Online Ensemble Learning

by

Nikunj Chandrakant Oza

B.S. (Massachusetts Institute of Technology) 1994
M.S. (University of California, Berkeley) 1998

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA, BERKELEY

Committee in charge:

Professor Stuart Russell, Chair
Professor Michael Jordan
Professor Leo Breiman

Fall 2001

The dissertation of Nikunj Chandrakant Oza is approved:

Chair Date

Date

Date

University of California at Berkeley

2001

Online Ensemble Learning

by
Nikunj Chandrakant Oza

Abstract

Online Ensemble Learning

by

Nikunj Chandrakant Oza

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Stuart Russell, Chair

This thesis presents online versions of the popular bagging and boosting algorithms. We demonstrate theoretically and experimentally that the online versions perform comparably to their original batch counterparts in terms of classification performance. However, our online algorithms yield the typical practical benefits of online learning algorithms when the amount of training data available is large.

Ensemble learning algorithms have become extremely popular over the last several years because these algorithms, which generate multiple *base* models using traditional machine learning algorithms and combine them into an *ensemble* model, have often demonstrated significantly better performance than single models. Bagging and boosting are two of the most popular algorithms because of their good empirical results and theoretical support. However, most ensemble algorithms operate in batch mode, i.e., they repeatedly read and process the entire training set. Typically, they require at least one pass through the training set for every base model to be included in the ensemble. The base model learning algorithms themselves may require several passes through the training set to create each base model. In situations where data is being generated continuously, storing data for batch learning is impractical, which makes using these ensemble learning algorithms impossible. These algorithms are also impractical in situations where the training set is large enough that reading and processing it many times would be prohibitively expensive.

This thesis describes online versions of bagging and boosting. Unlike the batch versions, our online versions require only one pass through the training examples in order regardless of the number of base models to be combined. We discuss how we derive the online algorithms from their batch counterparts as well as theoretical and experimental evidence that our online algorithms perform comparably to the batch versions in terms of classification performance. We also demonstrate that our online algorithms have the practical advantage of lower running time, especially for larger datasets. This makes our online algorithms practical for machine learning and data mining tasks where the amount of training data available is very large.

Professor Stuart Russell
Dissertation Committee Chair

To my parents.

Contents

List of Figures

List of Tables

Acknowledgements

This Ph.D. thesis represents a great deal more than the intellectual exercise that it was intended to represent. This thesis, and my Ph.D. program as a whole reflects my growth both intellectually and as a person. Many people have played significant roles in my growth during this period and earlier. First and foremost I give thanks to my parents. I think my former chess coach, Richard Shorman, said it best, "They clearly put a lot of tender love and care into you." I could not agree more. They have always been there for me and they are much more than I could ask for in a family. My uncle "Uttammama" (Uttam Oza) has been a part of my life since I was one year old. From patiently allowing me to read storybooks to him when I was two years old to making many suggestions related to my applications to undergraduate and graduate schools, he has always been there and it is impossible to enumerate everything he has done. During the time I have known him, he has added to his family. He and my aunt Kalindimami, and cousins Anand and Anuja have formed a second immediate family that have always been there. My extended families on my mom and dad's sides both here and in India have always expressed great confidence and pride in me, which has helped me immensely in my work and in life in general.

I made many friends while I was here. First and foremost, I thank Huma Dar. I tell people that she is my former classmate, former officemate, friend, political discussion-mate, Berkeley older sister, and parole officer. I have known Mehul Patel since my undergraduate days at MIT and he continues to be a good friend here at Berkeley and a link to the past. Marco van Akkeren and I spent much time discussing graduate student life. Marco, Megumi Yamato, and I had many discussions and I especially enjoyed our Sunday evening jaunts to Ben and Jerry's. Michael Anshelevich was also part of our group at International House and he helped me with one of the proofs in this thesis.

Several current and past members of RUGS (Russell's Unusual Graduate Students) have been helpful to me. Geoff Zweig, Jeff Forbes, Tim Huang, Ron Parr, Archie Russell, and John Binder were quite helpful to me when I was doing my Master's research. Dean Grannes, Eric Fosler, and Phil McLauchlan have helped me in many ways and I appreciate the chance that I had to work with them. My officemates over the years have helped me with my work and added much-needed humor to the office environment: David Blei, Huma

Dar, Daishi Harada, Andrew Ng, Magnus Stensmo, and Andy Zimdars. I bugged Andrew Ng and David Blei often to pick the probability components of their brains. Andrew Ng was especially helpful to me with one of the proofs in this thesis. Thanks guys!

Stuart Russell was my research advisor and always will be an advisor to me. He gave me the freedom to explore the field to decide what I wanted to work on. In my explorations I learned much about the field beyond my thesis topic, and I know that this will help me throughout my career. He patiently helped me through my explorations and took much time to discuss ideas with me. I credit most of my intellectual growth during my time at Berkeley to him. The funding that he gave me allowed me to work without worrying about the financial side of life. Mike Jordan, Leo Breiman, and Joe Hellerstein served on my qualifying exam committee and were generous with their time in helping me sort out exactly what I wanted to work on. Jitendra Malik has been helpful to me during my time here in many ways including serving on my Master's report committee.

During my time here, California MICRO and Schlumberger provided fellowship support for two years and the NSF Graduate Research Training Program in Spatial Cognition provided me with funding for two years. The NSF program helped me broaden my horizons by enabling me to learn about and conduct research in Cognitive Science and Psychology. A number of administrators were a great help to me while I was here. Sheila Humphreys, Mary Byrnes, Winnie Wang, Peggy Lau, and Kathryn Crabtree helped smooth out many bumps during my time at Berkeley.

I taught a C++ programming course in the Spring 1998 semester at City College of San Francisco. I thoroughly enjoyed the chance to run my own course. I became friends with Louie Giambattista while I was there and he remains a good friend to this day. As a graduate student I worked during one summer at General Electric Corporate Research and Development with Tomek Strzalkowski. Working with him was a great pleasure and he has helped me after that summer in many ways. I also spent two summers at NASA Ames Research Center with Kagan Tumer. He gave me much freedom to explore what I wanted to work on while I was there and took me under his wing as a research "little brother."

Ron Genise taught me algebra and geometry when I was in junior high school. He also designed a course in which we used a geometry exploration software that he and his son designed to explore concepts of geometry. He gave me my first taste of what research was

like, and I was hooked. His dedication to teaching inspired me and he has served as my mentor ever since I took his courses. Roger Poehlmann is my chess coach. His dedication to chess and his enjoyment of teaching has not only helped me improve as a tournament chess player but has helped me in my work by providing me a needed diversion. I look forward to continuing to work with him. Prof. Michael Rappa was my freshman advisor at MIT and both he and his wife Lynda have been friends ever since. They have helped me in more ways than I can count and I feel honored to know two such special people.

Chapter 1

Introduction

A supervised learning task involves constructing a mapping from input data (normally described by several features) to the appropriate outputs. In a classification learning task, each output is one or more classes to which the input belongs. In a regression learning task, each output is some other type of value to be predicted. In supervised learning, a set of training examples—examples with known output values—is used by a learning algorithm to generate a model. This model is intended to approximate the mapping between the inputs and outputs. For example, we may have data consisting of examples of credit card holders. In a classification learning task, our goal may be to learn to predict whether a person will default on his/her credit card based on the other information supplied. Each example may correspond to one credit card holder, listing that person's various features of interest (e.g., average daily credit card balance, other credit cards held, and frequency of late payment) as well as whether the person defaulted on his/her credit card. A learning algorithm would use the supplied examples to generate a model that approximates the mapping between each person's supplied financial information and whether or not he defaults on his credit card. This model can then be used to predict whether an applicant for a new credit card will default and; therefore, to decide whether to issue that person a card. A regression learning task's goal may be to predict a person's average daily credit card balance based on the other variables. In this case, a learning algorithm would generate a model that approximates the mapping from each person's financial information to the average daily balance. The generalization performance of a learned model (how closely the target outputs and the

model's predicted outputs agree for patterns that have not been presented to the learning algorithm) would provide an indication of how well the model has learned the desired mapping.

Many learning algorithms generate one model (e.g., a decision tree or neural network) that can be used to make predictions for new examples. *Ensembles*—also known as combiners, committees, or turnkey methods—are combinations of several models whose individual predictions are combined in some manner (e.g., averaging or voting) to form a final prediction. For example, one can generate three neural networks from a supplied training set and combine them into an ensemble. The ensemble could classify a new example by passing it to each network, getting each network's prediction for that example, and returning the class that gets the maximum number of votes. Many researchers have demonstrated that ensembles often outperform their *base* models (the component models of the ensemble) if the base models perform well on novel examples and tend to make errors on different examples (e.g., (Breiman, 1993; Oza & Tumer, 1999; Tumer & Oza, 1999; Wolpert, 1992)). To see why, let us define h_1, h_2, h_3 to be the three neural networks in the previous example and consider a new example x. If all three networks always agree, then whenever $h_1(x)$ is incorrect, $h_2(x)$ and $h_3(x)$ will also be incorrect, so that the incorrect class will get the majority of the votes and the ensemble will also be incorrect. On the other hand, if the networks tend to make errors on different examples, then when $h_1(x)$ is incorrect, $h_2(x)$ and $h_3(x)$ may be correct, so that the ensemble will return the correct class by majority vote. More precisely, if an ensemble has M base models having an error rate $\epsilon < 1/2$ and if the base models' errors are independent, then the probability that the ensemble makes an error is the probability that more than $M/2$ base models misclassify the example. This is precisely $P(B > M/2)$, where B is a $Binomial(M, \epsilon)$ random variable. In our three-network example, if all the networks have an error rate of 0.3 and make independent errors, then the probability that the ensemble misclassifies a new example is 0.21. Even better than base models that make independent errors would be base models that are somewhat anti-correlated. For example, if no two networks make a mistake on the same example, then the ensemble's performance will be perfect because if one network misclassifies an example, then the remaining two networks will correct the error.

Two of the most popular ensemble algorithms are bagging (Breiman, 1994) and boost-

ing (Freund & Schapire, 1996). Given a training set, bagging generates multiple bootstrapped training sets and calls the base model learning algorithm with each of them to yield a set of base models. Given a training set of size N, bootstrapping generates a new training set by repeatedly (N times) selecting one of the N examples at random, where all of them have equal probability of being selected. Some training examples may not be selected at all and others may be selected multiple times. A bagged ensemble classifies a new example by having each of its base models classify the example and returning the class that receives the maximum number of votes. The hope is that the base models generated from the different bootstrapped training sets disagree often enough that the ensemble performs better than the base models. Boosting is a more complicated algorithm that generates a sequence of base models. Boosting maintains a probability distribution over the training set. Each base model is generated by calling the base model learning algorithm with the training set weighted by the current probability distribution.[1] Then, the base model is tested on the training set. Those examples that are misclassified have their weights increased so that their weights represent half the total weight of the training set. The remaining half of the total training set weight is allocated for the correctly classified examples, which have their weights decreased accordingly. This new probability distribution and the training set are used to generate the next base model. Intuitively, boosting increases the weights of previously misclassified examples, thereby focusing more of the base model learning algorithm's attention on these hard-to-learn examples. The hope is that subsequent base models correct the mistakes of the previous models. Bagging and boosting are popular among ensemble methods because of their strong theoretical motivations (Breiman, 1994; Freund & Schapire, 1997) and the good experimental results obtained with them (Freund & Schapire, 1996; Bauer & Kohavi, 1999; Dietterich, 2000).

Most ensemble learning algorithms including bagging and boosting are *batch* algorithms. That is, they repeatedly read and process the entire set of training examples. They typically require at least one pass through the data for each base model to be generated. Often, the base model learning algorithms themselves require multiple passes through the training set to create each model. Suppose M is the number of base models to be incorpo-

[1] If the base model learning algorithm cannot accept weighted training sets, then one can generate a bootstrapped training set according to the weight distribution and pass that to the learning algorithm instead.

rated into the ensemble and T is the average number of times that the base model learning algorithm needs to pass through the training set to create a base model. In this case, bagging requires MT passes through the training set. Boosting requires $M(T+1)$ passes through the training set—for each base model, it needs T passes to create the base model and one pass to test the base model on the training set. We would prefer to learn the entire ensemble in an *online* fashion, i.e., using only one pass through the entire training set. This would make ensemble methods practical for use when data is being generated continuously so that storing data for batch learning is impractical. Online ensemble learning would also be helpful in data mining tasks in which the datasets are large enough that multiple passes through the datasets would require a prohibitively long training time.

In this thesis, we present online versions of bagging and boosting. Specifically, we discuss how our online algorithms mirror the techniques that bagging and boosting use to generate multiple distinct base models. We also present theoretical and experimental evidence that our online algorithms succeed in this mirroring, often obtaining classification performance comparable to their batch counterparts in less time. Our online algorithms are demonstrated to be more practical with larger datasets.

In Chapter 2, we present the relevant background for this thesis. We first review batch supervised learning in general and the base models that we use. We discuss what has been done in the areas of ensemble learning and online learning, which are the two essential components of this thesis. We also describe the little work that has been done in online ensemble learning. We discuss in more detail the motivation for ensemble learning and the various ensemble learning algorithms. We compare these algorithms in terms of how they bring about diversity among their constituent or base models, which is necessary in order for ensembles to achieve better performance. We also discuss online learning, including the motivation for it and the various online learning algorithms. We discuss the Weighted Majority (Littlestone & Warmuth, 1994) and Winnow (Littlestone, 1988) algorithms in more detail because, like online ensemble learning algorithms, they maintain several models and update them in an online manner. However, Weighted Majority and Winnow are designed to perform not much worse than the best of their constituent models, whereas our online ensemble algorithms, like typical batch ensemble methods, are designed to yield performance better than any of their base models. We discuss how this difference comes about.

In Chapter 3, we discuss the bagging algorithm in great detail. We then derive our online bagging algorithm which has two requirements. The first requirement is an online learning algorithm to produce the base models. The second requirement is a method of mirroring bagging's method of producing multiple distinct base models. In particular, bagging's bootstrap sampling method, which we discussed earlier, requires knowing N, which is often unavailable when learning online. Our online bagging algorithm avoids this requirement. It simulates the bootstrap sampling process by sending K copies of each new training example to update each base model, where K is a suitable Poisson random variable. We prove that the sampling distribution used in online bagging converges to what is used in batch bagging. We then prove that the ensemble returned by our online bagging algorithm converges to that returned by the batch algorithm given the same training set if the base model learning algorithms are proportional and if they return classifiers that converge toward each other as the size of the training set grows. By proportional, we mean that the batch and online base model learning algorithms return the same hypothesis given training sets where the relative proportion of every example to every other example is the same. The size of the bootstrap training set should not matter. For example, changing the training set by duplicating every example should not change the hypothesis returned by the base model learning algorithm. This is true for decision trees, decision stumps, and Naive Bayes classifiers, which are three of the base models that we experiment with in this thesis. However, this is not true for neural networks, which we also experiment with.

We discuss the results of our experiments comparing the generalization performances of online bagging and batch bagging on many real and synthetic datasets of varying sizes. We see that batch and online bagging mostly performed comparably to one another when using decision trees, decision stumps, and Naive Bayes classifiers as the base models. These base models satisfy the conditions we described in the last paragraph. When using neural networks as the base models, online bagging's performance suffered, especially on the smaller datasets. On larger datasets; however, the loss was small enough that it may be acceptable given the reduced running time. Additionally, we discuss experiments with our online bagging algorithm on a domain in which data is arriving continuously and the algorithm needs to supply a prediction for each example as it arrives. Online bagging with decision trees performed best on this problem and improved upon single decision trees.

