sigma\alpha^{s_i}$ because only the active attributes effect the normal weights. The value of w_i^c is $\sigma\alpha^{U-s_i}$ because U encodes the effect on a complemented attribute assuming it is 1 on every update. The -s_i factor corrects for the times that the complemented attribute is not 1.

Unnormalized Winnow also needs to keep track of the sum of the complemented weights, W^c. This is needed for prediction since a large number of complemented attributes have value 1 and are used in the prediction. The algorithm predicts 1 if and only if $W^c + \sum_{i \in \text{active}} (w_i - w_i^c) x_{i,t} \geq 1$. The $\sum_{i \in \text{active}} (w_i - w_i^c) x_{i,t}$ term includes all the normal weights from active attributes and subtracts off any extra weight from complemented attributes that are not 1. The update procedure makes the necessary changes to U and $\mathbf{s}$. It also updates the sum of the complemented weights by breaking the sum into two terms. First it computes the new weight total for the inactive attributes; second it computes the new weight total for the active attributes.

Unnormalized Winnow(α, σ)

Parameters

$\alpha > 1$ is the update multiplier.

$\sigma > 0$ is the initial weight value.

Initialization

$t \leftarrow 1$ is the current trial.

$U \leftarrow 0$ keeps track of the updates.

$W^c \leftarrow n\sigma$ is the sum of the complemented weights.

$s_i \leftarrow 0$ is the sum of $y_t x_{i,t}$ for each attribute.

Trials

Instance: $\mathbf{x}_t \in [0,1]^n$. For all $x_{i,t} > 0$, $w_i \leftarrow \sigma\alpha^{s_i}$ and $w_i^c \leftarrow \sigma\alpha^{U-s_i}$

Prediction: If $W^c + \sum_{i \in \text{active}} (w_i - w_i^c) x_{i,t} \geq 1$ then
predict $\hat{y}_t = 1$ else predict $\hat{y}_t = -1$.

Update: Let $y_t \in \{-1, 1\}$ be the correct label.

If $y_t(\mathbf{w_t} \cdot \mathbf{x_t}) \leq 0$ and $y_t = -1$ then

$U \leftarrow U + y_t$.

$\mathbf{s} \leftarrow \mathbf{s} + y_t \mathbf{x}$.

$W^c \leftarrow (W^c - \sum_{i \in \text{active}} w_i^c)\alpha^{y_t} + \sum_{i \in \text{active}} \alpha^{y_t(1-x_i)} w_i^c$.

$t \leftarrow t + 1$.

Figure D.1: Pseudo-code for Sparse Implementation of Unnormalized Winnow algorithm.

This implementation of Normalized Winnow is based on the fact there are groups of attributes where only a small number deviate from a fixed value. In our case, the normal attributes have fixed value 0 and the complemented attributes have fixed value 1. In general, we can implement Normalized Winnow for any number of groups where each group has its own fixed value. The cost of the algorithm is proportional to the total number of attributes that deviate from their fixed value plus the number of groups.

D.3 Normalized Winnow

Information on Normalized Winnow can be found in Section 2.3.3. In Figure D.2, we give the pseudo-code for Normalized Winnow that performs predictions and updates in $O(m_t)$ with complemented attributes. The code and explanation is almost identical to Unnormalized Winnow. The only exception is that we also need to keep track of the sum of the normal weights because of the normalization used in prediction. This is straightforward to implement since only the active attributes affect the normal weights.

Normalized Winnow(α, θ)

Parameters

$\alpha > 1$ is the update multiplier.

$0 < \theta < 1$ is the threshold.

Initialization

$t \leftarrow 1$ is the current trial.

$U \leftarrow 0$ keeps track of the updates.

$W \leftarrow n$ is the sum of the normal weights.

$W^c \leftarrow n$ is the sum of the complemented weights.

$s_i \leftarrow 0$ is the sum of $y_t x_{i,t}$ for each attribute.

Trials

Instance: $\mathbf{x}_t \in [0,1]^n$. For all $x_{i,t} > 0$, $w_i \leftarrow \alpha^{s_i}$ and $w_i^c \leftarrow \alpha^{U - s_i}$

Prediction: If $W^c + \sum_{i \in \text{active}} (w_i - w_i^c) x_{i,t} \geq \theta (W + W^c)$ then predict $\hat{y}_t = 1$ else predict $\hat{y}_t = -1$.

Update: Let $y_t \in \{-1, 1\}$ be the correct label.

If $y_t(\mathbf{w_t} \cdot \mathbf{x_t}) \leq 0$ and $y_t = -1$ then

$U \leftarrow U + y_t$.

$\mathbf{s} \leftarrow \mathbf{s} + y_t \mathbf{x}$.

$W \leftarrow W + \sum_{i \in \text{active}} w_i (\alpha^{y_t x_{i,t}} - 1)$.

$W^c \leftarrow (W^c - \sum_{i \in \text{active}} w_i^c) \alpha^{y_t} + \sum_{i \in \text{active}} \alpha^{y_t (1 - x_i)} w_i^c$.

$t \leftarrow t + 1$.

Figure D.2: Pseudo-code for Sparse Implementation of Normalized Winnow algorithm.

One possible modification is to add an attribute that is always 1. This increases m_t by one but allows the algorithm to represent a wider range of concepts as explained in Section 2.3.3.

D.4 ALMA

Information on the ALMA algorithm can be found in Section 2.3.5. Here we show how to implement the algorithm to take advantage of sparse instances. The main problem is that updates affect all the weights even if only a small number of attributes are non-zero. Just as with the Unnormalized Winnow, we use an extra variables to keep track of common changes to all the weights.

The pseudo-code for the sparse efficient implementation of ALMA is in Figure D.3. The algorithm performs predictions and updates in $O(m_t)$ time using the sparse instance representation. Unlike our original explanation of ALMA, the weights are not explicitly

stored. Instead they are computed using vector $\mathbf{v}$. These variable are related to the code given in Section 2.3.5 by the equality $z_i = v_i / v_{mult}$. Therefore v_{mult} keeps track of any multiplicative change on all the z_i variables. The algorithm also keeps track of $v_{sum1} = \|\mathbf{v_t}\|_p^p$ and $v_{sum2} = \|\mathbf{v_t}\|_{(p-1)q}^{(p-1)q}$. These are necessary to compute the various norms used by the algorithm. These norms allow the algorithm to compute the weights and normalize the weights.

D.5 Mistake-Driven Linear-threshold Algorithms

Some algorithms might be impossible to implement in a form that performs predictions and updates in $o(n)$ time. However, it still might be possible to get some performance improvement. Mistake-driven algorithms only perform updates on mistakes. For problems with few mistakes, the prediction procedure is executed much more frequently than the update procedure. Therefore making the prediction procedure more efficient can dramatically reduce the cost on some problems.