In Chapter 4, we discuss the boosting algorithm (specifically, AdaBoost) in detail and derive our online boosting algorithm from boosting. Just as with bagging, it appears that we require foreknowledge of N, the size of the training set, in order to give the training examples their proper weights. Once again, we avoid the requirement of knowing N using suitable Poisson random variables to approximate boosting's process of reweighting the training set. We prove that our online boosting algorithm's performance becomes comparable to that of the batch boosting algorithm when using Naive Bayes base models. We also review our experimental results comparing the performances of online boosting and batch boosting on the same datasets on which we compare online and batch bagging. We find that, in some cases, our online boosting algorithm significantly underperforms batch boosting with small training sets. Even when the training set is large, our online algorithm may underperform initially before finally catching up to the batch algorithm. This characteristic is common with online algorithms because they do not have the luxury of viewing the training set as a whole the way batch algorithms do. Online boosting's performance is especially worse when given a lossy online learning algorithm to create its base models. We experiment with "priming" our online boosting algorithm by running it in batch mode for some initial subset of the training set and running in online mode for the remainder of the training set. Priming is demonstrated to improve online boosting, especially when the online base model learning algorithm is lossy. We also compare the running times of our algorithms. In most cases, online boosting was faster than batch boosting while in other cases online boosting was slower. However, primed online boosting was almost always fastest among the boosting algorithms. We also compare online boosting and batch boosting in more detail by comparing the errors of their base models, which are important in determining the overall behavior of the boosting algorithms. Additionally, we discuss experiments with our online boosting algorithm on the same online domain that we experiment with in Chapter 3.

In Chapter 5, we summarize the contributions of this thesis and discuss future work. This thesis delivers online versions of bagging and boosting that allow one to obtain the high accuracy of these methods when the amount of training data available is such that repeatedly examining this data, as bagging and boosting require, is impractical. Our theoretical and empirical results give us the confidence that our online algorithms achieve this

goal.

Chapter 2

Background

In this chapter, we first introduce batch supervised learning in general as well as the specific algorithms that we use in this thesis. We then discuss the motivation for and existing work in the area of ensemble learning. We introduce the bagging and boosting algorithms because the online algorithms that we present in this thesis are derived from them. We then introduce and discuss online learning. We then describe the work that has been done in online ensemble learning, thereby motivating the work presented in this thesis.

2.1 Batch Supervised Learning

A batch supervised learning algorithm L_b takes a training set T as its input. The training set consists of N *examples* or *instances*. It is assumed that there is a distribution $\mathcal{D}$ from which each training example is drawn independently. The ith example is of the form (x_i, y_i), where x_i is a vector of values of several features or attributes and y_i represents the value to be predicted. In a classification problem, y_i represents one or more classes to which the example represented by x_i belongs. In a regression problem, y_i is some other type of value to be predicted. In the previous chapter, we gave the example of a classification problem in which we want to learn how to predict whether a credit card applicant is likely to default on his credit card given certain information such as average daily credit card balance, other credit cards held, and frequency of late payment. In this problem, each of the N examples in the training set would represent one current credit card holder for whom we

know whether he has defaulted up to now or not. If an applicant has an average daily credit card balance of $1500, five other credit cards, and pays late ten percent of the time, and has not defaulted so far, then he may be represented as $(x, y) = ((1500, 5, 10), No)$.

The output of a supervised learning algorithm is a hypothesis h that approximates the unknown mapping from the inputs to the outputs. In our example, h would map from information about the credit card applicant to a prediction of whether the applicant will default. In the experiments that we present in this thesis, we have a *test set*—a set of examples that we use to test how well the hypothesis h predicts the outputs on new examples. The examples in S are assumed to be independent and identically distributed (i.i.d.) draws from the same distribution $\mathcal{D}$ from which the examples in T were drawn. We measure the error of h on the test set S as the proportion of test cases that h misclassifies:

$$\frac{1}{|S|} \sum_{(x,y) \in S} I(h(x) \neq y)$$

where S is the test set and $I(v)$ is the indicator function—it returns 1 if v is true and 0 otherwise.

In this thesis, we use batch supervised learning algorithms for decision trees, decision stumps, Naive Bayes, and neural networks. We define each of these now.

2.1.1 Decision Trees and Decision Stumps

A decision tree is a tree structure consisting of nonterminal nodes, leaf nodes, and arcs. An example of a decision tree that may be produced in our example credit-card domain is depicted in Figure 2.1. Each nonterminal node represents a test on an attribute value. In the example decision tree, the top node (called the *root* node) tests whether the example to be classified has an income attribute value that is less than $50000. If it does not, then we go down the right arc, where we see a leaf node that says "NO." A leaf node contains the classification to be returned if we reach it. In this case, if the income is at least $50000, then we do not examine any other attribute values and we predict that the person will not default on his credit card. If the income is less than $50000, then we go down the left arc to the next nonterminal node, which tests whether the average daily checking account balance is less than $500. If it is, then we predict that the person will default on his credit card,

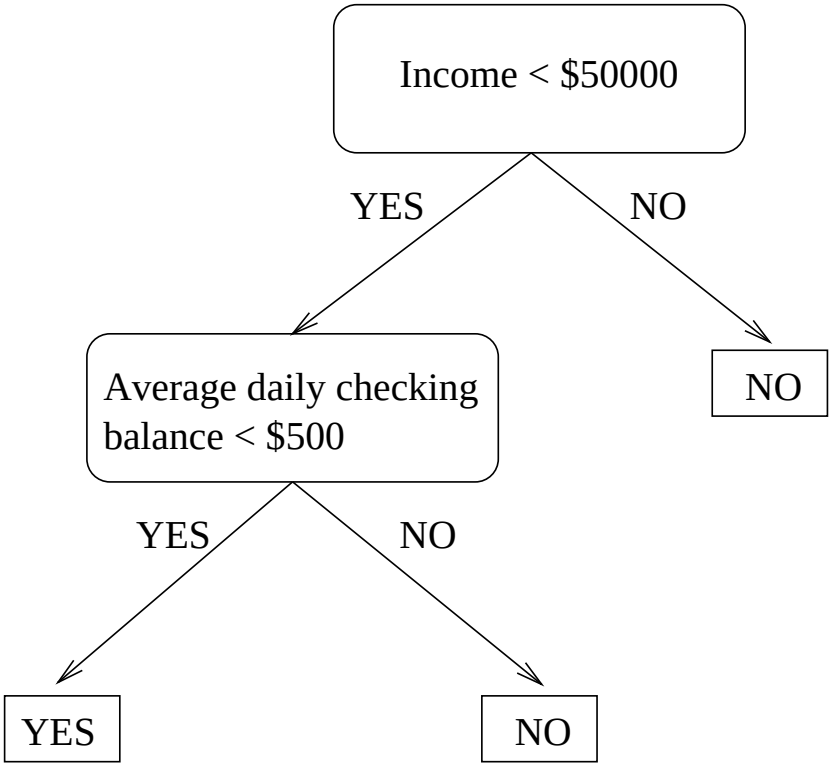

Figure 2.1: An example of a decision tree.

otherwise we predict that he will not default. Note that this tree never examines a person's total savings.

Decision trees are constructed in a top-down manner. A decision tree learning algorithm (ID3) is shown in Figure 2.2. If all the examples are of the same class, then the algorithm just returns a leaf node of that class. If there are no attributes left with which to construct a nonterminal node, then the algorithm has to return a leaf node. It returns a leaf node labeled with the class most frequently seen in the training set. If none of these conditions is true, then the algorithm finds the one attribute value test that comes closest to splitting the entire training set into groups such that each group only contains examples of one class. When such an attribute is selected, the training set is split according to that attribute. That is, for each value v of the attribute, a training set t_v is constructed such that all the examples in t_v have value v for the chosen attribute. The learning algorithm is called recursively on each of these training sets.

Decision_Tree_Learning(T,A,g)

if $T_{i,g}$ is the same for all $i \in \{1, 2, \ldots, |T|\}$,

return a leaf node labeled $T_{1,g}$.

else if $|A| = 0$,

return a leaf node labeled $\text{argmax}_c \sum_{i=1}^{|T|} I(T_{i,g} = c)$.

else

$b = Choose\_Best\_Attribute(T, A)$

Set $tree$ to be a nonterminal node with test b.

for each value v of attribute b,

$t_v = \emptyset$

for each example $T_i \in T$,

$v = T_{i,b}$

Add example T_i to set t_v.

for each value v of attribute b,

$subtree = Decision\_Tree\_Learning(t_v, A - b)$

Add a branch to $tree$ labeled v with subtree $subtree$.

return $tree$.

Figure 2.2: Decision Tree Learning Algorithm. This algorithm takes a training set T, attribute set A, goal attribute g, and default class d as inputs and returns a decision tree. T_i denotes the ith training example, $T_{i,b}$ denotes example i's value for attribute b, and $T_{i,g}$ denotes example i's value for the goal attribute g.

A decision stump is a decision tree that is restricted to having only one nonterminal node. That is, the decision tree algorithm is used to find the one attribute test that comes closest to splitting all the training data into groups of examples of the same class. One branch is constructed for each value of the attribute, and a leaf node is constructed at the end of that branch. The training set is split into groups according to the value of that attribute and each group is attached to the appropriate leaf. The class label most frequently seen in each group is the class label assigned to the corresponding leaf.

2.1.2 Naive Bayes

Bayes's theorem tells us how to optimally predict the class of an example. For an example x, we should predict the class y that maximizes

$$P(Y = y|X = x) = \frac{P(Y = y)P(X = x|Y = y)}{P(X = x)}.$$

Define A to be the set of attributes. If all the attributes are independent given the class, then we can rewrite $P(X = x|Y = y)$ as $\prod_{a=1}^{|A|} P(X_i = x_{(i)}|Y = y)$, where $x_{(a)}$ is the ath attribute value of example x. Each of the probabilities $P(Y = y)$ and $P(X_a = x_{(a)}|Y = y)$ for all classes Y and all possible values of all attributes X_a is estimated from a training set. For example, $P(X_a = x_{(a)}|Y = y)$ would be the fraction of class-y training examples that have $x_{(a)}$ as their ath attribute value. Estimating $P(X = x)$ is unnecessary because it is the same for all classes; therefore, we ignore it. Now we can return the class that maximizes

$$P(Y = y) \prod_{a=1}^{|A|} P(X_a = x_{(a)}|Y = y). \tag{2.1}$$

This is known as the Naive Bayes classifier. The algorithm that we use is shown in Figure 2.3. For each training example, we just increment the appropriate counts: N is the number of training examples seen so far, N_y is the number of examples in class y, and $N_{y,x_{(a)}}$ is the number of examples in class y having $x_{(a)}$ as their value for attribute a. $P(Y = y)$ is estimated by $\frac{N_y}{N}$ and, for all classes y and attribute values $x_{(a)}$, $P(X_a = x_{(a)}|Y = y)$ is estimated by $\frac{N_{y,x_{(a)}}}{N_y}$. The algorithm returns a classification function that returns, for an example x, the class y that maximizes

$$\frac{N_y}{N} \prod_{a=1}^{|A|} \frac{N_{y,x_{(a)}}}{N_y}.$$

Every factor in this equation estimates its corresponding factor in Equation 2.1.

2.1.3 Neural Networks

The multilayer perceptron is the most common neural network representation. It is often depicted as a directed graph consisting of nodes and arcs—an example is shown in

Naive-Bayes-Learning(T,A)

for each training example $(x, y) \in T$,

Increment N

Increment N_y

for $a \in \{1, 2, \ldots, A\}$

Increment $N_{y,x_{(a)}}$

return $h(x) = \text{argmax}_{y \in Y} \frac{N_y}{N} \prod_{a=1}^{A} \frac{N_{y,x_{(a)}}}{N_y}$

Figure 2.3: Naive Bayes Learning Algorithm. This algorithm takes a training set T and attribute set A as inputs and returns a Naive Bayes classifier. N is the number of training examples seen so far, N_y is the number of examples in class y, and $N_{y,x_{(a)}}$ is the number of examples in class y that have $x_{(a)}$ as their value for attribute a.

Figure 2.4. Each column of nodes is a *layer*. The leftmost layer is the *input layer*. The inputs of an example to be classified are entered into the input layer. The second layer is the *hidden layer*. The third layer is the *output layer*. Information flows from the input layer to the hidden layer to the output layer via a set of arcs. Note that the nodes within a layer are not directly connected. In our example, every node in one layer is connected to every node in the next layer, but this is not required in general. Also, a neural network can have more or less than one hidden layer and can have any number of nodes in each hidden layer.

Each non-input node, its incoming arcs, and its single outgoing arc constitute a *neuron*, which is the basic computational element of a neural network. Each incoming arc multiplies the value coming from its origin node by the weight assigned to that arc and sends the result to the destination node. The destination node adds the values presented to it by all the incoming arcs, transforms it with a nonlinear activation function (to be described later), and then sends the result along the outgoing arc. For example, the output of a hidden node z_j in our example neural network is

$$z_j = g\left(\sum_{i=1}^{|A|} w_{i,j}^{(1)} x_i\right)$$

where $w_{i,j}^{(k)}$ is the weight on the arc in the kth layer of arcs that goes from unit i in the kth layer of nodes to unit j in the next layer (so $w_{i,j}^{(1)}$ is the weight on the arc that goes from

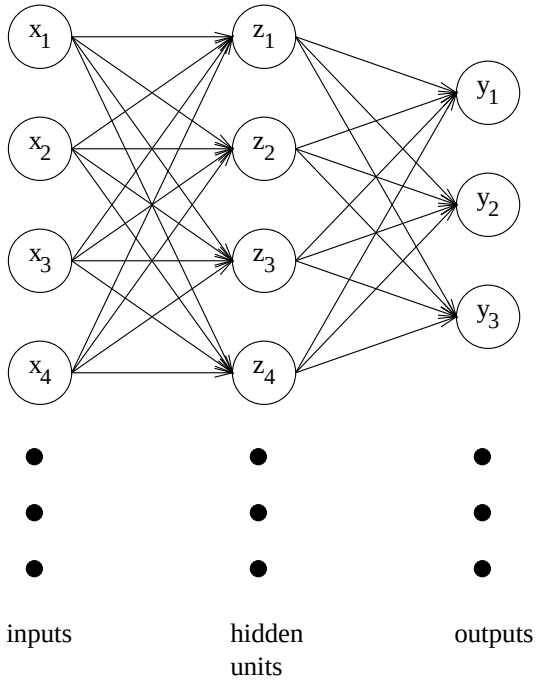

Figure 2.4: An example of a multilayer feedforward perceptron.

input unit i to hidden unit j) and g is a nonlinear activation function. A commonly used activation function is the sigmoid function:

$$g(a) \equiv \frac{1}{1 + exp(-a)}.$$

The output of an output node y_j is

$$y_j = g\left(\sum_{i=1}^{Z} w_{i,j}^{(2)} z_i\right)$$

where Z is the number of hidden units. The outputs are clearly nonlinear functions of the inputs. Neural networks used for classification problems typically have one output per class. The example neural network depicted in Figure 2.4 is of this type. The outputs lie in the range $[0, 1]$. Each output value is a measure of the network's confidence that the example presented to it is a member of that output's corresponding class. Therefore, the class corresponding to the highest output value is returned as the prediction.

Neural network learning performs nonlinear regression given a training set. The most widely used method for setting the weights in a neural network is the backpropagation

algorithm (Bryson & Ho, 1969; Rumelhart, Hinton, & Williams, 1986). For each training example in the training set, its inputs are presented to the input layer of the network and the predicted outputs are calculated. The difference between each predicted output and the corresponding target output is calculated. This error is then propagated back through the network and the weights on the arcs of the networks are adjusted so that if the training example is presented to the network again, then the error would be less. The learning algorithm typically cycles through the training set many times—each cycle is called an *epoch* in the neural network literature.