Linear-threshold algorithms are generally easy to modify to perform predictions in $O(m_t)$ when using sparse instances. The algorithm can spend a small amount of extra effort during the update procedure to put the weights in a form that makes the predictions more efficient. For example, if the algorithm uses complemented attributes then we can expand $w_{i,t}^c(1 - x_i)$ and subtract $w_{i,t}^c$ from $w_{i,t}$ and subtract $w_{i,t}^c$ from the threshold. This removes the need to use the complemented attributes for prediction since their weight has been moved to the normal attributes. Using the sparse instance representation, the algorithm can perform the prediction in $O(m_t)$ using these weights.

We use a similar trick with the Tracking Unnormalized Winnow algorithm of Chapter 6 to perform updates in $O(n)$ and predictions in $O(m_t)$. In general, this trick can be applied to speed up prediction for problems with groups of attributes where each group has a nominal value and only a small number number of attributes deviate from the corresponding nominal value.

ALMA(p, B, C)

Parameters

$B \geq 0$ controls the algorithm margin.

$C > 0$.

$p \geq 2$ and $q = \frac{p}{p-1}$ control the norms.

Initialization

$t \leftarrow 1$ is the current trial.

$k = 0$ is the number of updates.

$v_{i,1} = 0$ stores information on weights.

$v_{sum1} \leftarrow 0$ corresponds to $\|\mathbf{v_t}\|_p^p$.

$v_{sum2} \leftarrow 0$ corresponds to $\|\mathbf{v_t}\|_{(p-1)q}^{(p-1)q}$.

$v_{mult} \leftarrow 1$.

Trials

Instance: $\mathbf{x}_t \in [0,1]^n$.

Prediction: For $x_{i,t} > 0$ compute $w_{i,t} = \dfrac{sign(v_{i,t})|v_{i,t}|^{p-1}}{v_{mult} v_{sum1}^{(p-2)/p}}$.

If $\mathbf{w_t} \cdot \mathbf{x_t} \geq 0$ then predict $\hat{y}_t = 1$ else predict $\hat{y}_t = -1$.

Update: Let $y_t \in \{-1, 1\}$ be the correct label and $\hat{\delta} = B\|\mathbf{x_t}\|_p \sqrt{\frac{p-1}{k}}$.

If $y_t(\mathbf{w_t} \cdot \mathbf{x_t}) \leq \hat{\delta}$ then

$\eta \leftarrow \dfrac{C}{\sqrt{k(p-1)}\|\mathbf{x_t}\|_p}$

$v_{i,t+1} = v_{i,t} + v_{mult}\eta y_t x_{i,t}$.

$v_{sum1} \leftarrow v_{sum1} + \sum_{i \in active} \left(v_{i,t+1}^p - v_{i,t}^p \right)$

$v_{sum2} \leftarrow v_{sum2} + \sum_{i \in active} \left(v_{i,t+1}^{(p-1)q} - v_{i,t}^{(p-1)q} \right)$

$\|\mathbf{w_t}\|_q = \dfrac{v_{sum2}^{1/q}}{v_{mult} v_{sum1}^{(p-2)/p}}$.

If $\|\mathbf{w_t}\|_q > 1$ then

$v_{mult} \leftarrow v_{mult}\|\mathbf{w_t}\|_q$.

$k \leftarrow k + 1$.

Else

$v_{i,t+1} = v_{i,t}$.

$t \leftarrow t + 1$.

Figure D.3: Pseudo-code for the sparse ALMA algorithm.

D.6 Conclusion

In this appendix, we give implementation details for various linear-threshold algorithms that allow them to perform predictions and updates in $O(m_t)$, where m_t is the number of nonzero attributes in trial t. These linear-threshold algorithms are given in a form that allows them to learn arbitrary linear-threshold functions. The algorithms include Perceptron, Unnormalized Winnow, Normalized Winnow, Balanced Winnow, ALMA, and Tracking ALMA.

We also give implementation details to perform $O(m_t)$ predictions for any linear-threshold algorithm. This is beneficial for mistake-driven algorithms because they tend to make more predictions than updates. For example, Tracking Unnormalized Winnow can benefit from this technique even though we do not currently have a way to implement $O(m_t)$ updates for this algorithm.

Appendix E

Alternative proof for Tracking Unnormalized Winnow

In this appendix, we give a proof for an upper-bound on the number of mistakes made by Tracking Unnormalized Winnow that does not depend on ζ. Refer to Chapter 6 for more information on the Tracking Unnormalized Winnow and the terms used in this proof.

Our proof uses many of same lemmas as Chapter 6. One lemma we can not use is Lemma 6.6 because it bounds the maximum value of an algorithm weight in terms of ζ. Instead, we prove an alternative lemma that bounds the maximum weight in terms of the total number of promotions made by the algorithm.

Lemma E.1 *If the starting weight $\sigma \leq (\alpha - 1)/\alpha$ then the maximum value of any weight of the Tracking Unnormalized Winnow algorithm is $(\alpha - 1)|P|$, where P is the set of trials where a promotion occurs.*

Proof An algorithm weight can only increase on a promotion. Assume the first promotion occurs at trial t. On the first promotion, $w_{j,t+1} = \alpha^{x_{j,t}}\sigma$. Since $\max x_{j,t} \leq 1$, $w_{j,t+1} \leq \alpha\sigma \leq \alpha - 1$. Our goal is to show that the maximum increase for a weight on any of the remaining promotions is $\alpha - 1$. We break the proof into two cases.

Assume the value of weight $w_{j,t}$ is at most 1. Since $\max x_{j,t} \leq 1$, the new weight, after a promotion, is at most $\alpha w_{j,t}$. This gives a change in weight of

$$\alpha w_{j,t} - w_{j,t} = (\alpha - 1)w_{j,t} \leq (\alpha - 1).$$

Assume that the current value of weight w_i is greater than 1. For a promotion to occur, $\sum_{i=1}^{n} w_{i,t}x_{i,t} < 1$. Therefore $w_{j,t}x_{j,t} < \sum_{i=1}^{n} w_{i,t}x_{i,t} < 1$ implies that $x_{j,t} <$

$1/w_{j,t}$. This give a change in weight of at most

$$\alpha^{1/w_{j,t}} w_{j,t} - w_{j,t} = \left(e^{\frac{\ln \alpha}{w_{j,t}}} - 1 \right) w_{j,t}.$$

Using the fact that $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$ and the fact that $w_{j,t} > 1$,

$$\left(e^{\frac{\ln \alpha}{w_{j,t}}} - 1 \right) w_{j,t} = \ln \alpha + \frac{(\ln \alpha)^2}{w_{j,t} 2!} + \frac{(\ln \alpha)^3}{w_{j,t}^2 3!} + \cdots$$

$$\leq \ln \alpha + \frac{(\ln \alpha)^2}{2!} + \frac{(\ln \alpha)^3}{3!} + \cdots = e^{\ln \alpha} - 1 = \alpha - 1.$$

Given that the weight after the first promotion is at most $\alpha - 1$ and that the weight increases by at most $\alpha - 1$ on the remaining promotions, the maximum value of a weight is $(\alpha - 1)|P|$. ■

This bound on the maximum weight is problematic because it involves $|P|$, one of the mistake-bound components which we are trying to upper-bound. To help solve this problem we need another lemma.