2.2 Ensemble Learning

2.2.1 Motivation

As we discussed in Chapter 1, *ensembles* are combinations of multiple *base* models, each of which may be a traditional machine learning model such as a decision tree or Naive Bayes classifier. When a new example is to be classified, it is presented to the ensemble's base models and their outputs are combined in some manner (e.g., voting or averaging) to yield the ensemble's prediction. Intuitively, we would like to have base models that perform well and do not make highly-correlated errors. We can see the intuition behind this point graphically in Figure 2.5. The goal of the learning problem depicted in the figure is to separate the positive examples ('+') from the negative examples ('-'). The figure depicts an ensemble of three linear classifiers. For example, line C classifies examples above it as negative examples and examples below it as positive examples. Note that none of the three lines separates the positive and negative examples perfectly. For example, line C misclassifies all the positive examples in the top half of the figure. Indeed, no straight line can separate the positive examples from the negative examples. However, the ensemble of three lines, where each line gets one vote, correctly classifies all the examples—for every example, at least two of the three linear classifiers correctly classifies it, so the majority is always correct. This is the result of having three very different linear classifiers in the ensemble. This example clearly depicts the need to have base models whose errors are not highly correlated. If all the linear classifiers make mistakes on the same examples

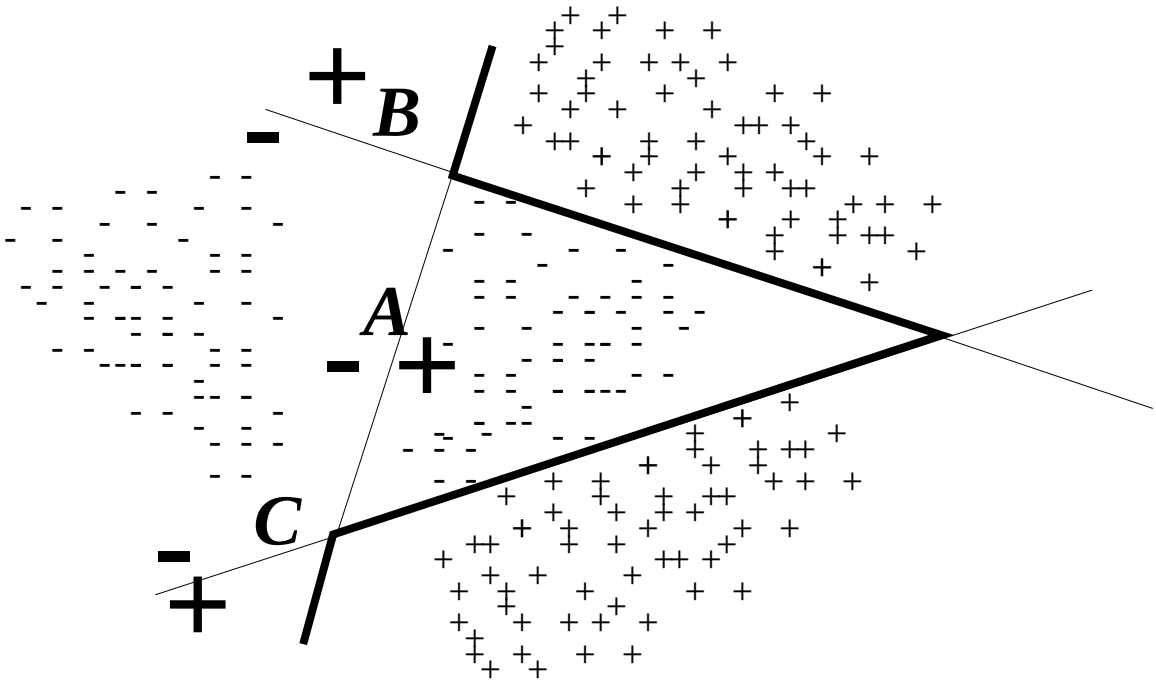

Figure 2.5: An ensemble of linear classifiers. Each line A, B, and C is a linear classifier. The boldface line is the ensemble that classifies new examples by returning the majority vote of A, B, and C.

(for example if the ensemble consisted of three copies of line A), then a majority vote over the lines would also make mistakes on the same examples, yielding no performance improvement.

Another way of explaining the superior performance of the ensemble is that the class of ensemble models has greater expressive power than the class of individual base models. We pointed out earlier that, in the figure, no straight line can separate the positive examples from the negative examples. Our ensemble is a piecewise linear classifier (the bold line), which is able to perfectly separate the positive and negative examples. This is because the class of piecewise linear classifiers has more expressive power than the class of single linear classifier.

The intuition that we have just described has been formalized (Tumer & Ghosh, 1996; Tumer, 1996). Ensemble learning can be justified in terms of the *bias* and *variance* of the learned model. It has been shown that, as the correlations of the errors made by the base models decrease, the variance of the error of the ensemble decreases and is less than the

variance of the error of any single base model. If E_{add} is the average additional error of the base models (beyond the Bayes error, which is the minimum possible error that can be obtained), E_{add}^{ave} is the additional error of an ensemble that computes the average of the base models' outputs, and ρ is the average correlation of the errors of the base models, then Tumer and Ghosh (1996) have shown that

$$E_{add}^{ave} = \frac{1 + \rho(M-1)}{M} E_{add},$$

where M is the number of base models in the ensemble. The effect of the correlations of the errors made by the base models is made clear by this equation. If the base models always agree, then $\rho = 1$; therefore, the errors of the ensemble and the base models would be the same and the ensemble would not yield any improvement. If the base models' errors are independent, then $\rho = 0$, which means the ensemble's error is reduced by a factor of M relative to the base models' errors. It is possible to do even better by having base models with anti-correlated errors. If $\rho = \frac{1}{M-1}$, then the ensemble's error would be zero.

Ensemble learning can be seen as a tractable approximation to full Bayesian learning. In full Bayesian learning, the final learned model is a mixture of a very large set of models—typically all models in a given family (e.g., all decision stumps). If we are interested in predicting some quantity Y, and we have a set of models h_i and a training set T, then the final learned model is

$$P(Y|T) = \sum_i P(Y|h_i)P(h_i|T) = \sum_i P(Y|h_i)\frac{P(T|h_i)P(h_i)}{P(T)}.$$

Full Bayesian learning combines the explanatory power of all the models ($P(Y|h_i)$) weighted by the posterior probability of the models given the training set ($P(h_i|T)$). However, full Bayesian learning is intractable because it uses a very large (possibly infinite) set of models. Ensembles can be seen as approximating full Bayesian learning by using a mixture of a small set of the models having the highest posterior probabilities ($P(h_i|T)$) or highest likelihoods ($P(T|h_i)$). Ensemble learning lies between traditional learning with single models and full Bayesian learning in that it uses an intermediate number of models.

2.2.2 Ensemble Learning Algorithms

The ensemble example shown in Figure 2.5 is an artificial example. We normally cannot expect to get base models that make mistakes on completely separate parts of the input space and ensembles that classify all the examples correctly. However, there are many algorithms that attempt to generate a pool of base models that make errors that are as uncorrelated as possible. Methods such as bagging (Breiman, 1994), boosting (Freund & Schapire, 1996)[1], and cross-validation partitioning (Krogh & Vedelsby, 1995; Tumer & Ghosh, 1996) promote diversity by presenting each base model with a different subset of training examples or different weight distributions over the examples. The method of error-correcting output codes (Dietterich & Bakiri, 1995) presents each base model with the same training inputs but different labels—for each base model, the algorithm constructs a random partitioning of the labels into two new labels. The training data with new labels are used to train the base models. Input Decimation (Oza & Tumer, 1999, 2001; Tumer & Oza, 1999) and Stochastic Attribute Selection Committees (SASC) (Zheng & Webb, 1998) instead promote diversity by presenting each base model with a different subset of input features. SASC presents each base model (the number of base models is determined by hand) with a random subset of features. Input Decimation uses as many base models as there are classes and trains each base model using a subset of features having maximum correlation with the presence or absence of its corresponding class. However, in both SASC and Input Decimation, all training patterns are used with equal weight to train all the base models.

The methods we have just discussed are distinguished by their methods of training the base models. We can also distinguish methods by the way they combine their base models. Majority voting is one of the most basic methods of combining (Battiti & Colla, 1994; Hansen & Salamon, 1990) and is the method used in bagging. If the classifiers provide probability values, simple averaging is an effective combining method and has received a lot of attention (Lincoln & Skrzypek, 1990; Perrone & Cooper, 1993; Tumer & Ghosh, 1996). Weighted averaging has also been proposed and different methods for computing the weights of the classifiers have been examined (Benediktsson, Sveinsson, Ersoy, &

[1] We explain bagging and boosting in more detail later in this chapter.

Swain, 1994; Hashem & Schmeiser, 1993; Jacobs, 1995; Jordan & Jacobs, 1994; Lincoln & Skrzypek, 1990; Merz, 1999). Boosting uses a weighted averaging method where each base model's weight is proportional to its classification accuracy. The combining schemes described so far are linear combining techniques, which have been mathematically analyzed in depth (Breiman, 1994; Hashem & Schmeiser, 1993; Perrone & Cooper, 1993; Tumer & Ghosh, 1996). There are also non-linear ensemble schemes include rank-based combining (Al-Ghoneim & Vijaya Kumar, 1995; Ho, Hull, & Srihari, 1994), belief-based methods (Rogova, 1994; Xu, Krzyzak, & Suen, 1992; Yang & Singh, 1994), and order-statistic combiners (Tumer & Ghosh, 1998; Tumer, 1996).

In this thesis, the ensemble methods that we use are bagging and boosting, which we explain now.

Bagging

*B*ootstrap *Agg*regat*ing* (bagging) generates multiple bootstrap training sets from the original training set and uses each of them to generate a classifier for inclusion in the ensemble. The algorithms for bagging and doing the bootstrap sampling (sampling with replacement) are shown in Figure 2.6. To create a bootstrap training set from a training set of size N, we perform N Multinomial trials, where in each trial, we draw one of the N examples. Each example has probability $1/N$ of being drawn in each trial. The second algorithm shown in Figure 2.6 does exactly this—N times, the algorithm chooses a number r from 1 to N and adds the rth training example to the bootstrap training set S. Clearly, some of the original training examples will not be selected for inclusion in the bootstrap training set and others will be chosen one time or more. In bagging, we create M such bootstrap training sets and then generate classifiers using each of them. Bagging returns a function $h(x)$ that classifies new examples by returning the class y that gets the maximum number of votes from the base models $h_1, h_2, \ldots, h_M$. In bagging, the M bootstrap training sets that are created are likely to have some differences. If these differences are enough to induce noticeable differences among the M base models while leaving their performances reasonably good, then the ensemble will probably perform better than the base models individually. (Breiman, 1996a) defines models as *unstable* if differences in their training

Bagging(T,M)

For each $m = 1, 2, \ldots, M$,

$T_m = Sample\_With\_Replacement(T, N)$

$h_m = L_b(T_m)$

Return $h_{fin}(x) = \text{argmax}_{y \in Y} \sum_{m=1}^{M} I(h_m(x) = y)$

Sample_With_Replacement(T,N)

$S = \emptyset$

For $i = 1, 2, \ldots, N$,

$r = random\_integer(1, N)$

Add $T[r]$ to S.

Return S.

Figure 2.6: Batch Bagging Algorithm and Sampling with Replacement: T is the original training set of N examples, M is the number of base models to be learned, L_b is the base model learning algorithm, the h_i's are the classification functions that take a new example as input and return the predicted class from the set of possible classes Y, $random\_integer(a, b)$ is a function that returns each of the integers from a to b with equal probability, and $I(A)$ is the indicator function that returns 1 if event A is true and 0 otherwise.

sets tend to induce significant differences in the models and *stable* if not. Another way of stating this is that bagging does more to reduce the variance in the base models than the bias, so bagging performs best relative to its base models when the base models have high variance and low bias. He notes that decision trees are unstable, which explains why bagged decision trees often outperform individual decision trees; however, Naive Bayes classifiers are stable, which explains why bagging with Naive Bayes classifiers tends not to improve upon individual Naive Bayes classifiers.

Boosting

The AdaBoost algorithm, which is the boosting algorithm that we use, generates a sequence of base models with different weight distributions over the training set. The AdaBoost algorithm is shown in Figure 2.7. Its inputs are a set of N training examples, a base model learning algorithm L_b, and the number M of base models that we wish to combine.

AdaBoost$(\{(x_1, y_1), \ldots, (x_N, y_N)\}, L_b, M)$

Initialize $D_1(n) = 1/N$ for all $n \in \{1, 2, \ldots, N\}$.

For $m = 1, 2, \ldots, M$:

$h_m = L_b(\{(x_1, y_1), \ldots, (x_N, y_N)\}, D_m)$.

Calculate the error of $h_m : \epsilon_m = \sum_{n:h_m(x_n) \neq y_n} D_m(n)$.

If $\epsilon_m \geq 1/2$ then,

set $M = m - 1$ and abort this loop.

Update distribution D_m:

$$D_{m+1}(n) = D_m(n) \times \begin{cases} \frac{1}{2(1-\epsilon_m)} & \text{if } h_m(x_n) = y_n \\ \frac{1}{2\epsilon_m} & \text{otherwise} \end{cases}$$

Output the final hypothesis:

$h_{fin}(x) = \text{argmax}_{y \in Y} \sum_{m:h_m(x)=y} log\frac{1-\epsilon_m}{\epsilon_m}$.

Figure 2.7: AdaBoost algorithm: $\{(x_1, y_1), \ldots, (x_N, y_N)\}$ is the set of training examples, L_b is the base model learning algorithm, and M is the number of base models to be generated.

AdaBoost was originally designed for two-class classification problems; therefore, for this explanation we will assume that there are two possible classes. However, AdaBoost is regularly used with a larger number of classes. The first step in Adaboost is to construct an initial distribution of weights D_1 over the training set. The first distribution in AdaBoost is one that assigns equal weight to all N training examples. We now enter the loop in the algorithm. To construct the first base model, we call L_b with distribution D_1 over the training set.[2] After getting back a hypothesis h_1, we calculate its error ϵ_1 on the training set itself, which is just the sum of the weights of the training examples that h_1 misclassifies. We require that $\epsilon_1 < 1/2$ (this is the *weak learning* assumption—the error should be less than what we would achieve through randomly guessing the class)—if this condition is not satisfied, then we stop and return the ensemble consisting of the previously-generated base models. If this condition is satisfied, then we calculate a new distribution D_2 over the training examples as follows. Examples that were correctly classified by h_1 have their weights

[2] If L_b cannot take a weighted training set, then one can call it with a training set generated by sampling with replacement from the original training set according to the distribution D_m.

multiplied by $\frac{1}{2(1-\epsilon_1)}$. Examples that were misclassified by h_1 have their weights multiplied by $\frac{1}{2\epsilon_1}$. Note that, because of our condition $\epsilon_1 < 1/2$, correctly classified examples have their weights reduced and misclassified examples have their weights increased. Specifically, examples that h_1 misclassified have their total weight increased to $1/2$ under D_2 and examples that h_1 correctly classified have their total weight reduced to $1/2$ under D_2. We then go into the next iteration of the loop to construct base model h_2 using the training set and the new distribution D_2. We construct M base models in this fashion. The ensemble returned by AdaBoost is a function that takes a new example as input and returns the class that gets the maximum weighted vote over the M base models, where each base model's weight is $log(\frac{1-\epsilon_m}{\epsilon_m})$, which is proportional to the base model's accuracy on the weighted training set presented to it.

Clearly, the heart of AdaBoost is the distribution updating step. The idea behind it is as follows. We can see from the algorithm that ϵ_m is the sum of the weights of the misclassified examples. The misclassified examples' weights are multiplied by $\frac{1}{2\epsilon_m}$, so that the sum of their weights is increased to $\epsilon_m * \frac{1}{2\epsilon_m} = \frac{1}{2}$. The correctly classified examples start out having total weight $1 - \epsilon_m$, but their weights are multiplied by $\frac{1}{2(1-\epsilon_m)}$, therefore, the sum of their weights decreases to $(1 - \epsilon_m) * \frac{1}{2(1-\epsilon_m)} = \frac{1}{2}$. The point of this weight adjustment is that the next base model will be generated by a weak learner (i.e., the base model will have error less than $1/2$); therefore, at least some of the examples misclassified by the previous base model will have to be learned.

Boosting does more to reduce bias than variance. For this reason, boosting tends to improve upon its base models most when they have high bias and low variance. Examples of such models are Naive Bayes classifiers and decision stumps (decision trees of depth one). Boosting's bias reduction comes from the way it adjusts its distribution over the training set. The examples that a base model misclassifies have their weights increased, causing the base model learning algorithm to focus more on those examples. If the base model learning algorithm is biased against certain training examples, those examples get more weight, yielding the possibility of correcting this bias. However, this method of adjusting the training set distribution causes boosting to have difficulty when the training data is noisy (Dietterich, 2000). Noisy examples are normally difficult to learn. Because of this, the weights assigned to the noisy examples tend to be higher than for the other

examples, often causing boosting to focus too much on those noisy examples and overfit the data.

2.3 Online Learning

Online learning is the area of machine learning concerned with learning each training example once (perhaps as it arrives) and never examining it again. Online learning is necessary when data arrives continuously so that it may be impractical to store data for batch learning or when the dataset is large enough that multiple passes through the dataset would take too long. An online learning algorithm L_o takes as its input a current hypothesis h and a new training example (x, y). The algorithm returns a new hypothesis that is updated to reflect the new example. Clearly, an online learning algorithm can be used wherever a batch algorithm is required by simply calling the online algorithm once for each example in the training set. A *lossless* online learning algorithm is an algorithm that returns a hypothesis identical to what its corresponding batch algorithm would return given the same training set.

Some researchers have developed online algorithms for learning traditional machine learning models such as decision trees—in this thesis, we use the lossless online decision tree learning algorithm of (Utgoff, Berkman, & Clouse, 1997). Given an existing decision tree and a new example, this algorithm adds the example to the example sets at the appropriate nonterminal and leaf nodes and then confirms that all the attributes at the nonterminal nodes and the class at the leaf node are still the best. If any of them are not, then they are updated and the subtrees below them are rechecked as necessary. Naive Bayes model learning is performed the same way in online and batch modes, so online learning is clearly lossless. Batch neural network learning is often performed by making multiple passes (known in the literature as *epochs*) through the data with each training example processed one at a time. So neural networks can be learned online by simply making one pass through the data. However, there would clearly be some loss associated with only making one pass through the data.

Two online algorithms that have been well-analyzed in the theoretical machine learn-

Initial conditions: For all $i \in \{1, \ldots, M\}$, $w_i = 1$.

Weighted-Majority(x)

For $i = 1, \ldots, M$,

$y_i = h_i(x)$.

If $\sum_{i:y_i=1} w_i \geq \sum_{i:y_i=0} w_i$, then return 1, else return 0.

If the target output y is not available, then exit.

For $i = 1, \ldots, M$,

If $y_i \neq y$ then $w_i \leftarrow \frac{w_i}{2}$.

Figure 2.8: Weighted Majority Algorithm: $\mathbf{w} = [\mathbf{w_1 w_2} \ldots \mathbf{w_M}]$ is the vector of weights corresponding to the M predictors, x is the latest example to arrive, y is the correct classification of example x, the y_i are the predictions of the experts h_i.

ing literature are the Weighted Majority Algorithm (Littlestone & Warmuth, 1994) and the Winnow Algorithm (Littlestone, 1988) (see (Blum, 1996) for a brief review of these algorithms). Both the Weighted Majority and Winnow algorithms maintain weights on several predictors and increase or decrease their weights depending on whether the individual predictors correctly classify or misclassify, respectively, the training example currently being considered. For example, Figure 2.8 contains the Weighted Majority algorithm as listed in (Blum, 1996).

The first step in the algorithm is an initialization step that is only performed once—it sets the weights w_i of all the predictors to 1. The remaining steps are performed once for each training example as it arrives. First, the algorithm calculates the predictions of each predictor on the new example x. The ensemble's prediction is then returned—it is the class that gets the maximum total weighted vote over all the predictors. Each predictor that makes a mistake on that example has its weight reduced in half. Weighted Majority and Winnow have shown promise in the few empirical tests that have been performed (e.g., (Armstrong, Freitag, Joachims, & Mitchell, 1995; Blum, 1997)). These algorithms have also been proven to perform not much worse than the best individual predictor. For example, given a sequence of training examples and the pool of predictors $\mathcal{A}$, if there is a predictor that makes at most m mistakes, then the Weighted Majority algorithm will make

at most $c(log|A| + m)$ mistakes, where c is a constant. This is not surprising, since we would expect the bad predictors to make many more mistakes than the good predictors, leading to much lower weights for the bad predictors. Eventually only the good predictors would influence the prediction of the entire model.

Work in universal prediction (Merhav & Feder, 1998; Singer & Feder, 1999) has yielded algorithms that produce combined predictors that also are proven in the worst case to perform not much worse than the best individual predictor. Additionally, Singer and Feder (1999) discuss a universal linear prediction algorithm that produces a weighted mixture of sequential linear predictors. Specifically, universal prediction is concerned with the problem of predicting the tth observation $x[t]$ given the $t-1$ observations $x[1], x[2], \ldots, x[t-1]$ seen so far. We would like to use a method that minimizes the difference between our prediction $\hat{x}[t]$ and the observed value $x[t]$. A linear predictor of the form $\hat{x}_p[t] = \sum_{j=1}^{p} c_{p,j}^{(t-1)} x[t-j]$ can be used, where p is the order (the number of past observations used to make a prediction) and $c_{p,j}^{(t-1)}$ for $j \in \{1, 2, \ldots, p\}$ are coefficients obtained by minimizing the sum of squared differences between the previous $t-1$ observations and predictions: $\sum_{j=1}^{t-1}(x[j] - \hat{x}[j])^2 = \sum_{j=1}^{t-1}(x[j] - \sum_{k=1}^{p} c_{p,k}^{t-1} x[j-k])^2$. Using a pth-order linear predictor requires us to select a particular order p, which is normally very difficult. This motivates the use of a universal predictor $\hat{x}_u$, which yields a performance-weighted combination of the outputs of each of the different sequential linear predictors of orders 1 through some M:

$$\hat{x}_u[t] = \sum_{k=1}^{M} \mu_k[t] \hat{x}_k[t],$$

where

$$\begin{aligned} \mu_k[t] &= \frac{exp(-\frac{1}{2c} l_{t-1}(x, \hat{x}_k))}{\sum_{j=1}^{M} exp(-\frac{1}{2c} l_{t-1}(x, \hat{x}_j))}, \\ l_t(x, \hat{x}_k) &= \sum_{s=1}^{t} (x[t] - \hat{x}_k[t])^2. \end{aligned}$$

We can compare the universal predictor to the full Bayesian model shown in Equation 2.2. $\mu_k[t]$ is the universal predictor's version of $P(Y|h_i)$, i.e., $\mu_k[t]$ is a normalized measure of the predictive performance of the kth-order model, just as $P(Y|h_i)$ is a normalized measure of the performance of hypothesis h_i. These measures are used to weight the models being combined.

The universal predictor is a special case of the full Bayesian model because of the special structure of the predictors being combined. Specifically, the sequential linear predictors used in a universal predictor have much in common, so that they can be computed in a time-recursive and order-recursive manner (see (Singer & Feder, 1999) for the algorithm). This recursive nature means that the complexity of the universal prediction algorithm is only $O(Mn)$, where n is the total length of the sequence x. The full Bayesian learner is more general in that its models need not have such a structure—the only requirement is that the models h_i in the hypothesis class be mutually exclusive such that $\sum_i P(h_i|T) = 1$. Therefore, the complexity of the full Bayesian learner could, in the worst case, be the number of models multiplied by the complexity of learning each model. Most ensembles also do not have such a structure among their base models. For example, in ensembles of decision trees or neural networks, there is no recursive structure among the different instances of the models; therefore the complexity of the learning algorithm is the number of models multiplied by the complexity of each base model's learning algorithm.

The reader may have noticed that Weighted Majority, Winnow, and the universal prediction algorithm just described may be thought of as ensemble learning algorithms because they combine multiple predictors. Ensemble learning algorithms generally perform better than their base models; therefore, one may wonder why Weighted Majority, Winnow, and the universal prediction algorithm are only proven to have error that is at most slightly worse than the best predictor. This is because they do not attempt to bring about diversity among their predictors the way ensemble algorithms do. Many ensemble algorithms require their base models to use different input features, outputs, training examples, distributions over training examples, or initial parameters to bring about diversity. On the other hand, Weighted Majority, Winnow, and the universal prediction algorithm do not assume that there are any differences in their predictors. Without any way to reduce the correlations in the errors made by the predictors, these algorithms are unable to guarantee performance better than any individual predictors.

Breiman(T,M,N_b)

for $m = 1, \ldots, M$,

Initialize $e = 0, t = 0, T_m = \emptyset$.

Do until $|T_m| = N_b$,

Get the next training example (x, y) in T.

$t \leftarrow t + 1$.

If $m = 1$ or $y \neq \text{argmax}_c \sum_{i=1}^{m-1} I(h_i(x) = c)$,

then add (x, y) to T_m and $e \leftarrow e + 1$,

else add (x, y) to T_m with probability $\frac{e^{TR}(m)}{1-e^{TR}(m)}$, where

if $m = 1$

then $e^{TR}(1) = \frac{e}{t}$,

else $e^{TR}(m) = 0.75e^{TR}(m-1) + 0.25\frac{e}{t}$.

$h_m = L_b(T_m)$.

return $h_{fin}(x) = \text{argmax}_y \sum_{m=1}^{M} I(h_m(x) = y)$

Figure 2.9: Breiman's blocked ensemble algorithm: Among the inputs, T is the training set, M is the number of base models to be constructed, and N_b is the size of each base model's training set (T_m for $m \in \{1, 2, \ldots, M\}$). L_b is the base model learning algorithm, t is the number of training examples examined in the process of creating each T_m and e is the number of these examples that the ensemble previously constructed base models ($h_1, \ldots, h_{m-1}$) misclassifies.

2.4 Online Ensemble Learning

There has been some recent work on learning ensembles in an online fashion. Breiman (1999) devised a "blocked" online boosting algorithm that trains the base models using consecutive subsets of training examples of some fixed size. The algorithm's pseudocode is given in Figure 2.9. The user may choose M—the number of base models—to be some fixed value or may allow it to grow up to the maximum possible which is at most $|T|/N_b$, where T is the original training set and N_b is the user-chosen number of training examples used to create each base model. For the first base model, the first N_b training examples in the training set T are selected. To generate a training set for the mth base model for $m > 1$, the algorithm draws the next training example from T and classifies it by unweighted voting over the $m - 1$ base models generated so far. If the example is misclassified, then it

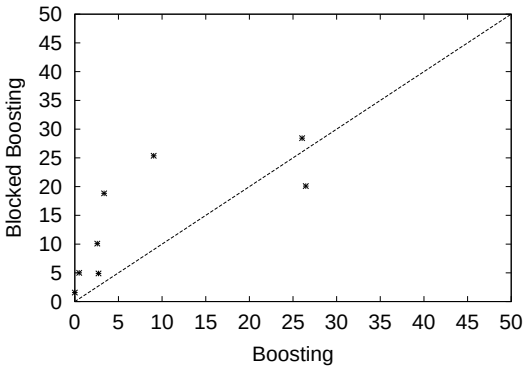

Figure 2.10: Test Error Rates: Boosting vs. Blocked Boosting with decision tree base models.

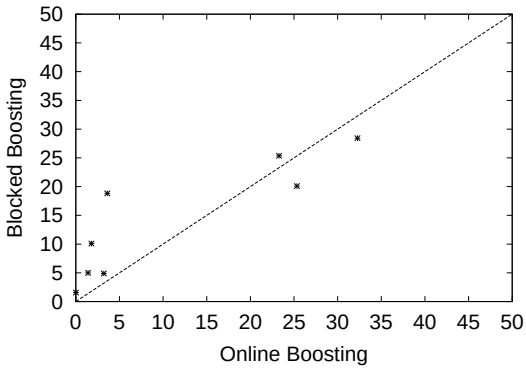

Figure 2.11: Test Error Rates: Online Boosting vs. Blocked Boosting with decision tree base models.

is included in the new base model's training set T_m; otherwise it is included in T_m with a probability proportional to the fraction of training examples drawn for this model that are misclassified (e/t) and the previous base model's error $e^{TR}(m-1)$ (done for the purpose of smoothing). This method of selecting examples is done in order to train each base model using a training set in which half the examples have been correctly classified by the ensemble consisting of the previous base models and half have been misclassified. This process of selecting examples is done until N_b examples have been selected for inclusion in T_m, at which time the base model learning algorithm L_b is called with T_m to get base model h_m. Breiman's algorithm returns a function that classifies a new example by returning the class that receives the maximum number of votes over the base models $h_1, h_2, \ldots, h_M$. Breiman discusses experiments with his algorithm using decision trees as base models and N_b ranging from 100 to 800. His experiments with one synthetic dataset showed that the test error decreased more rapidly when using larger subsets of training examples. However, each training example is only used to update one base model and each base model is only trained with N_b examples, which is a relatively small number. It is generally not clear when this is sufficient to achieve performance comparable to a typical batch ensemble algorithm in which all the training examples are available to generate all of the base models.

Figure 2.10 shows a scatterplot of the results of comparing AdaBoost to Breiman's blocked boosting algorithm on the first eight UCI datasets (Table 3.1) used in the experi-

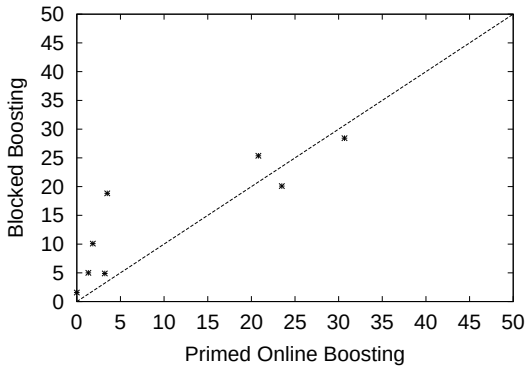

Figure 2.12: Test Error Rates: Primed Online Boosting vs. Blocked Boosting with decision tree base models.

ments described in Chapter 3 and Chapter 4. This scatterplot, like the other ones in this thesis, compares the error on the test set of two algorithms. Every point represents one dataset. The diagonal line contains the points at which the errors of the two algorithms are equal. In Figure 2.10, the points above the line represent experiments in which boosting had a lower test set error than blocked boosting. Points below the line represent experiments in which blocked boosting had the lower test set error. Figure 2.11 compares the online boosting algorithm we present in Chapter 4 with the blocked boosting algorithm. Figure 2.12 shows the results of comparing our primed online boosting algorithm presented in Chapter 4 with the blocked algorithm. In our primed algorithm, we train with the lesser of the first 20% of the training set or the first 10000 training examples in batch mode and train with the remainder in online mode. In the blocked boosting algorithm, each decision tree was trained with 100 examples ($N_b = 100$) except for the Promoters dataset, which only had 84 training examples, so we used $N_b = 50$. Overall, AdaBoost and both our online boosting and primed online boosting algorithms perform better than the blocked boosting algorithm.

Fern and Givan (2000) present both an online bagging algorithm and online boosting algorithm. Their online bagging algorithm is shown in Figure 2.13. This algorithm selects each new training example to update each base model with some probability p that the user fixes in advance. L_o is an online base model learning algorithm that takes a current

Online-Bag($\mathbf{h} = \{h_1, h_2, \ldots, h_M\}$,$(x, y)$,$p$,$L_o$**)**
for $m = 1, \ldots, M$,
With probability p, do
$h_m \leftarrow L_o(h_m, (x, y))$.

Figure 2.13: Online Bagging algorithm. $\mathbf{h} = \{h_1, h_2, \ldots, h_M\}$ is the set of base models to be updated, (x, y) is the next training example, p is the user-chosen probability that each example should be included in the next base model's training set, and L_o is the online base model learning algorithm that takes a base model and training example as inputs and returns the updated base model.

hypothesis and training example as input and returns a new hypothesis updated with the new example. In experiments with various settings for p and depth-limited decision trees as the base models, their online bagging algorithm never performed significantly better than a single decision tree. With low values of p, the ensembles' decision trees are quite diverse because their training sets tend to be very different; however, each tree gets too few training examples, causing each of them to perform poorly. Higher values of p allow the trees to get enough training data to perform well, but their training sets have enough in common to yield low diversity among the trees and little performance gain from combining. Figure 2.14 shows the results of comparing batch bagging to Fern and Givan's bagging algorithm on several UCI datasets. Figure 2.15 gives the results of comparing our online bagging algorithm from Chapter 3 to Fern and Givan's algorithm with $p = 0.7$, which gave them the best results in their tests. However, we allowed their algorithm to use decision trees with no depth limit in order to allow for a fair comparison. As we mentioned earlier, bagging tends to work best with base models having high variance and low bias. Therefore, using decision trees with no depth limit would tend to work best. Both batch bagging and our online bagging algorithm perform comparably to Fern and Givan's algorithm.