Lemma E.2 *If $r < a \ln(r) + b$, $a \geq e^4$, and $b > 1$ then $r < a(\ln a)^2 + b \ln a$.*

Proof Let $f(x) = a \ln x + b$. If we graph $y = x$ and $y = f(x)$ for all $x \geq 1$ then the lines must cross at one point $s > 1$. For all $z \geq s$, $z \geq f(z)$. Therefore, if we find a point $t > 1$ such that $t \geq f(t)$ then $r < t$. Next we show that $t = a(\ln a)^2 + b \ln a$ satisfies $t \geq f(t)$. We break the proof into two cases.

First assume that $a(\ln a)^2 \geq b \ln a$.

$$f(t) = a \ln(a(\ln a)^2 + b \ln a) + b \leq a \ln(2a(\ln a)^2) + b = a \ln 2 + a \ln a + 2a \ln \ln a + b.$$

Using the definition of t,

$$t - f(t) \geq a(\ln a)^2 + b \ln a - a \ln 2 - a \ln a - 2a \ln \ln a - b.$$

Using that fact that $a \geq e^4$ and $b > 1$ the last formula is at least

$$a(\ln a)^2 - a \ln 2 - a \ln a - 2a \ln \ln a \geq 4a \ln a - a \ln 2 - a \ln a - 2a \ln \ln a$$

$$= (a \ln a - a \ln 2) + (a \ln a - a \ln a) + (2a \ln a - 2a \ln \ln a) > 0$$

Next assume $a(\ln a)^2 \leq b \ln a$. This implies $b \geq a \ln a$. Let $b = ak$ where $k \geq \ln a$.

$$f(t) = a \ln(a(\ln a)^2 + b \ln a) + b \leq a \ln(2b \ln a) + b = a \ln 2 + a \ln b + a \ln \ln a + b.$$

Using the definition of t and k,

$$t - f(t) \geq a(\ln a)^2 + ak \ln a - a \ln 2 - a \ln(ak) - a \ln \ln a - ak.$$

Using the fact that $a \geq e^4$, the last formula is

$$\geq 4a \ln a + 4ak - a \ln 2 - a \ln(ak) - a \ln \ln a - ak$$

$$\geq (a \ln a - a \ln 2) + (a \ln a - a \ln a) + (ak - a \ln k) + (a \ln a - a \ln \ln a) + (ak - ak) > 0.$$

■

Finally, we give an upper-bound on the number of mistakes made by Tracking Unnormalized Winnow. Notice that this mistake-bound does not depend on ζ and is only logarithmic in λ. Unlike Theorem 6.1, there is an extra condition that $\lambda \geq (1+\delta)/50$.

Theorem E.3 *For instances generated from a concept sequence C, if $\alpha = 1 + \delta$, $\epsilon = \sigma = \frac{\delta}{50\lambda}$ and $\lambda \geq (1+\delta)/50$ then the number of mistakes is less than*

$$\frac{2.05+\delta}{1+\delta}\left[\frac{2H(C)}{\delta^2(2-\delta)}\left(\ln\frac{2.05H(C)}{\delta^2(2+\delta-\delta^2)}\right)^2 + \left(\frac{2\ln(50\lambda)\,H(C)}{\delta^2(2-\delta)} + \frac{N}{\delta}\right)\ln\frac{2.05H(C)}{\delta^2(2+\delta-\delta^2)}\right].$$

Proof First we want to substitute Lemma 6.5 into Lemma 6.7. The lemma condition that $\alpha < e$ is satisfied since $\alpha = 1 + \delta \leq 2 < e$. The condition that $\epsilon < 1/\lambda$ is satisfied since $\epsilon\lambda = \delta/50 < 1$. Now we can proceed with the substitution. Using the fact that $\epsilon = \sigma$ and our bound on the maximum weight from Lemma E.1, we derive

$$H(C)\log_\alpha\left(\frac{(\alpha-1)|P|}{\epsilon}\right) + N > (1+\delta)|P| - (1-\delta)\frac{\alpha}{1-\epsilon\lambda}|P| \ .$$

Converting this to the form handled by Lemma E.2 gives

$$|P| \leq \frac{H(C)\frac{\ln|P|}{\ln\alpha}}{1+\delta-\frac{\alpha(1-\delta)}{1-\epsilon\lambda}} + \frac{H(C)\left(\frac{\ln(\alpha-1)+\ln(1/\epsilon)}{\ln\alpha}\right)+N}{1+\delta-\frac{\alpha(1-\delta)}{1-\epsilon\lambda}}$$

Substituting $\alpha = 1 + \delta$ and $\epsilon = \frac{\delta}{50\lambda}$ and using the fact that $\ln(1+\delta) \geq \delta - \delta^2/2$, we can compute the variables a and b used in Lemma E.2.

$$a = \frac{H(C)}{\left(1 + \delta - \frac{\alpha(1-\delta)}{1-\epsilon\lambda}\right)\ln\alpha} = \frac{(50-\delta)H(C)}{49\delta(1+\delta)\ln(1+\delta)} \leq \frac{(100-2\delta)H(C)}{49\delta^2(1+\delta)(2-\delta)}$$

$$b = \frac{H(C)\left(\frac{\ln(\alpha-1)+\ln(1/\epsilon)}{\ln\alpha}\right) + N}{1 + \delta - \frac{\alpha(1-\delta)}{1-\epsilon\lambda}} = \frac{50-\delta}{49\delta(1+\delta)}\left(\frac{H(C)\ln(50\lambda)}{\ln(1+\delta)} + N\right)$$
$$= \frac{(100-2\delta)\ln(50\lambda)\,H(C)}{49\delta^2(1+\delta)(2-\delta)} + \frac{50-\delta}{49\delta(1+\delta)}N.$$

Lemma E.2 shows us that

$$|P| < a(\ln a)^2 + b\ln a$$

Again using Lemma 6.5,

$$|P| + |D| < \left(1 + \frac{\alpha}{1-\epsilon\lambda}\right)|P| < \left(1 + \frac{\alpha}{1-\epsilon\lambda}\right)(a(\ln a)^2 + b\ln a).$$

The preceding is less than

$$\frac{2.05+\delta}{1+\delta}\left[\frac{2H(C)}{\delta^2(2-\delta)}\left(\ln\frac{2.05H(C)}{\delta^2(2+\delta-\delta^2)}\right)^2 + \left(\frac{2\ln(50\lambda)\,H(C)}{\delta^2(2-\delta)} + \frac{N}{\delta}\right)\ln\frac{2.05H(C)}{\delta^2(2+\delta-\delta^2)}\right].$$

■

The main result in this appendix is to give an upper-bound on the number of mistakes made by Tracking Unnormalized Winnow that does not depend on ζ. The proof is also interesting in that it gives some insight into how an adversary can increase an algorithm weight well beyond the value of the corresponding target weight.