Fern and Givan's online boosting algorithm is an online version of Arc-x4 (Breiman, 1996b). Arc-x4 is similar to AdaBoost except that when a base model h_m is presented with a training example, that example is given weight $1 + w^4$, where w is the number of previous base models $h_1, \ldots, h_{m-1}$ that have misclassified it. The pseudocode for the online algorithm is shown in Figure 2.16. In this algorithm, each example is given weight

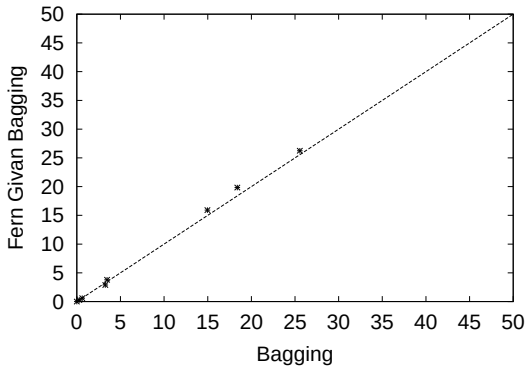

Figure 2.14: Test Error Rates: Bagging vs. Fixed Fraction Online Bagging with decision tree base models.

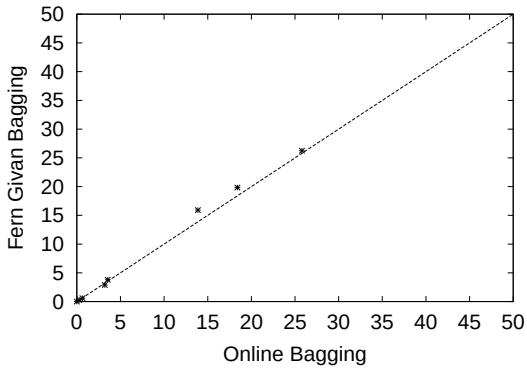

Figure 2.15: Test Error Rates: Online Bagging vs. Fixed Fraction Online Bagging with decision tree base models.

Online-Arc-x4(h $= \{h_1, h_2, \ldots, h_M\}$,$(x, y)$,$L_o$**)**

Initialize $w = 0$.

for $m = 1, \ldots, M$,

$h_m \leftarrow L_o(h_m, (x, y), 1 + w^4)$.

if $(h_m(x) \neq y)$ then $w \leftarrow w + 1$.

Figure 2.16: Online Boosting algorithm. $\mathbf{h} = \{h_1, h_2, \ldots, h_M\}$ is the set of base models to be updated, (x, y) is the next training example, and L_o is the online base model learning algorithm that takes a base model, training example, and its weight as inputs and returns the updated base model.

$1+w^4$ to update each base model just like Arc-x4. Here, L_o is an online base model learning algorithm that takes the current hypothesis, a training example, and its weight as input and returns a new hypothesis updated to reflect the new example with the supplied weight. This algorithm was tested on four machine learning datasets, three of which are part of the UCI Machine Learning Repository (Blake, Keogh, & Merz, 1999), and several branch prediction problems from computer architecture. The main goal of their work was to apply ensembles to branch prediction and similar resource-constrained online domains. For this reason, they allowed their algorithm a fixed amount of memory and examined the trade-off between having a larger number of shallow decision trees and having a smaller number of deep decision trees. Their results suggest that, given limited memory, a boosted ensemble

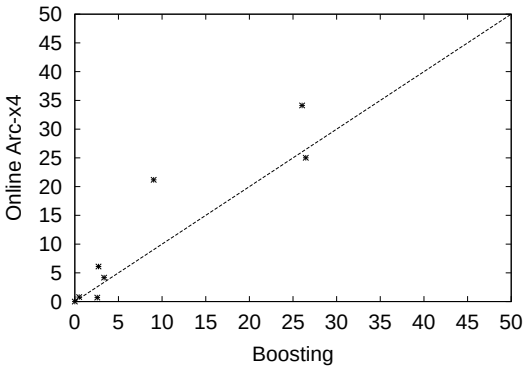

Figure 2.17: Test Error Rates: Boosting vs. Online Arc-x4 with decision tree base models.

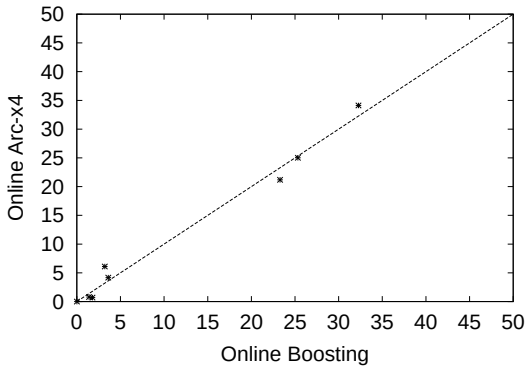

Figure 2.18: Test Error Rates: Online Boosting vs. Online Arc-x4 with decision tree base models.

with a greater number of smaller decision trees is generally superior to one with fewer large trees. They often achieved results much better than a single decision tree, but did not compare their results to any batch ensemble algorithms including Arc-x4 and AdaBoost. We compared AdaBoost and our original and primed online boosting algorithms to their online Arc-x4 algorithm and show the results in Figures 2.17, 2.18, and 2.19, respectively. We allowed their online Arc-x4 algorithm to use decision trees without depth limits but with pruning just as we did with AdaBoost and our online boosting algorithm. Batch boosting performs better than online Arc-x4. Our online boosting algorithm and online Arc-x4 perform comparably. Our primed online boosting algorithm outperformed online Arc-x4 slightly.

Our approach to online bagging and online boosting is different from the methods described in that we focus on reproducing the advantages of bagging and boosting in an online setting. Like the batch versions of bagging and boosting and most other batch ensemble algorithms, our algorithms make all the training data available for training all the base models and still obtain enough diversity among the base models to yield good ensemble performance.

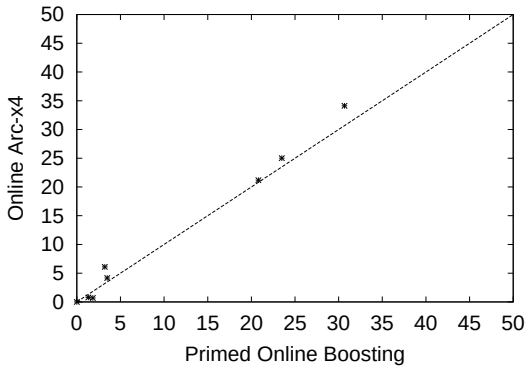

Figure 2.19: Test Error Rates: Primed Online Boosting vs. Online Arc-x4 with decision tree base models.

Chapter 3

Bagging

In this chapter, we first describe the bagging (Breiman, 1994) algorithm and some of the theory behind it. We then derive the online bagging algorithm. Finally, we compare the performances of the two algorithms theoretically and experimentally.

3.1 The Bagging Algorithm

As we explained in Chapter 2, ensemble methods perform best when they create base models that are different from one another. *B*ootstrap *Agg*regat*ing* (bagging) (Breiman, 1994), does this by drawing multiple bootstrap training sets from the original training set and using each of these to construct a base model. Because these bootstrap training sets are likely to be different, we expect the base models to have some differences. The algorithms for bagging and doing the bootstrap sampling (sampling with replacement) are shown in Figure 3.1. Figure 3.2 depicts the bagging algorithm in action. To create a bootstrap training set from the original training set of size N, we perform N multinomial trials where, in each trial, we draw one of the N examples. Each example has probability $1/N$ of being drawn in each trial. The second algorithm shown in Figure 3.1 does exactly this—N times, the algorithm chooses a number r from 1 to N and adds the rth training example to the bootstrap training set S. Clearly, some of the original training examples will not be selected for inclusion in the bootstrap training set and others will be chosen one or more times. In bagging, we create M such bootstrap training sets and then generate classifiers

Bagging(T,M)

For each $m = 1, 2, \ldots, M$,

$T_m = Sample\_With\_Replacement(T, N)$

$h_m = L_b(T_m)$

Return $h_{fin}(x) = \text{argmax}_{y \in Y} \sum_{m=1}^{M} I(h_m(x) = y)$

Sample_With_Replacement(T,N)

$S = \emptyset$

For $i = 1, 2, \ldots, N$,

$r = random\_integer(1, N)$

Add $T[r]$ to S.

Return S.

Figure 3.1: Batch Bagging Algorithm and Sampling with Replacement: T is the original training set of N examples, M is the number of base models to be learned, L_b is the base model learning algorithm, the h_i's are the classification functions that take a new example as input and return the predicted class, $random\_integer(a, b)$ is a function that returns each of the integers from a to b with equal probability, and $I(A)$ is the indicator function that returns 1 if event A is true and 0 otherwise.

using each of them. In Figure 3.2, the set of three arrows on the left (which have "Sample w/ Replacement" above them) depicts sampling with replacement three times ($M = 3$). The next set of arrows depicts calling the base model learning algorithm on these three bootstrap samples to yield three base models. Bagging returns a function $h(x)$ that classifies new examples by returning the class y out of the set of possible classes Y that gets the maximum number of votes from the base models $h_1, h_2, \ldots, h_M$. In Figure 3.2, three base models vote for the class. In bagging, the M bootstrap training sets that are created are likely to have some differences. If these differences are enough to induce some differences among the M base models while leaving their performances reasonably good, then, as described in Chapter 2, the ensemble is likely to perform better than the base models.

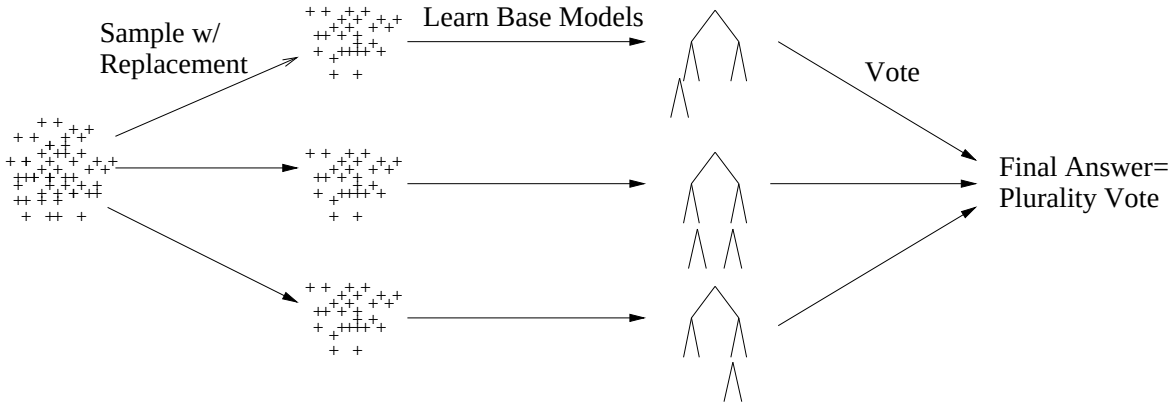

Figure 3.2: The Batch Bagging Algorithm in action. The points on the left side of the figure represent the original training set that the bagging algorithm is called with. The three arrows pointing away from the training set and pointing toward the three sets of points represent sampling with replacement. The base model learning algorithm is called on each of these samples to generate a base model (depicted as a decision tree here). The final three arrows depict what happens when a new example to be classified arrives—all three base models classify it and the class receiving the maximum number of votes is returned.

3.2 Why and When Bagging Works

It is well-known in the ensemble learning community that bagging is more helpful when the base model learning algorithm is *unstable*, i.e., when small changes in the training set lead to large changes in the hypothesis returned by the learning algorithm (Breiman, 1996a). This is consistent with what we discussed in Chapter 2: an ensemble needs to have base models that perform reasonably well but are nevertheless different from one another. Bagging is not as helpful with stable base model learning algorithms because they tend to return similar base models in spite of the differences among the bootstrap training sets. Because of this, the base models almost always vote the same way, so the ensemble returns the same prediction as almost all of its base models, leading to almost no improvement over the base models.

Breiman illustrates this as follows. Given a training set T consisting of N independent draws from some distribution D, we can apply a learning algorithm and get a predictor $h(x, T)$ that returns a predicted class given a new example x. If (X, Y) is a new example drawn from distribution D, then the probability that X is classified correctly is

$$r(T) \quad = \quad P(Y = h(X, T))$$

$$= \sum_{c=1}^{C} P(h(X,T) = c|Y = c)P(Y = c),$$

where {1,2,...,C} is the set of possible classes to which an example can belong. Let us define

$$Q(c|X) = P_{\mathbf{T}}(h(X,\mathbf{T}) = c),$$

i.e., the probability over the set $\mathbf{T}$ of randomly drawn training sets that the predictor predicts class c, then the probability that X is classified correctly, averaged over randomly drawn training sets, is

$$\begin{aligned} r &= \sum_{c=1}^{C} E(Q(c|X)|Y = c)P(Y = c) \\ &= \sum_{c=1}^{C} \int Q(c|x)P(c|x)P_X(dx) \end{aligned}$$

where $P_X(x)$ is the probability that an example X is drawn under distribution D. Breiman defines an *aggregated* classifier $h_A(x) = \text{argmax}_i\, Q(i|x)$, i.e., the predictor constructed by aggregating the predictors constructed on all the possible training sets (bagging obviously approximates this). We can write the probability of correct classification of the aggregated classifier as

$$r_A = \sum_{c=1}^{C} \int I(\underset{i}{\text{argmax}}\, Q(i|x) = c)P(c|x)P_X(dx).$$

The improvement in classification performance that we get by aggregating compared to not aggregating is

$$r_A - r = \sum_{c=1}^{C} \int [I(\underset{i}{\text{argmax}}\, Q(i|x) = c) - Q(c|x)]P(c|x)P_X(dx). \tag{3.1}$$

Equation 3.1 shows that aggregating especially helps with unstable base classifiers. If the classifiers are too stable, i.e., the classifiers induced by the various training sets tend to agree, then $Q(c|X)$ will tend to be close to 0 or 1, in which case $Q(c|X)$ is close to $I(\text{argmax}_i\, Q(i|x) = c)$ and $r_A - r$ will be quite low. With unstable base classifiers, $Q(c|X)$ will tend to be away from 0 or 1 leading to higher values of $r_A - r$, i.e., a greater benefit

from aggregation. Clearly aggregation is often impossible because we typically cannot obtain all possible training sets. Bagging approximates aggregation by drawing a relatively small number of bootstrap training sets. Nevertheless, Equation 3.1 still demonstrates that the more diverse the base models in terms of their predictions, the more bagging improves upon them in terms of classification performance.

3.3 The Online Bagging Algorithm

Bagging seems to require that the entire training set be available at all times because, for each base model, sampling with replacement is done by performing N random draws over the entire training set. However, we are able to avoid this requirement as follows. We noted earlier in this chapter that, in bagging, each original training example may be replicated zero, one, two, or more times in each bootstrap training set because the sampling is done with replacement. Each base model's bootstrap training set contains K copies of each of the original training examples where

$$P(K = k) = \binom{N}{k} \left(\frac{1}{N}\right)^k \left(1 - \frac{1}{N}\right)^{N-k} \tag{3.2}$$

which is the binomial distribution. Knowing this, instead of sampling with replacement by performing N random draws over the entire training set, one could just read the training set in order, one example at a time and draw each example a random number K times according to Equation 3.2. If one has an online base model learning algorithm then, as each training example arrives, for each of the base models, we could choose K according to Equation 3.2 and use the learning algorithm to update the base model with the new example K times. This would simulate sampling with replacement but allow us to keep just one training example in memory at any given time—the entire training set does not have to be available. However, in many online learning scenarios, we do not know N—the number of training examples—because training data continually arrives. This means that we cannot use Equation 3.2 to choose the number of draws for each training example.

However, as $N \to \infty$, which is reasonable in an online learning scenario, the distribution of K tends to a Poisson(1) distribution: $P(K = k) = \frac{exp(-1)}{k!}$. Now that we

OnlineBagging($\mathbf{h}, d$)

For each base model h_m, ($m \in \{1, 2, \ldots, M\}$) in $\mathbf{h}$,

Set k according to $Poisson(1)$.

Do k times

$h_m = L_o(h_m, d)$

Return $\mathbf{h}(x) = \text{argmax}_{y \in Y} \sum_{m=1}^{M} I(h_m(x) = y)$.