Appendix F

General Results for Adversarial On-line Algorithms

In this appendix, we give some general results for adversarial on-line learning. We divide the results into two sections. The first sections deals with traditional on-line learning where the label for each instance is returned before the next trial. The second section deals primarily with delayed on-line learning where the label for an instance might be received during some future trial. See Chapter 8 for more information on delayed on-line learning.

We use some of the same notation as defined in Chapter 8. Let $\text{Mist}(B, s)$ be the number of mistakes algorithm B makes on s, a sequence of instances. Let $\text{Mist}(B)$ be the maximum number of mistakes made by algorithm B over a set of instance sequences. The particular set of sequences should be clear from context. When dealing with randomized algorithms, the previous notation refers to a random variable; therefore, we often refer to the expectation of the variable. Let Opt_D be the algorithm that minimizes $\text{Mist}(B)$ over all traditional deterministic algorithms B. Let Opt_R be the algorithm that minimizes $E[\text{Mist}(B)]$ over all traditional randomized algorithms B.

F.1 Traditional On-line Learning

We start with two transformations that can be applied to a traditional on-line algorithm B. The pseudo-code for algorithm B is found in Figure F.1. The prediction procedure of this algorithm is somewhat non-standard. The algorithm returns a probability vector, $\hat{y}_t$, over all the possible labels, Y. The algorithm makes a prediction on trial t by sampling from this distribution. This distribution encodes any randomization used by algorithm B.

Another possibility for randomization is to allow an algorithm to make randomized

Algorithm B
Initialization
$t \leftarrow 0$ is the trial number.
Initialize algorithm state $s \leftarrow s_0$.
Trials
$t \leftarrow t + 1$.
Instance: $\mathbf{x}_t$.
Prediction: $\hat{y}_t \leftarrow \text{Pred}(s, \mathbf{x}_t)$.
Update: Let y_t be the correct label.
$s \leftarrow \text{Update}(s, \mathbf{x}_t, y_t, \hat{y}_t)$.

Figure F.1: Pseudo-code for on-line algorithm B.

updates to its state. However, for every algorithm B that performs randomized updates to its state, there is another algorithm $\hat{B}$ that only uses randomization in prediction and makes the same expected number of mistakes on any sequence of instances. Algorithm $\hat{B}$ works by keeping track of a distribution over algorithm B states. This distribution is based on the randomized updates used by algorithm B. Algorithm $\hat{B}$ then predicts by sampling from this distribution. This forces algorithm $\hat{B}$ to have the same expected number of mistakes as algorithm B. Therefore, we only consider randomized predictions.

The first transformation converts a randomized on-line learning algorithm B to a deterministic learning algorithm *DR-B* with a similar mistake bound. Assume the prediction procedure of algorithm B returns a probability vector, $\hat{y}_t$ over the possible label outputs. Algorithm B predicts based on sampling this distribution. Algorithm *DR-B* modifies this by predicting the label with the highest probability. Algorithm *DR-B* makes at most twice the number of expected mistakes as algorithm B since the worst case corresponds to algorithm B predicting the wrong label with 1/2 probability on any given trial. This transformation and bound is from [AW95]. A popular use of this result is to show that the optimal deterministic algorithm makes at most two times the number of mistakes as the optimal randomized algorithm. In other words, randomization helps by at most a factor of two.

The second transformation is to convert an on-line algorithm to a mistake-driven algorithm. Traditionally, a mistake-driven algorithm is an algorithm that only updates

its state when it makes a mistake on an instance [Lit88]. In other words, the algorithm ignores any instance that it predicts correctly. We generalize this definition to handle randomized algorithms. A mistake-driven algorithm is an algorithm that only updates its state when it could have made a mistake on a instance. More formally, let $\hat{y}_t$ be a prediction probability distribution over the labels and let y_t be the correct label. A mistake-driven algorithm can only updates its state on trial t when $\hat{y}_t(y_t) < 1$.

The transformation is straightforward. Assume B is an on-line algorithm. The algorithm *MD-B* does not update on any instance when $\hat{y}_t(y_t) = 1$. These instances can not change the state of algorithm *MD-B*, so *MD-B* is a mistake-driven algorithm. Littlestone gives a similar transformation to convert algorithms to a mistake-driven form [Lit88, Lit95]. However, his transformation only applies to deterministic algorithms and his proof only handles fixed concepts without noise. Littlestone's transformation works by skipping any instances that are predicted correctly. This transformation was also used to convert Bayesian algorithms into algorithms that perform well against adversaries [Lit95, LM97].

Next, we prove that the *MD* transformation retains the existing mistake bound for a specific type of adversary called a subsequence adversary.

Definition F.1 *An adversary S is a subsequence adversary if, for every sequence $s \in S$, the adversary can generate every subsequence of s.*

Theorem F.2 *For a traditional on-line learning problem with instances generated by a subsequence adversary, $E[Mist(MD\text{-}B)] \leq E[Mist(B)]$.*

Proof Since the instances are generated by a subsequence adversary, there must exist a sequence of instances s that maximizes the expected number of mistakes for algorithm *MD-B* where all the trials that could cause a mistake occur at the beginning of the sequence. Let m be the first trial that algorithm *MD-B* must predict correctly. If trial m does not exist then set $m = \infty$. Up to trial m, both algorithm B and *MD-B* expect to make the same number of mistakes. Since *MD-B* makes no further mistakes past trial m, $\text{Mist}(MD\text{-}B) = Mist(MD\text{-}B, s) \leq Mist(B, s) \leq \text{Mist}(B)$. ■

Intuitively, this theorem says that if an algorithm would get any advantage from an instance that is predicted correctly then a subsequence adversary will never generate an instance that is predicted correctly. An adversary that is not a subsequence adversary may be forced to generate instances that are correctly classified. The information in these instances could be beneficial for the learning algorithm.