Figure 3.3: Online Bagging Algorithm: $\mathbf{h}$ is the classification function returned by online bagging, d is the latest training example to arrive, and L_o is the online base model learning algorithm.

have removed the dependence on N, we can perform online bagging as follows (see Figure 3.3): as each training example is presented to our algorithm, for each base model, choose $K \sim Poisson(1)$ and update the base model with that example K times. New examples are classified the same way as in bagging: unweighted voting over the M base models.

Online bagging is a good approximation to batch bagging to the extent that their sampling methods generate similar distributions of bootstrap training sets and their base model learning algorithms produce similar hypotheses when trained with similar distributions over training examples. We examine this issue in the next section.

3.4 Convergence of Batch and Online Bagging

In this section, we prove that, under certain conditions, the classification function returned by online bagging converges to that returned by batch bagging as the number of base models and the number of training examples tends to infinity. This means that the classification performances of the ensembles returned by batch and online bagging also converge. The overall structure of our argument is as follows. After mentioning some known definitions and theorems, we first prove that the bootstrap sampling distribution used in online bagging converges to that of batch bagging. Then we prove that the distribution over base models under online bagging converges to the analogous distribution under batch bagging subject to certain conditions on the base model learning algorithms which we discuss. We

finally establish the convergence of the classification functions returned by online and batch bagging.

3.4.1 Preliminaries

In this section, we state some standard definitions (Grimmett & Stirzaker, 1992) and prove some essential lemmas.

Definition 1 *A sequence of random variables* $X_1, X_2, \ldots$ converges in probability *to a random variable* X *(written* $X_n \xrightarrow{P} X$*) if, for all* $\epsilon > 0$, $P(|X_n - X| > \epsilon) \to 0$ *as* $n \to \infty$.

Definition 2 *A sequence of vectors* $\vec{X}_1, \vec{X}_2, \ldots$ *whose elements are random variables converges in probability to a vector* $\vec{X}$ *of random variables (written as* $\vec{X}_n \xrightarrow{P} \vec{X}$*) if the sequence of random variables* $\vec{X}_{1,i}, \vec{X}_{2,i}, \ldots$ *(the* i*th elements of* $\vec{X}_1, \vec{X}_2, \ldots$*) converges in probability to* $\vec{X}_i$ *for all* $i \in \{1, 2, \ldots, N\}$ *where* N *is the number of elements in all the vectors.*

The following is Scheffé's Theorem (Billingsley, 1995). We state the version for discrete distributions because that is all we need.

Theorem 1 *Define* $p_1, p_2, \ldots$ *to be a sequence of distributions and* p *to be another distribution such that* $lim_{n\to\infty} p_n(\theta) = p(\theta)$ *for all values* $\theta \in \Theta$*, where* Θ *is the set of support for* p_n *and* p*, i.e.,* $\sum_{\theta\in\Theta} p_n(\theta) = \sum_{\theta\in\Theta} p(\theta) = 1$ *for all* n*. Then we have*

$$\lim_{n\to\infty} \sum_{\theta\in\Theta} |p_n(\theta) - p(\theta)| = 0.$$

Corollary 1 *Given the same conditions as in Scheffe's Theorem, we have*

$$\lim_{n\to\infty} \sum_{\theta\in\Theta_s} |p_n(\theta) - p(\theta)| = 0,$$

for all $\Theta_s \subseteq \Theta$.

We now prove the convergence of the sampling distributions used in online and batch bagging. Let T denote the training set presented to the batch bagging and online bagging algorithms. Use $Multinomial(A, \frac{1}{N})$ to denote the multinomial distribution with

A trials where, in each trial, each of N possible elements is chosen with probability $\frac{1}{N}$. Sampling with replacement (bootstrapping) in the batch bagging algorithm is done by performing N independent multinomial trials where each trial yields one of the N training examples from T, each of which has probability $\frac{1}{N}$ of being drawn. This distribution is $Multinomial(N, \frac{1}{N})$. Define $\vec{\theta}_b^m$ to be a vector of length N where the ith element θ_{bi}^m represents the fraction of the trials in which the ith original training example $T(i)$ is drawn into the bootstrapped training set T_m of the mth base model when sampling with replacement. Therefore, $\vec{\theta}_b^m \sim \frac{1}{N} Multinomial(N, \frac{1}{N})$. For example, if we have five training examples ($N = 5$), then one possible value for $\vec{\theta}_b^m$ is $[\ 0.4 \quad 0 \quad 0.2 \quad 0.2 \quad 0.2\ ]$. Given these, we have $N\vec{\theta}_b^m = [\ 2 \quad 0 \quad 1 \quad 1 \quad 1\ ]$, This means that, out of the five examples in T_m, there are two copies of $T(1)$, and one copy of each of $T(3)$, $T(4)$, and $T(5)$. Example $T(2)$ was left out. Define $P_b(\vec{\theta}_b)$ to be the probability of obtaining $\vec{\theta}_b$ under batch bagging's bootstrapping scheme.

Define $\vec{\theta}_o^m$ to be the online bagging version of $\vec{\theta}_b^m$. Recall that, under online bagging, each training example is chosen a number of times according to a $Poisson(1)$ distribution. Since there are N training examples, there are N such trials; therefore, the total number of examples drawn has a $Poisson(N)$ distribution. Therefore, each element of $\vec{\theta}_o^m$ is distributed according to a $\frac{1}{N'} Poisson(1)$ distribution, where $N' \sim Poisson(N)$. For example, if we have five training examples ($N = 5$) and $\vec{\theta}_o^m = [\ 0.4 \quad 0 \quad 0.2 \quad 0.2 \quad 0.2\ ]$, then 40% of the bootstrapped training set is copies of $T(1)$, and $T(3)$, $T(4)$, and $T(5)$ make up 20% of the training set each, and $T(2)$ was left out. However N', which is the total size of the bootstrapped training set, is not fixed. Clearly, we would need to have that $N'\vec{\theta}_o^m$ is a vector of integers so that the number of times each example from T is included in the bootstrap training set is an integer. Define $P_o(\vec{\theta}_o)$ to be the probability of obtaining $\vec{\theta}_o$ under online bagging's bootstrap sampling scheme.

We now show that the bootstrap sampling method of online bagging is equivalent to performing N' multinomial trials where each trial yields one of the N training examples, each of which has probability $\frac{1}{N}$ of being drawn. Therefore $\vec{\theta}_o^m \sim \frac{1}{N'} \sum_{t=0}^{\infty} P(N' = t) Multinomial(t, \frac{1}{N})$ and each element of $\vec{\theta}_o^m$ is distributed according to $\frac{1}{N'} \sum_{t=0}^{\infty} P(N' = t) Binomial(t, \frac{1}{N})$. Note that this is the same as the bootstrap sampling distribution for batch bagging except that the number of multinomial trials is not fixed. This lemma makes

our subsequent proofs easier.

Lemma 1 *$X \sim Poisson(1)$ if and only if $X \sim \sum_{t=0}^{\infty} P(N' = t) Binomial(t, \frac{1}{N})$.*

Proof: We prove this by showing that the probability generating functions (Grimmett & Stirzaker, 1992) for the two distributions are the same. The probability generating function for a $Poisson(\lambda)$ random variable is $G_{Poisson(\lambda)}(s) = exp(\lambda(s-1))$. The probability generating function for a $Binomial(t, p)$ random variable is $G_{Bin(t,p)}(s) = ((1-p)+ps)^t$. As mentioned above, the distribution $\sum_{t=0}^{\infty} P(N' = t) Binomial(t, \frac{1}{N})$ involves performing $N' = t$ Bernoulli trials where $N' \sim Poisson(N)$. By standard results (Grimmett & Stirzaker, 1992), we can obtain the generating function for this distribution by composition:

$$\begin{aligned} G_{Poisson(N)}(G_{Bin\left(1,\frac{1}{N}\right)}(s)) &= exp\left(N\left(\left(1-\frac{1}{N}\right)+\frac{1}{N}s-1\right)\right) \\ &= exp(s-1), \end{aligned}$$

but this is the generating function for a $Poisson(1)$ random variable, which is what we wanted to show. ∎

We next show that the distributions of the bootstrapped training set proportion vectors $\vec{\theta}$ under online and batch bagging converge as the number of training sets M (corresponding to the number of base models) increases and as the number of training examples N in the original training set increases. Specifically, these distributions converge in probability. Define $\theta_{\mathbf{b}}^{\mathbf{M}} = \frac{1}{M}\sum_{m=1}^{M} \vec{\theta}_b^m$ and $\theta_{\mathbf{o}}^{\mathbf{M}} = \frac{1}{M}\sum_{m=1}^{M} \vec{\theta}_o^m$, which are the averages of the bootstrap distribution vectors over the training set for M base models. The elements of these average vectors are $\theta_{\mathbf{b}n}^{\mathbf{M}} = \frac{1}{M}\sum_{m=1}^{M} \theta_{bn}^m$ and $\theta_{\mathbf{o}n}^{\mathbf{M}} = \frac{1}{M}\sum_{m=1}^{M} \theta_{on}^m$.

Lemma 2 *As $M \to \infty$ and/or $N \to \infty$, $\theta_{\mathbf{o}}^{\mathbf{M}} \xrightarrow{P} \theta_{\mathbf{b}}^{\mathbf{M}}$.*

Proof: In the batch version of bagging, to generate each base model's bootstrapped training set, we perform N independent trials where the probability of drawing each example is $\frac{1}{N}$. We define the following indicator variables, for $m \in \{1, 2, \ldots, M\}$, $n \in \{1, 2, \ldots, N\}$ and $j \in \{1, 2, \ldots, N\}$,

$$X_{mnj} = \begin{cases} 1 & \text{if example } n \text{ is drawn on the } j\text{th trial for the } m\text{th model} \\ 0 & \text{otherwise} \end{cases}$$

Clearly, $P(X_{mnj} = 1) = \frac{1}{N}$ for all m, n, and j. The fraction of the mth base model's bootstrapped training set that consists of draws of example number n is $Y_{mn} = \frac{1}{N}\sum_{j=1}^{N} X_{mnj}$. Therefore, we have

$$\begin{aligned} E\left(\frac{1}{N}\sum_{j=1}^{N} X_{mnj}\right) &= \frac{1}{N} \\ var\left(\frac{1}{N}\sum_{j=1}^{N} X_{mnj}\right) &= \frac{1}{N} var\left(X_{mnj}\right) \\ &= \frac{1}{N^2}\left(1 - \frac{1}{N}\right). \end{aligned}$$

Our bagged ensemble consists of M base models, so we do the above bootstrapping process M times. Over M models, the average fraction of the bootstrapped training set that consists of draws of example number n is

$$\theta_{\mathbf{b}n}^{\mathbf{M}} = \frac{1}{M}\sum_{m=1}^{M} Y_{mn}.$$

We have

$$\begin{aligned} E(\theta_{\mathbf{b}n}^{\mathbf{M}}) &= E\left(\frac{1}{M}\sum_{m=1}^{M} Y_{mn}\right) = \frac{1}{N} \\ var\left(\theta_{\mathbf{b}n}^{\mathbf{M}}\right) &= var\left(\frac{1}{M}\sum_{m=1}^{M} Y_{mn}\right) = \frac{1}{MN^2}\left(1 - \frac{1}{N}\right). \end{aligned}$$

Therefore, by the Weak Law of Large Numbers, as $M \to \infty$ or $N \to \infty$, $\theta_{\mathbf{b}n}^{\mathbf{M}} \xrightarrow{P} \frac{1}{N}$; therefore, $\theta_{\mathbf{b}}^{\mathbf{M}} \xrightarrow{P} \frac{1}{N}\mathbf{1_N}$, where $\mathbf{1_N}$ is a vector of length N where every element is 1.

Now, we show that, as $M \to \infty$ or $N \to \infty$, $\theta_{\mathbf{o}}^{\mathbf{M}} \xrightarrow{P} \frac{1}{N}\mathbf{1_N}$, which implies that $\theta_{\mathbf{o}}^{\mathbf{M}} \xrightarrow{P} \theta_{\mathbf{b}}^{\mathbf{M}}$ (Grimmett & Stirzaker, 1992).

As mentioned earlier, in online bagging, we can recast the bootstrap sampling process as performing N' independent multinomial trials where the probability of drawing each training example is $\frac{1}{N}$ and $N' \sim Poisson(N)$.

For online bagging, let us define X_{mnj} the same way that we did for batch bagging except that $j \in \{1, 2, \ldots, N'\}$. Clearly, $P(X_{mnj} = 1) = \frac{1}{N}$ for all m, n, and j. The

fraction of the mth base model's bootstrapped training set that consists of draws of example number n is $Y_{mn} = \frac{1}{N'} \sum_{j=1}^{N'} X_{mnj}$. Therefore, we have

$$\begin{aligned} E\left(\frac{1}{N'}\sum_{j=1}^{N'} X_{mnj}\right) &= \sum_{n=0}^{\infty} P(N'=n) E\left(\frac{1}{N'}\sum_{j=1}^{N'} X_{mnj} \mid N'=n\right) \\ &= \sum_{n=0}^{\infty} P(N'=n)\frac{1}{n} nE(X_{mnj}) \\ &= \frac{1}{N}\sum_{n=0}^{\infty} P(N'=n) \\ &= \frac{1}{N} \end{aligned}$$

where, for $n = 0$, we are defining $E(\frac{1}{n}\sum_{j=1}^{n} X_{mnj}|n=0) = \frac{1}{N}$. This is done merely for convenience in this derivation—one can define this to be any value from 0 to 1 and it would not matter in the long run since $P(N'=0) \to 0$ as $N \to \infty$.

We also have, by standard results (Grimmett & Stirzaker, 1992),

$$var\left(\frac{1}{N'}\sum_{j=1}^{N'} X_{mnj}\right) = E\left(var\left(\frac{1}{N'}\sum_{j=1}^{N'} X_{mnj} \mid N'\right)\right) + var\left(E\left(\frac{1}{N'}\sum_{j=1}^{N'} X_{mnj} \mid N'\right)\right).$$

Let us look at the second term first. Since $E\left(\frac{1}{N'}\sum_{j=1}^{N'} X_{mnj}|N'\right) = \frac{1}{N}$, the second term is just the variance of a constant which is 0. So we only have to worry about the first term.

$$\begin{aligned} E\left(var\left(\frac{1}{N'}\sum_{j=1}^{N'} X_{mnj}|N'\right)\right) &= \sum_{n=0}^{\infty} var\left(\frac{1}{N'}\sum_{j=1}^{N'} X_{mnj}|N'=n\right) P(N'=n) \\ &= \sum_{n=0}^{\infty} P(N'=n)\frac{1}{n} var(X_{mnj}). \end{aligned}$$

Clearly, we would want $var(\frac{1}{N'}\sum_{j=1}^{N'} X_{mnj}|N'=0) = 0$, because with $N'=0$, there would be no multinomial trials, so $X_{mnj} = 0$ and the variance of a constant is 0.

Continuing the above derivation, we have

$$\begin{aligned} \sum_{n=1}^{\infty} P(N'=n)\frac{1}{n^2} nvar(X_{mnj}) &= \sum_{n=1}^{\infty} P(N'=n)\frac{1}{n}\frac{1}{N}\left(1-\frac{1}{N}\right) \\ &= \frac{1}{N}\left(1-\frac{1}{N}\right)\sum_{n=1}^{\infty}\frac{1}{n}P(N'=n) \\ &\leq \frac{1}{N}\left(1-\frac{1}{N}\right). \end{aligned}$$

So we have

$$var\left(\frac{1}{N'}\sum_{j=1}^{N'} X_{mnj}\right) \leq \frac{1}{N}\left(1-\frac{1}{N}\right).$$

We have M base models, so we repeat the above bootstrap process M times. Over M base models, the average fraction of the bootstrapped training set consisting of draws of example n is

$$\theta^{\mathbf{M}}_{\mathbf{o}n} = \frac{1}{M}\sum_{m=1}^{M} Y_{mn}.$$

We have

$$E(\theta^{\mathbf{M}}_{\mathbf{o}n}) = E\left(\frac{1}{M}\sum_{m=1}^{M} Y_{mn}\right) = \frac{1}{N}$$
$$var(\theta^{\mathbf{M}}_{\mathbf{o}n}) = \frac{1}{M^2} M var(Y_{mn}) \leq \frac{1}{M}\frac{1}{N}\left(1-\frac{1}{N}\right).$$

Therefore, by the Weak Law of Large Numbers, as $M \to \infty$ or $N \to \infty$, $\theta^{\mathbf{M}}_{\mathbf{o}n} \xrightarrow{P} \frac{1}{N}$, which means that $\theta^{\mathbf{M}}_{\mathbf{o}} \xrightarrow{P} \frac{1}{N}\mathbf{1_N}$. As mentioned earlier, this implies that $\theta^{\mathbf{M}}_{\mathbf{o}} \xrightarrow{P} \theta^{\mathbf{M}}_{\mathbf{b}}$. ∎

Now that we have established the convergence of the sampling distributions, we go on to demonstrate the convergence of the bagged ensembles themselves.

3.4.2 Main Result

Let T be a training set with N examples and let $\vec{\theta}$ be a vector of length N where $\sum_{i=1}^{N} \vec{\theta}_i = 1$ and $0 \leq \vec{\theta}_i \leq 1$ for all $i \in \{1, 2, \ldots, N\}$. Define $h_b(x, \vec{\theta}, T)$ to be a function that classifies a new example x as follows. First, draw a bootstrap training set S from T of the same size as T in which, for all i, $N\vec{\theta}_i$ copies of the ith example in T are placed in S. The second step is to create a hypothesis by calling a batch learning algorithm on S. In our proof of convergence, we use h_b's to represent the base models in the batch bagging algorithm. Define $h_o(x, \vec{\theta}, s, T)$ to be the analogous function returned by an online learning

algorithm.[1] We use h_o's to represent the base models in our online bagging algorithm. Recall that the size of the bootstrap training set is not fixed in online bagging. For this reason, we include an additional input s, which is the size of the bootstrap training set. Recall that $s \sim Poisson(N)$. Clearly, $s\vec{\theta}$ must be a vector of integers because the number of copies of each example from T included in the bootstrap training set must be an integer.

Define $h_b^M(x,T) = \text{argmax}_{y \in Y} \sum_{m=1}^{M} I(h_b(x, \vec{\theta}_b^m, T) = y)$, which is the classification function returned by the batch bagging algorithm when asked to return an ensemble of M base models given a training set T. Define $h_o^M(x,T) = \text{argmax}_{y \in Y} \sum_{m=1}^{M} I(h_o(x, \vec{\theta}_o^m, s, T) = y)$, which is the analogous function returned by online bagging. The distributions over $\vec{\theta}_b$ and $\vec{\theta}_o$ induce distributions over the base models $P_b(h_b(x, \vec{\theta}_b, T))$ and $P_o(h_o(x, \vec{\theta}_o, s, T))$. In order to show that $h_o^M(x,T) \rightarrow h_b^M(x,T)$ (i.e., that the ensemble returned by online bagging converges to that returned by batch bagging), we need to have $P_o(h_o(x, \vec{\theta}_o, s, T)) \rightarrow P_b(h_b(x, \vec{\theta}_b, T))$ as $M \rightarrow \infty$ and $N \rightarrow \infty$. However, this is clearly not true for all batch and online base model learning algorithms because there are bootstrap training sets that online bagging may produce that batch bagging cannot produce. In particular, the bootstrap training sets produced by batch bagging are always of the same size N as the original training set T. This is not true of online bagging—in fact, as $N \rightarrow \infty$, the probability that the bootstrap training set is of size N tends to 0. Therefore, suppose the base model learning algorithms return some null hypothesis h_0 if the bootstrap training set does not have exactly N examples. In this case, as $N \rightarrow \infty$, $P_o(h_0(x)) \rightarrow 1$, i.e., under online bagging, the probability of getting the null hypothesis for a base model tends to 1. However, $P_b(h_0(x)) = 0$. In this case, clearly $h_o^M(x,T)$ does not converge to $h_b^M(x,T)$.

For our proof of convergence, we require that the batch and online base model learning algorithms be *proportional*.

Definition 3 *Let $\vec{\theta}$, $h_b(x, \vec{\theta}, T)$, and $h_o(x, \vec{\theta}, s, T)$ be as defined above. If $h_o(x, \vec{\theta}, s, T) = h_b(x, \vec{\theta}, T)$ for all $\vec{\theta}$ and s, then we say that the batch algorithm that produced h_b and the online algorithm that produced h_o are* proportional *learning algorithms.*

This clearly means that our online bagging algorithm is assumed to use an online base

[1] Online learning algorithms do not need to be called with the entire training set at once. We just notate it this way for convenience and because, to make the proofs easier, we recast the online bagging algorithm's online sampling process as an offline sampling process in our first lemma.

model learning algorithm that is lossless relative to the base model learning algorithm used in batch bagging. However, our assumption is actually somewhat stronger. We require that our base model learning algorithms return the same hypothesis given the same T and θ. In particular, we assume that the size s of the bootstrapped training set does not matter—only the proportions of every training example relative to every other training example matter. For example, if we were to create a new bootstrapped training set T_n by repeating each example in the current bootstrapped training set T_c, then note that $\vec{\theta}$ would be the same for both T_n and T_c and, of course, the original training set T would be the same. We assume that our base model learning algorithms would return the same hypothesis if called with T_n as they would if called with T_c. This assumption is true for decision trees, decision stumps, and Naive Bayes classifiers because they only depend on the relative proportions of training examples having different attribute and class values. However, this assumption is not true for neural networks and other models generated using gradient-descent learning algorithms. For example, training with T_n would give us twice as many gradient-descent steps as training with T_c, so we would not expect to get the same hypothesis in these two cases.

One may worry that it is possible to get values for $\vec{\theta}_o$ that one cannot get for $\vec{\theta}_b$. In particular, all bootstrap training sets drawn under batch bagging are of size N, so for all possible $\vec{\theta}_b$, $N\vec{\theta}_b$ is a vector of integers. However, this is not true for all possible $\vec{\theta}_o$. For example, if online bagging creates a bootstrap training set of size $N+1$, then $(N+1)\vec{\theta}_o$ would be a vector of integers. If $N\vec{\theta}_o$ is not a vector of integers, then clearly batch bagging cannot draw $\vec{\theta}_o$. That is, $P_b(\vec{\theta}_o) = 0$ while $P_o(\vec{\theta}_o) > 0$. Define Θ_b and Θ_o to be the set of possible values of $\vec{\theta}_b$ and $\vec{\theta}_o$, respectively and $\Theta = \Theta_b \cup \Theta_o$. Define $\Theta_s = \{\vec{\theta} \in \Theta : P_b(\vec{\theta}) = 0\}$, i.e., the set of $\vec{\theta}$ that can be obtained under online bagging but not under batch bagging. We might be worried if our base model learning algorithms return some null hypothesis for $\vec{\theta} \in \Theta_s$. We can see why as follows. We have, for all $y \in Y$,

$$
\begin{aligned}
P(h_o^M(x) = y) &\rightarrow \sum_{\vec{\theta} \in \Theta} P_o(\vec{\theta}) I(h_o(x, \vec{\theta}, s, T) = y) \\
P(h_b^M(x) = y) &\rightarrow \sum_{\vec{\theta} \in \Theta} P_b(\vec{\theta}) I(h_b(x, \vec{\theta}, T) = y)
\end{aligned}
$$

as $M \rightarrow \infty$. We can rewrite these as follows:

$$P(h_o^M(x) = y) \rightarrow \sum_{\vec{\theta} \in \Theta - \Theta_s} P_o(\vec{\theta}) I(h_o(x, \vec{\theta}, s, T) = y) + \sum_{\vec{\theta} \in \Theta_s} P_o(\vec{\theta}) I(h_o(x, \vec{\theta}, s, T) = y)$$

$$P(h_b^M(x) = y) \rightarrow \sum_{\vec{\theta} \in \Theta - \Theta_s} P_b(\vec{\theta}) I(h_b(x, \vec{\theta}, T) = y).$$

If our base model learning algorithms return some null hypothesis for $\vec{\theta} \in \Theta_s$, then the second term in the equation for $P(h_o^M(x) = y)$ may prevent convergence of $P(h_o^M(x) = y)$ and $P(h_b^M(x) = y)$. We clearly require some smoothness condition whereby small changes in $\vec{\theta}$ do not yield dramatic changes in the prediction performance. It is generally true that since $\vec{\theta}_{\mathbf{o}}^{\mathbf{M}} \xrightarrow{P} \vec{\theta}_{\mathbf{b}}^{\mathbf{M}}$, $f(\vec{\theta_o}) \xrightarrow{P} f(\vec{\theta_b})$ if f is a continuous function. Our classification functions clearly have discontinuities because they return a class which is discrete-valued. However, given Lemma 4, we only require that our classification functions $h_b(x, \vec{\theta}, T)$ and $h_o(x, \vec{\theta}, s, T)$ converge in probability to some classifier $L(x, \theta, T)$ as $N \rightarrow \infty$. Of course, obtaining such convergence requires that $L(x, \theta, T)$ be bounded away from a decision boundary. That is, for every $\epsilon > 0$, there must exist an N_o such that for all $N > N_o$, $L(x, \vec{\theta}, T) = L(x, \theta, T)$ for all $\vec{\theta}$ in an ϵ-neighborhood around θ. This requirement is clearly related to the notion of stability that we discussed in Section 3.2. Decision trees and neural networks are unstable while Naive Bayes classifiers and decision stumps are stable; therefore, small changes in $\vec{\theta}$ are more likely to cross decision boundaries in case of decision trees and neural networks than in case of Naive Bayes classifiers and decision stumps; therefore, we can expect convergence of online bagging to batch bagging to be slower with unstable base models than with stable ones.

Theorem 2 *If $h_b(x, \vec{\theta}, T) = h_o(x, \vec{\theta}, s, T)$ for all $\vec{\theta}$ and s and if $h_b(x, \vec{\theta}, T)$ and $h_o(x, \vec{\theta}, s, T)$ converge in probability to some classifier $L(x, \theta, T)$ as $N \rightarrow \infty$, then $h_o^M(x, T) \rightarrow h_b^M(x, T)$ as $M \rightarrow \infty$ and $N \rightarrow \infty$ for all x.*

Proof: Let us define $h(x, \vec{\theta}, T) = h_b(x, \vec{\theta}, T) = h_o(x, \vec{\theta}, s, T)$. Let $h(x, \vec{\theta}_{\mathbf{b}}^{\mathbf{M}}, T)$ and $h(x, \vec{\theta}_{\mathbf{o}}^{\mathbf{M}}, T)$ denote the distributions over base models under batch and online bagging, respectively. Clearly, $h(x, \vec{\theta}_{\mathbf{o}}^{\mathbf{M}}, T) \xrightarrow{P} h(x, \vec{\theta}_{\mathbf{b}}^{\mathbf{M}}, T)$. Since $h_b^M(x, T)$ and $h_o^M(x, T)$ are created using M draws from $h(x, \vec{\theta}_{\mathbf{b}}^{\mathbf{M}}, T)$ and $h(x, \vec{\theta}_{\mathbf{o}}^{\mathbf{M}}, T)$, which are distributions that converge in

probability, we immediately get $h_o^M(x,T) \xrightarrow{P} h_b^M(x,T)$. ∎

To summarize, we have proven that the classification function of online bagging converges to that of batch bagging as the number of base models M and the number of training examples N tend to infinity if the base model learning algorithms are proportional and if the base models themselves converge to the same classifier as $N \to \infty$. We noted that decision trees, decision stumps, and Naive Bayes classifiers are proportional, but gradient-descent learning algorithms such as backpropagation for neural networks typically are not proportional. Base model convergence and therefore the convergence of online bagging to batch bagging would tend to be slower with unstable base models such as decision trees and neural networks than with Naive Bayes and decision stumps. This is clearly related to the notion of stability that is important to the performance of bagging.

We now compare online bagging and batch bagging experimentally.

3.5 Experimental Results

We now discuss the results of our experiments that compare the performances of bagging, online bagging, and the base model learning algorithms. We used four different types of base models in both batch and online bagging: decision trees, decision stumps, Naive Bayes classifiers, and neural networks. For decision trees, we reimplemented the ITI learning algorithm (Utgoff et al., 1997) in C++. ITI allows decision trees to be learned in batch and online mode. The online algorithm is lossless. The batch and online Naive Bayes learning algorithms are essentially identical, so the online algorithm is clearly lossless. We implemented both batch and lossless online learning algorithms for decision stumps. As we mentioned before, the learning algorithms for decision trees, decision stumps, and Naive Bayes classifiers are also proportional. For neural networks, we implemented the backpropagation algorithm. In the batch ensemble algorithms, we ran neural network learning for ten epochs. In the online ensemble algorithms, we can run through the training set only once; however, we can vary the number of update steps per example. We present results with one update step and ten update steps per training example. We get worse results with

these online methods than with the multi-epoch batch method, and we will see how much loss this leads to in our ensemble algorithms. We ran all of our ensemble algorithms to generate ensembles of 100 base models. We ran all of our experiments on Dell 6350 computers having 600 MegaHertz Pentium III processors.

3.5.1 The Data

We tested our algorithms on nine UCI datasets (Blake et al., 1999), two datasets (Census Income and Forest Covertype) from the UCI KDD archive (Bay, 1999), and three synthetic datasets. These are batch datasets, i.e., there is no natural order in the data. With these datasets, we use our learning algorithms to generate a hypothesis using a training set and then test the hypothesis using a separate test set. We also performed experiments with an online dataset, in that the data arrives as a sequence and the learning algorithm is expected to generate a prediction for each example immediately upon arrival. The algorithm is then given the correct answer which is used to incrementally update the current hypothesis. We describe this set of experiments in Section 3.5.4. We give the sizes and numbers of attributes and classes of the batch datasets in Table 3.1. Every dataset except Census Income was supplied as just one dataset, so we tested the batch algorithms by performing ten runs of five-fold cross validation on all the datasets except Census Income, for which we just ran with the supplied training and test set. In general n-fold cross validation consists of randomly dividing the data into n nearly equal sized groups of examples and running the learning algorithm n times such that, on the ith run, the ith group is used as the test set and the remaining groups are combined to form the training set. After learning with the training set, the accuracy on the ith test set is recorded. This process yields n results. Therefore, our ten runs of five-fold cross validation generated 50 results which we averaged to get the results for our batch algorithms. Because online algorithms are often sensitive to the order in which training examples are supplied, for each training set that we generated, we ran our online algorithms with five different orders of the examples in that training set. Therefore, the online algorithms' results are averages over 250 runs.

All three of our synthetic datasets have two classes. The prior probability of each class is 0.5, and every attribute except the last one is conditionally dependent upon the class and

Table 3.1: The datasets used in our experiments. For the Census Income dataset, we have given the sizes of the supplied training and test sets. For the remaining datasets, we have given the sizes of the training and test sets in our five-fold cross-validation runs.

Data Set	Training Set	Test Set	Inputs	Classes
Promoters	84	22	57	2
Balance	500	125	4	3
Breast Cancer	559	140	9	2
German Credit	800	200	20	2
Car Evaluation	1382	346	6	4
Chess	2556	640	36	2
Mushroom	6499	1625	22	2
Nursery	10368	2592	8	5
Connect4	54045	13512	42	3
Synthetic-1	80000	20000	20	2
Synthetic-2	80000	20000	20	2
Synthetic-3	80000	20000	20	2
Census Income	199523	99762	39	2
Forest Covertype	464809	116203	54	7

Table 3.2: $P(A_a = 0|A_{a+1}, C)$ for $a \in \{1, 2, \ldots, 19\}$ in Synthetic Datasets

$P(A_a = 0)$	$A_{a+1} = 0$	$A_{a+1} = 1$
$C = 0$	0.8	0.2
$C = 1$	0.9	0.1

the next attribute. We set up the attributes this way because the Naive Bayes model only represents the probabilities of each attribute given the class, and we wanted data that is not realizable by a single Naive Bayes classifier so that bagging and boosting are more likely to yield improvements over a single classifier. The probabilities of each attribute except the last one (A_a for $a \in \{1, 2, \ldots, 19\}$) are as shown in Table 3.2. Note that each attribute A_a depends quite strongly on A_{a+1}.