We want to stress that many practical on-line problems will not have an adversary generating the instances. In these cases, a more aggressive algorithm that sometimes updates on correct predictions can improve performance [LL00]. Still, the algorithm must be careful to avoid extra updates that increase the number of mistakes.

Next is a useful lemma that is similar to Theorem F.2 but is restricted to deterministic algorithms. This lemma shows that the adversary can create a sequence of mistakes at the start of learning.

Lemma F.3 *Assume B is a deterministic on-line algorithm. If the instances are generated by a subsequence adversary then there exists a sequence s such that $|s| = Mist(MD\text{-}B)$ and $Mist(B, s) = Mist(MD\text{-}B)$*

Proof Since the instance are generated by a subsequence adversary, and *MD-B* is deterministic and mistake-driven, we can take any sequence of instances that maximize the mistake bound of *MD-B* and remove any instances that are predicted correctly so that the start of the sequence is composed entirely of instances that cause mistakes. Let s be this portion of the sequence that has all the mistakes. Algorithm B must also make mistakes on all the instances in s. ■

This next lemma generalizes the previous lemma to handle randomized algorithms.

Lemma F.4 *Assume B is an on-line algorithm. If the instances are generated by a subsequence adversary then there exists a sequence s such that $|s| = Mist(Opt_D)$ and $E[Mist(B, s)] \geq Mist(Opt_D)/2$.*

Proof Convert algorithm B to *DR-B*. Since *DR-B* is deterministic by Lemma F.3 there

must exist a sequence of instances such that $|s| \geq \text{Mist}(Opt_D)$ and $\text{Mist}(DR\text{-}B, s) = |s|$. For every instance in s, the probability of a mistake by B must be at least $1/2$ based on how the derandomization transformation works. Therefore $E[\text{Mist}(B, s)] \geq |s|/2 \geq \text{Mist}(Opt_D)/2$. ■

F.2 Delayed On-line Learning

The previous results all dealt with the traditional on-line model. Next, we prove some related results with the delayed on-line model of Chapter 8. The main purpose of these results is to show that the optimal mistake-driven algorithm for traditional on-line learning may not give optimal results with the transformation of Chapter 8.

The results are based on a simple learning problem described in Section 8.2.3. Consider a learning problem that has a finite number l of instances and assume the learning problem allows all 2^l possible binary target functions. The adversary selects one of these target functions and can generate all l instances with the label determined by the selected target function. The only exception is that for up to A instances the adversary is allowed to return the opposite label. We call this learning problem $H_A(l)$.

We solve $H_A(l)$ with deterministic algorithm L_D. Algorithm L_D keeps a label count for each instance. Every time the algorithm receives an instance, the algorithm predicts according to the majority label. If there is a tie the algorithm predicts 1.

Theorem F.5 *Algorithm L_D is an optimal deterministic algorithm for problem $H_A(l)$ in the traditional on-line model and makes at most $l + 2A$ mistakes.*

Proof When $A = 0$, every deterministic algorithm can be forced to make at least one mistake on each instance. The first time an instance appears, the adversary just selects the opposite label of the algorithm. This is always possible since all 2^l target functions are allowed. Therefore every deterministic algorithm must make at least l mistakes. Based on our description, l is the maximum number of mistakes made by L_D on problem $H_0(l)$.

When $A > 0$ a result by Littlestone implies that any deterministic algorithm must make at least $l + 2A$ mistakes in the worst-case [Lit89]. Next we prove that the L_D algorithm makes at most $l + 2A$ mistakes and therefore is optimal.

For instance x, let a be the number of times the instance appears with the wrong label. In the worst-case, algorithm L_D can make a mistake on all a of these incorrectly labeled instances and another $a + 1$ correctly labeled instances. After $a + 1$ correctly labeled x instances, the algorithm can never make another mistakes on a correctly labeled instances since the correct label will always have a higher label count. Summing over all the instances, algorithm L_D makes at most $l + 2A$ mistakes. ■

Using the on-line-to-delayed transformation $OD2$ from Section 8.2.1, we can convert L_D into an algorithm that solves the delayed on-line learning problem.

Theorem F.6 *The algorithm OD2-L_D makes at most $kl + 2A$ mistakes on the delayed version of problem $H_A(l)$ when the maximum delay of any instance is k.*

Proof We start by considering a single instance x. This instance can appear multiple times in a sequence of instances. Let a be the number of times this instance appears with the incorrect label. After $a + 1$ correct labels appear for instance x, algorithm $OD2$-L_D can only predict the correct label.

For the rest of the proof, we need to make a distinction between when the attributes of an instance with a particular label arrives and when the label arrives. Assume that repeat r of instance x corresponds to the $a+1$ instance with the correct label. Therefore there must be $r - a - 1$ instances with the incorrect label in these first r repeats. The label for repeat r must be returned by trial $r + k$. This means that any mistakes on repeats greater than $r + k - 1$ can only be caused by an incorrect label. Since there are at most $a - r + a + 1$ incorrect labels left, only $2a - r + 1$ mistakes can be caused after repeat $r + k - 1$. Assuming a mistake on each of these first repeats gives at most $r + k - 1 + 2a - r + 1 = k + 2a$ mistakes.

This analysis can be reused for each instance. Therefore the total number of mistakes is at most $kl + 2A$. ■

Theorem F.6 is tight. We can prove this by showing a sequence of instances that generates $kl + 2A$ mistakes.

Theorem F.7 *If multi-set $\mathcal{D}$ contains at at least $2A$ elements with a delay of 1 and at least kl elements with a delay of k then there exists a delayed sequence of instances from problem $H_A(l)$ where algorithm OD2-L_D makes $kl + 2A$ mistakes.*

Proof Pick any instance and repeat the instance $2A$ times. Give each of these repeated instances a delay of 1 and alternate the label starting with label 0. This causes algorithm $OD2$-L_D to make $2A$ mistakes on these instances and uses all A label flips. At this point, the algorithm has an equal count of 0 and 1 labels for all instances.

Repeat each instance k times. Give each instance a delay of k and a label of 0. Algorithm $OD2$-L_D predicts 1 for all of these labels and therefore makes a mistake on all of these instances. This adds an additional kl mistakes. ■

Unfortunately, unlike in the traditional on-line model, the mistake-driven version of L_D does not do as well. Because of the nature of the mistake-driven algorithm, *MD-L_D* only keeps track of the last label it receives. At the start, the algorithm predicts 1. After that, it always predicts based on the last label.

Theorem F.8 *If multi-set $\mathcal{D}$ contains at least $2A + l$ elements with a delay of 1 and at least $A + l$ elements with a delay of 2 through k then there exists a delayed sequence of instances from problem $H_A(l)$ where algorithm OD2-MD-L_D makes $kl + (k + 1)A$ mistakes.*

Proof We start by considering a single instance x that is repeated. The correct label for this instance is 0. The first k repeats of instance x all receive their label at trial $k + 1$ and the label is 0 for all k instances. This causes a mistake on the first k trials. The $k + 1$ trial is updated on a single mistake from the first k trials. The remaining trials are ignored because after the first update, the mistake-driven algorithm makes the correct prediction.

Instance x is again repeated for trial $k+1$; a label of 1 is returned at the start of trial $k+2$. This causes a mistake as the algorithm is currently predicting 0 for the label. After the update, the algorithm is back to predicting 1. Notice that this is the same state the algorithm was in during trial 1. Therefore, we can use the same sequence of labels with more copies of instance x. This generates k mistakes for the first k trials and $k+1$ mistakes for each noisy instance. This gives a total of $k+(k+1)A$ mistakes caused by instance x.

For the other $l-1$ instances, we can always force k mistakes on each instance by repeating each instance k times with a label of 0 and a delay of k on each instance. This gives $kl+(k+1)A$ total mistakes. ■

Theorem F.8 shows that mistake-driven algorithms are not necessarily the best basic algorithms to use with the delayed on-line learning transformations given in Chapter 8. See Section 8.2 for more details.

F.3 Summary

In this appendix, we give some general results for adversarial on-line learning. We divide the results into two sections. The first sections deals with traditional on-line learning where the label for each instance is returned before the next trial. The second section deals primarily with delayed on-line learning where the label for an instance might be received during some future trial.

For traditional on-line learning, we give the details for two transformations. The first transformation converts a randomized algorithm B into a deterministic algorithm *DR-B*. The second transformation converts an arbitrary on-line algorithm B into a mistake-driven algorithm *MD-B*. We show that *MD-B* has the same upper-bound on mistakes as B for a wide range of problems. We also give related information on how an adversary can force an on-line algorithm to make a large number of mistakes during the initial trials of learning. This is useful for the lower-bound results in Chapter 8.

For delayed on-line learning, we given additional results on a learning problem that

was used as an example in Chapter 8. We give an algorithm, L_D, that is optimal for this problem in the traditional on-line setting and show that the $OD2$-L_D transformation is optimal for the delayed instance problem. Unfortunately, algorithm $OD2$-MD-L_D is not optimal showing that, unlike the traditional on-line setting, the mistake-driven transformation is not optimal for subsequence adversaries.

References

[AFJM95] Robert Armstrong, Dayne Freitag, Thorsten Joachims, and Tom Mitchell. WebWatcher: A learning apprentice for the World Wide Web. In *AAAI Spring Symposium on Information Gathering*, pages 6–12, 1995.

[AKC+00] I. Androutsopoulos, J. Koutsias, K. Chandrinos, G. Paliouras, and C. Spyropoulos. An evaluation of naive Bayesian anti-spam filtering. In *Proceedings of the Workshop on Machine Learning in the New Information Age*, pages 9–17, 2000.

[AW95] Peter Auer and Manfred K. Warmuth. Tracking the best disjunction. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 312–321. IEEE Computer Society Press, 1995.

[AW98] Peter Auer and Manfred K. Warmuth. Tracking the best disjunction. *Machine Learning*, 32(2):127–150, 1998.

[BB00] Avrim Blum and Carl Burch. On-line learning and the metrical task system problem. *Machine Learning*, 39(1):35–58, 2000.

[BB01] Michele Banko and Eric Brill. Scaling to very very large corpora for natural language disambiguation. In *Meeting of the Association for Computational Linguistics*, pages 26–33, 2001.

[BHL91] Avrim Blum, Lisa Hellerstein, and Nick Littlestone. Learning in the presence of finitely or infinitely many irrelevant attributes. In *Proceedings of the Third Annual Conference on Computational Learning Theory*, pages 157–166, 1991.

[BKV03] N. Bel, C. Koster, and M. Villegas. Cross-lingual text categorization. In *7th European Conference on Research and Advanced Technology for Digital Libraries*, pages 126–139, 2003.

[Blu95] Avrim Blum. Empirical support for winnow and weighted-majority algorithms: Results on a calendar scheduling domain. In *Proceeding of the Twelfth International Conference on Machine Learning*, pages 64–72, 1995.

[BM98] Avrim Blum and Tom Mitchell. Combining labeled and unlabeled data with co-training. In *Proceeding of the Eighth Annual Conference on Computational Learning Theory*, pages 92–100, 1998.

[Bre96] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[Bre98] Leo Breiman. Arcing classifiers. *The Annals of Statistics*, 26(3):801–849, 1998.

[CBFH+97] Nicolò Cesa-Bianchi, Yoav Freund, David Haussler, David P. Helmbold, Robert E. Schapire, and Manfred K. Warmuth. How to use expert advice. *Journal of the Association for Computing Machinery*, 44(3):427–485, 1997.

[CDF+ta] Mark Craven, Dan DiPasquo, Dayne Feitag, Andrew Mccallum, Tom Mitchel, Kamal Nigam, and Sean Slattery. The 4 universities data set, 1998; http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-20/www/data/.

[CS96] William W. Cohen and Yoram Singer. Context-sensitive learning methods for text categorization. In Hans-Peter Frei, Donna Harman, Peter Schäuble, and Ross Wilkinson, editors, *Proceedings of SIGIR-96, 19th ACM International Conference on Research and Development in Information Retrieval*, pages 307–315, Zürich, CH, 1996. ACM Press, New York, US.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[Dav01] Brian D. Davison, 2001. personal communication.

[DeG86] Morris H. DeGroot. *Probability and Statistics*. Addison-Wesley, Menlo Park, California, 1986.

[DGL91] L. Devroye, L. Györfi, and G. Lugosi. *A Probabilistic Theory of Pattern Recognition*. Springer, New York, 1991.

[DH73] Richard O. Duda and Peter E. Hart. *Pattern Classification and Scene Analysis*. Wiley, New York, 1973.

[DKR97] I. Dagan, Y. Karov, and D. Roth. Mistake-driven learning in text categorization. In *Proceedings of the 2nd Conference on Empirical Methods in Natural Language Processing*, pages 55–63, 1997.

[DNMml] C.L. Blake D.J. Newman, S. Hettich and C.J. Merz. UCI repository of machine learning databases, 1998; http://www.ics.uci.edu/∼mlearn/MLRepository.html.

[DP97] 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.

[Elk01] Charles Elkan. The foundations of cost-sensitive learning. In *Proceeding of 17th International Joint Conference on Artificial Intelligence*, pages 973–978, 2001.

[FG03] Alan Fern and Robert Givan. Online ensemble learning: An empirical study. *Machine Learning*, 53(1-2):71–109, 2003.

[Fox90] Christopher Fox. A stop list for general text. *SIGIR Forum*, 24(1-2):19–21, 1990.

[FS96] Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm. In *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 148–156, 1996.

[FS98] Yoav Freund and Robert E. Schapire. Large margin classification using the perceptron algorithm. In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, 1998.

[Gal90] Stephen I. Gallant. Perceptron-based learning algorithms. *Neural Networks, IEEE Transactions on*, 1(2):179–191, 1990.

[Gen01] Claudio Gentile. A new approximate maximal margin classification algorithm. *Machine Learning*, 2:213–242, 2001.

[Gen03] Claudio Gentile. The robustness of the p-norm algorithms. *Machine Learning*, 53(3):265–299, 2003.

[GLS01] Adam J. Grove, Nick Littlestone, and Dale Schuurmans. General convergence results for linear discriminant updates. *Machine Learning*, 43(2):173–210, 2001.

[GR96] Andrew R. Golding and Dan Roth. Applying Winnow to context-sensitive spelling correction. In *Proceeding of the Thirteenth International Conference on Machine Learning*, 1996.

[GR99] Andrew R. Golding and Dan Roth. A Winnow-based approach to context-sensitive spelling correction. *Machine Learning*, 34:107–130, 1999.

[GW99] Claudio Gentile and Manfred K. Warmuth. Linear hinge loss and average margin. In *Advances in Neural Information Processing Systems 11*, pages 225–231. MIT Press, 1999.

[HBdu] S. Hettich and S. D. Bay. The UCI KDD archive, 1999; http://kdd.ics.uci.edu/.

[Her01] Mark Herbster. Learning additive models online with fast evaluating kernels. In *Proceedings of the Fourteenth Annual Conference on Computational Learning Theory*, pages 444–460. Springer, 2001.

[HL91] David P. Helmbold and Philip M. Long. Tracking drifting concepts using random examples. In *Proceedings of the Third Annual Conference on Computational Learning Theory*, pages 13–23, 1991.

[HLL00a] David P. Helmbold, Nick Littlestone, and Peter M. Long. Apple tasting. *Information and Computation*, 161(2):85–139, 2000.

[HLL00b] David P. Helmbold, Nick Littlestone, and Peter M. Long. Online learning with linear loss constraints. *Information and Computation*, 161(2):140–171, 2000.

[HLSS00] David P. Helmbold, Darrel D. Long, Tracey L. Sconyers, and Bruce Sherrod. Adaptive disk spin-down for mobile computers. *Mobile Networks and Applications*, 5(4):285–297, 2000.

[HMM+03] Tim Hesterberg, Shaun Monaghan, David S. Moore, Ashley Clipson, and Rachel Epstein. *The Practice of Business Statistics*. W. H. Freeman and Company, New York, 2003.

[HPW99] David P. Helmbold, Sandra Panizza, and Manfred K. Warmuth. Direct and indirect algorithms for on-line learning of disjunctions. *Lecture Notes in Computer Science*, 1572:138–152, 1999.

[HW98] Mark Herbster and Manfred K. Warmuth. Tracking the best expert. *Machine Learning*, 32(2):151–178, 1998.

[HW01] Mark Herbster and Manfred K. Warmuth. Tracking the best linear predictor. *Machine Learning*, 1:281–309, 2001.

[JG03] Norbert Jankowski and Krzysztof Grabczewski. Toward optimal svm. In *The Third Internation Conference on Artifical Intelligence and Applications*, pages 451–456, 2003.

[Joa98] Thorsten Joachims. Making large-scale support vector machine learning practical. In A. Smola B. Schölkopf, C. Burges, editor, *Advances in Kernel Methods: Support Vector Machines*. MIT Press, Cambridge, MA, 1998.

[KMB03] Cornelis H.A. Koster, M.Seutter, and Jean G. Beney. Multi-classification of patent applications with winnow. In *Perspectives of System Informatics: 5th International Andrei Ershov Memorial Conference*, pages 545–554, 2003.

[KPR91] Anthony Kuh, Thomas Petsche, and Ronald L. Rivest. Learning time-varying concepts. In *Advances in Neural Information Processing Systems 3*, pages 183–189. Morgan Kaufmann Publishers, Inc., 1991.

[KR98] Yuval Krymolowski and Dan Roth. Incorporating knowledge in natural language learning: A case study. In Sanda Harabagiu, editor, *Use of WordNet in Natural Language Processing Systems*, pages 121–127. Association for Computational Linguistics, Somerset, New Jersey, 1998.

[KS95] Norbert Klasner and Hans-Ulrich Simon. From noise-free to noise-tolerant and from on-line to batch learning. In *Computational Learning Theory*, pages 250–257, 1995.

[KSW02] Jyrki Kivinen, Alex J. Smola, and Robert C. Williamson. Large margin classification for moving targets. In *Proceedings of the 13th Annual International Conference on Algorithmic Learning Theory*, pages 113–127. Springer, 2002.

[KW97] Jyrki Kivinen and Manfred K. Warmuth. Additive versus exponentiated gradient updates for linear prediction. *Information and Computation*, 132(1):1–64, 1997.

[Lew92] David D. Lewis. An evaluation of phrasal and clustered representations on a text categorization task. In *SIGIR '92: Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in*

Information Retrieval, pages 37–50, New York, NY, USA, 1992. ACM Press.

[LH02] Gabriel H. Loh and Dana S. Henry. Applying machine learning for ensemble branch predictors. In *Proceedings of the 15th international conference on Industrial and engineering applications of artificial intelligence and expert systems: developments in applied artificial intelligence*, pages 264–274, London, UK, 2002. Springer-Verlag.

[Lit88] Nick Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2:285–318, 1988.

[Lit89] Nick Littlestone. *Mistake Bounds and Linear-threshold Learning Algorithms*. PhD thesis, Computer Science, University of California, Santa Cruz, 1989. Technical Report UCSC-CRL-89-11.

[Lit91] Nick Littlestone. Redundant noisy attributes, attribute errors, and linear-threshold learning using winnow. In *Proceedings of the Third Annual Conference on Computational Learning Theory*, pages 147–156, 1991.

[Lit94] Nick Littlestone, 1994. Unpublished manuscript that generalizes the WMA algorithm.

[Lit95] Nick Littlestone. Comparing several linear-threshold learning algorithms on tasks involving superfluous attributes. In *Proceeding of the Twelfth International Conference on Machine Learning*, pages 353–361, 1995.

[Lit96] Nick Littlestone, 1996. personal communication.

[Lit97] Nick Littlestone, 1997. personal communication.

[LL00] Yi Li and Philip Long. The relaxed online maximum margin algorithm. In *Advances in Neural Information Processing Systems 12*, pages 498–504. MIT Press, 2000.

[LM97] Nick Littlestone and Chris Mesterharm. An apobayesian relative of winnow. In *Advances in Neural Information Processing Systems 9*, pages 204–210. MIT Press, 1997.

[LSCP96] David D. Lewis, Robert E. Schapire, James P. Callan, and Ron Papka. Training algorithms for linear text classifiers. In Hans-Peter Frei, Donna Harman, Peter Schäuble, and Ross Wilkinson, editors, *Proceedings of SIGIR-96, 19th ACM International Conference on Research and Development in Information Retrieval*, pages 298–306, Zürich, CH, 1996. ACM Press, New York, US.

[LW94] Nick Littlestone and Manfred K. Warmuth. The weighted majority algorithm. *Information and Computation*, 108:212–261, 1994.

[McCow] Andrew Kachites McCallum. Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering, 1996; http://www.cs.cmu.edu/ mccallum/bow.

[Mes00] Chris Mesterharm. A multi-class linear learning algorithm related to winnow. In *Advances in Neural Information Processing Systems 12*, pages 519–525. MIT Press, 2000.

[Mes01] Chris Mesterharm. Transforming linear-threshold learning algorithms into multi-class linear learning algorithms. Technical Report dcs-tr-460, Rutgers University, 2001.

[Mes02] Chris Mesterharm. Tracking linear-threshold concepts with winnow. In *Proceedings of the 15th Annual Conference on Computational Learning Theory*, pages 138–152. Springer, 2002.

[Mes03] Chris Mesterharm. Tracking linear-threshold concepts with winnow. *Journal of Machine Learning Research*, 4:819–838, 2003.

[Mes05] Chris Mesterharm. On-line learning with delayed label feedback. In *Proceedings of the 16th Annual International Conference on Algorithmic Learning Theory*, pages 399–413, 2005.

[MGH+05] Michael B. Monagan, Keith O. Geddes, K. Michael Heal, George Labahn, Stefan M. Vorkoetter, James McCarron, and Paul DeMarco. *Maple 10 Programming Guide*. Maplesoft, Waterloo ON, Canada, 2005.

[MHBD01] Sofus A. Macskassy, Haym Hirsh, Arunava Banerjee, and Aynur A. Dayanik. Using text classifiers for numerical classification. In Bernhard Nebel, editor, *Proceeding of 17th International Joint Conference on Artificial Intelligence*, pages 885–890, Seattle, US, 2001.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, Boston, MA, 1997.

[MKCN98] Surapant Meknavin, Boonserm Kijsirikul, Ananlada Chotimongkol, and Cholwich Nuttee. Combining trigram and winnow in thai OCR error correction. In *Proceedings of the 36th annual meeting on Association for Computational Linguistics*, volume 2, pages 836–842, 1998.

[OR01] Nikunj C. Oza and Stuart J. Russell. Experimental comparisons of online and batch versions of bagging and boosting. In *Knowledge Discovery and Data Mining*, pages 359–364, 2001.

[PM96] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve world wide web latency. *Computer Communication Review*, 3:22–36, 1996.

[Ros62] Frank Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, Washington, DC, 1962.

[RtY01] Dan Roth and Wen tau Yih. Relational learning via propositional algorithms: An information extraction case study. In *Proceeding of 17th International Joint Conference on Artificial Intelligence*, pages 1257–1263, 2001.

[RZ98] Dan Roth and Dmitry Zelenko. Part of speech tagging using a network of linear separators. In *Proceedings of the 36th annual meeting on Association for Computational Linguistics*, pages 1136–1142, 1998.

[Sid02] Advaith Siddharthan. Resolving relative clause attachment ambiguities using machine learning techniques and wordnet hierarchies. In *Proceedings of the 5th National Colloquium for Computational Linguistics in the UK*, pages 45–49, 2002.

[TCS03] V. Tesprasit, P. Charoenpornsawat, and V. Sornlertlamvanich. A context-sensitive homograph disambiguation in thai text-to-speech synthesis. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology*, pages 103–105, 2003.

[ZDJ01] Tong Zhang, Fred Damerau, and David Johnson. Text chunking using regularized winnow. In *Meeting of the Association for Computational Linguistics*, pages 539–546, 2001.

[ZLA03] Bianca Zadrozny, John Langford, and Naoki Abe. A simple method for cost-sensitive learning. Technical Report RC22666, IBM, 2003.

Vita

Jon Christian Mesterharm

1988-1992	Attended Virginia Tech, Blacksburg, Virginia.
1992	B.A. in Computer Engineering, Virginia Tech.
1992-1999	Attended Rutgers University, New Brunswick, New Jersey.
1999	M.S. in Computer Science, Rutgers University.
2000-2007	Attended Rutgers University, New Brunswick, New Jersey.
2007	Ph.D. in Computer Science, Rutgers University.

1993-1995	Teaching Assistant, Rutgers Department of Computer Science.
1995-1999	Research Assistant, NEC Institute, Princeton, New Jersey.
2000-2006	Teaching Assistant, Rutgers Department of Computer Science.

1997	Nick Littlestone and Chris Mesterharm. An Apobayesian Relative of Winnow. In Neural Information Processing Systems 9, pages 204-210, 1997.
2000	Chris Mesterharm. A Multi-class Linear Learning Algorithm. In Neural Information Processing Systems 12, pages 519-525, 2000.
2002	Chris Mesterharm, Tracking Linear-threshold Concepts with Winnow, In Proceedings of the 15th Annual Conference on Computational Learning Theory, pages 138-152, 2002.
2003	Chris Mesterharm. Using Linear-threshold Algorithms to Combine Multi-class Sub-experts. In Proceeding of the 20th International Conference on Machine Learning, pages 544-551, 2003.
2003	Chris Mesterharm. Tracking Linear-threshold Concepts with Winnow. In Journal of Machine Learning Research 4, pages 819-838, 2003.
2005	Chris Mesterharm. On-line Learning with Delayed Label Feedback. In Proceedings of the 16th Annual International Conference on Algorithmic Learning Theory, pages 399-413, 2005.

2006 Alexander Strehl, Chris Mesterharm, Michael Littman, and Haym Hirsh. Experience-Efficient Learning in Associative Bandit Problems. In Proceedings of the 23rd International Conference on Machine Learning, pages 889-896, 2006.