The only difference between the three synthetic datasets is $P(A_{20}|C)$. In Synthetic-1, $P(A_{20} = 0|C = 0) = 0.495$ and $P(A_{20} = 0|C = 1) = 0.505$. In Synthetic-2, these probabilities are 0.1 and 0.8, while in Synthetic-3, these are 0.01 and 0.975, respectively. These

Table 3.3: Results (fraction correct): batch and online bagging (with Decision Trees)

Dataset	Decision Tree	Bagging	Online Bagging
Promoters	0.7837	**0.8504**	**0.8613**
Balance	0.7664	**0.8161**	**0.8160**
Breast Cancer	0.9531	**0.9653**	**0.9646**
German Credit	0.6929	**0.7445**	**0.7421**
Car Evaluation	0.9537	**0.9673**	**0.9679**
Chess	0.9912	**0.9938**	**0.9936**
Mushroom	1.0	1.0	1.0
Nursery	0.9896	**0.9972**	**0.9973**

differences lead to varying difficulties for learning Naive Bayes classifiers and; therefore, different performances of single classifiers and ensembles. We tested our algorithms with all four base models on these synthetic datasets even though they were designed with Naive Bayes classifiers in mind.

3.5.2 Accuracy Results

Tables 3.3, 3.4, 3.5, 3.6, and 3.7 show the accuracies of the single model, bagging, and online bagging with decision trees, Naive Bayes, decision stumps, neural networks with one update step per training example, and neural networks with ten updates per training example, respectively. In case of decision trees, Naive Bayes, and decision stumps, the batch and online single model results are the same because the online learning algorithms for these models are lossless. Therefore, we only give one column of single model results. In case of neural networks, we show the batch and online single neural network results separately because online neural network learning is lossy. Boldface entries represent cases when the ensemble algorithm significantly (t-test, $\alpha = 0.05$) outperformed a single model while italicized entries represent cases when the ensemble algorithm significantly underperformed relative to a single model. Entries for a batch or online algorithm that have a "*" next to them represent cases for which it significantly outperformed its online or batch counterpart, respectively.

The results of running decision tree learning and batch and online bagging with de-

Table 3.4: Results (fraction correct): batch and online bagging (with Naive Bayes)

Dataset	Naive Bayes	Bagging	Online Bagging
Promoters	0.8774	*0.8354*	*0.8401*
Balance	0.9075	0.9067	0.9072
Breast Cancer	0.9647	**0.9665**	**0.9661**
German Credit	0.7483	0.748	0.7483
Car Evaluation	0.8569	0.8532	0.8547
Chess	0.8757	0.8759*	*0.8749*
Mushroom	0.9966	0.9966	0.9966
Nursery	0.9031	0.9029	*0.9027*
Connect4	0.7214	0.7212	**0.7216**
Synthetic-1	0.4998	0.4996	0.4997
Synthetic-2	0.7800	0.7801	0.7800
Synthetic-3	0.9251	0.9251	0.9251
Census-Income	0.7630	**0.7637**	**0.7636**
Forest Covertype	0.6761	0.6762	0.6762

cision tree base models are shown in Table 3.3. We ran decision tree learning and the ensemble algorithms with decision tree base models on only the eight smallest datasets in our collection because the ITI algorithm is too expensive to use with larger datasets in online mode.[2] Batch bagging and online bagging performed comparably to each other. This can be seen in Figure 3.4, a scatterplot of the error rates of batch and online boosting on the test sets—each point represents one dataset. Points above the diagonal line represent datasets for which the error of online bagging was higher than that of batch bagging and points below the line represent datasets for which online bagging had lower error. Batch and online bagging significantly outperformed decision trees on all except the Mushroom dataset, which is clearly so easy to learn that one decision tree was able to achieve perfect classification performance on the test set, so there was no room for improvement.

Table 3.4 gives the results of running Naive Bayes classifiers and batch and online

[2]This is because, as explained in Chapter 2, when a decision tree is updated online, the tests at each node of the decision tree have to be checked to confirm that they are still the best tests to use at those nodes. If any tests have to be changed then the subtrees below that node may have to be changed. This requires running through the appropriate training examples again since they have to be assigned to different nodes in the decision tree. Therefore the decision trees must store their training examples, which is clearly impractical when the training set is large.

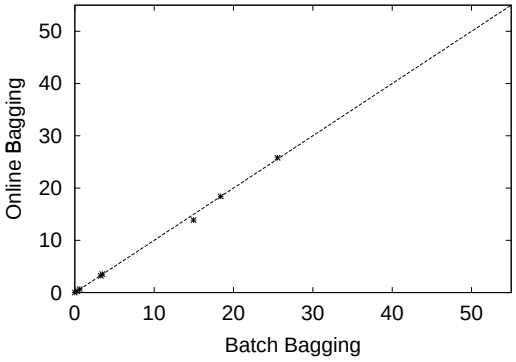

Figure 3.4: Test Error Rates: Batch Bagging vs. Online Bagging with decision tree base models.

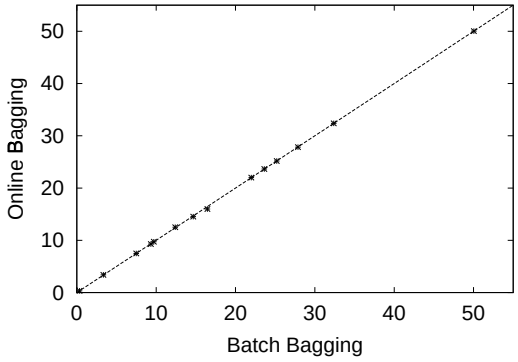

Figure 3.5: Test Error Rates: Batch Bagging vs. Online Bagging with Naive Bayes base models.

bagging with Naive Bayes base models. The bagging algorithms performed comparably to each other (Figure 3.5) and mostly performed comparably to the single models. We expected this because of the *stability* of Naive Bayes, which we discussed in Chapter 2.

Table 3.5 shows the results of running decision stumps and bagging and online bagging with decision stumps. Bagging and online bagging only significantly outperformed decision stumps for the two smallest datasets—for the remaining datasets, the bagging algorithms performed comparably to decision stumps. From both the table and Figure 3.6, we can see that batch and online bagging with decision stumps performed comparably to each other. Just as with Naive Bayes classifier learning, decision stump learning is known to be stable (Breiman, 1996a), so once again we do not have enough diversity among the base models to yield significantly better performance for bagging and online bagging.

Table 3.6 shows the results of running neural network learning and bagging and online bagging with neural network base models. For the online results in this table, the neural networks were trained using only one update step per training example. Recall that, with our batch algorithms, we trained each neural network by cycling through the dataset 10 times (epochs). We can see from both the table and Figure 3.7 that online neural network learning often performed much worse than batch neural network learning; therefore, it is not surprising that online bagging often performed much worse than batch bagging (Figure 3.8), especially on the smaller datasets. Unfortunately, online bagging also did not

Table 3.5: Results (fraction correct): batch and online bagging (with decision stumps)

Dataset	Decision Stump	Bagging	Online Bagging
Promoters	0.7710	**0.8041**	**0.8113**
Balance	0.5989	**0.7170**	**0.7226**
Breast Cancer	0.8566	0.8557	0.8564
German Credit	0.6862	0.6861	0.6862
Car Evaluation	0.6986	0.6986	0.6986
Chess	0.6795	0.6798	0.6792
Mushroom	0.5617	0.5617	0.5617
Nursery	0.4184	0.4185	0.4177
Connect4	0.6581	0.6581	0.6581
Synthetic-1	0.5002	0.4996	0.4994
Synthetic-2	0.8492	0.8492	0.8492
Synthetic-3	0.9824	0.9824	0.9824
Census-Income	0.9380	0.9380	0.9380
Forest Covertype	0.6698	0.6698	0.6698

improve upon online neural networks to the extent that batch bagging improved upon batch neural networks. This difficulty is remedied for the case of the online algorithms where the neural networks were trained with ten update steps per training example. As shown in Table 3.7, online bagging performed significantly better than online neural networks most of the time. The batch algorithms continued to perform significantly better than the online algorithms with 10 updates per example (Figure 3.9 and Figure 3.10), but not to as large an extent as in the one update case.

In summary, with base models for which we have proportional learning algorithms (decision tree, decision stump, Naive Bayes), online bagging achieved comparable classification performance to batch bagging. As expected, both bagging algorithms improved significantly upon decision trees because decision tree learning is unstable, and did not improve significantly upon Naive Bayes and decision stumps which are stable learning algorithms. With neural networks, the poorer performances of the online base models led to poorer performance of the online bagging algorithm. However, especially in larger datasets, the loss in performance due to online neural network learning was low, which led to low loss in the performance of online bagging. This small performance reduction may be ac-

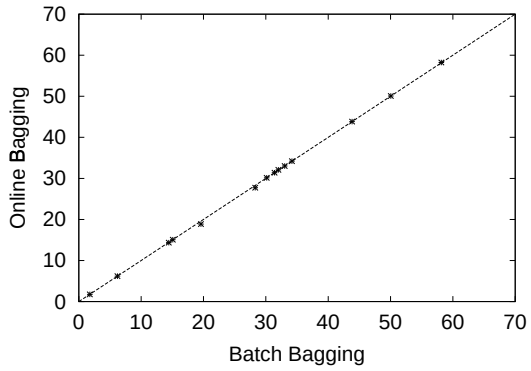

Figure 3.6: Test Error Rates: Batch Bagging vs. Online Bagging with decision stump base models.

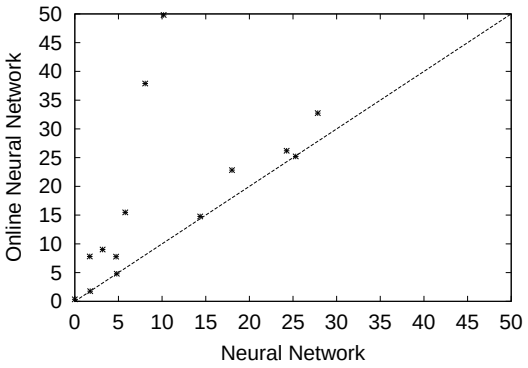

Figure 3.7: Test Error Rates: Batch Neural Network vs. Online Neural Network (one update per example).

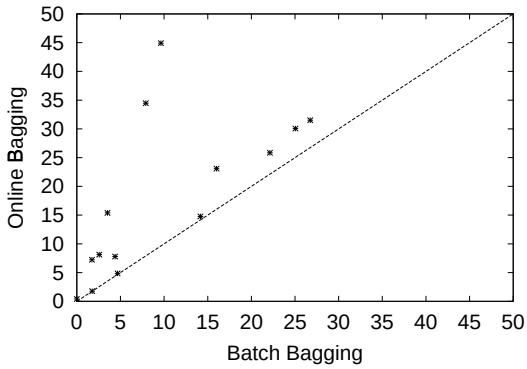

Figure 3.8: Test Error Rates: Batch Bagging vs. Online Bagging with neural network base models.

ceptable given the great reduction in running time. We examine the running times of the algorithms in the next section.

3.5.3 Running Times

The running times of all the algorithms on all the datasets explained in the previous section are shown in Tables 3.8, 3.9, 3.10, and 3.11. The running times with decision trees, decision stumps, and Naive Bayes classifiers are generally comparable and quite low relative to the running times for boosting that we will examine in the next chapter.

Table 3.6: Results (fraction correct): batch and online bagging (with neural networks). The column "Neural Network" gives the average test set performance of running backpropagation with 10 epochs on the entire training set. "Online Neural Net" is the result of running backpropagation with one update step per training example.

Dataset	Neural Network	Bagging	Online Neural Net	Online Bagging
Promoters	0.8982*	0.9036*	0.5018	0.5509
Balance	0.9194*	**0.9210***	0.6210	**0.6554**
Breast Cancer	0.9527*	**0.9561***	0.9223	0.9221
German Credit	0.7469	0.7495*	0.7479	*0.6994*
Car Evaluation	0.9422*	**0.9648***	0.8452	0.8461
Chess	0.9681*	**0.9827***	0.9098	**0.9277**
Mushroom	1*	1*	0.9965	0.9959
Nursery	0.9829*	*0.9743**	0.9220	*0.9188*
Connect4	0.8199*	**0.8399***	0.7717	*0.7691*
Synthetic-1	0.7217*	**0.7326***	0.6726	**0.6850**
Synthetic-2	0.8564*	**0.8584***	0.8523	0.8524
Synthetic-3	0.9824	0.9824	0.9824	0.9824
Census-Income	0.9519	**0.9533***	0.9520	0.9514
Forest Covertype	0.7573*	**0.7787***	0.7381	**0.7416**

The running times of online bagging with ten-update neural networks are lower than the running times of batch bagging largely because each neural network in batch bagging runs through the training set 10 times, so bagging overall runs through the training set 1000 times, whereas online bagging only runs through the training set once. This makes an especially big difference on larger datasets where fewer passes through the training set means less data being swapped between the computer's cache, main memory, and virtual memory. Not surprisingly, online bagging with one update per neural network ran much faster than batch bagging because online bagging not only passed through the dataset fewer times than batch bagging, but had substantially fewer backpropagation updates per training example (one update per base model) than batch bagging (ten updates per base model).

Clearly, the main advantage that online bagging has over batch bagging is the same as the advantage that online learning algorithms generally have over batch learning algorithms—the ability to incrementally update their hypotheses with new training examples. Given a

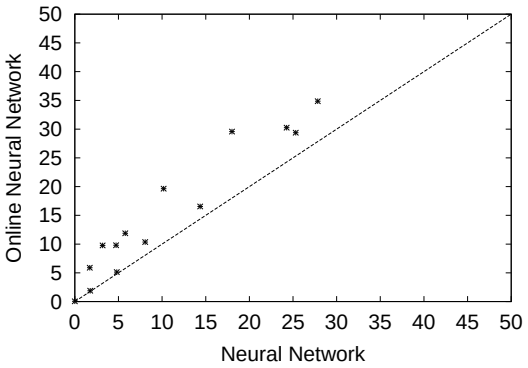

Figure 3.9: Test Error Rates: Batch Neural Network vs. Online Neural Network (10 updates per example).

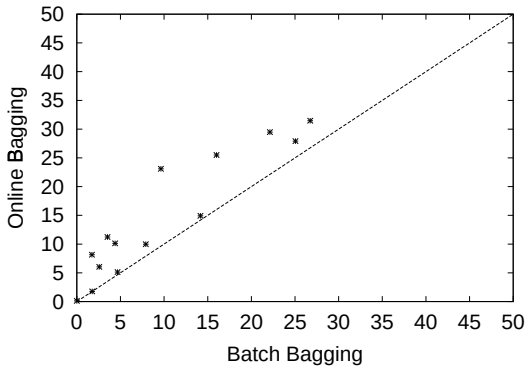

Figure 3.10: Test Error Rates: Batch Bagging vs. Online Bagging with neural network base models.

fixed training set, batch learning algorithms often run faster than online learning algorithms because batch algorithms can often set their parameters once and for all by viewing the entire training set at once while online algorithms have to update their parameters once for every training example. However, in situations where new training examples are arriving continually, batch learning algorithms generally require that the current hypothesis be thrown away and the entire previously-learned training set plus the new examples be learned. This often requires far too much time and is impossible in cases where there is more data than can be stored. The ability to incrementally update learned models is most critical in situations in which data is continually arriving and a prediction must be returned for each example as it arrives. We describe such a scenario in the next section.

3.5.4 Online Dataset

In this section, we discuss our experiments with a dataset that represents a true online learning scenario. This dataset is from the Calendar APprentice (CAP) project (Mitchell, Caruana, Freitag, McDermott, & Zabowski, 1994). The members of this project designed a personal calendar program that helps users keep track of their meeting schedules. The software provides all the usual functionality of calendar software such as adding, deleting, moving, and copying appointments. However, this software has a learning component that learns user preferences of the meeting time, date, location, and duration. After the

Table 3.7: Results (fraction correct): online algorithms (with neural networks trained with 10 update steps per training example).

Dataset	Neural Network	Bagging	Online Neural Net	Online Bagging
Promoters	0.8982*	0.9036*	0.8036	*0.7691*
Balance	0.9194*	**0.9210***	0.8965	**0.9002**
Breast Cancer	0.9527*	**0.9561***	0.9020	0.8987
German Credit	0.7469*	0.7495*	0.7062	**0.7209**
Car Evaluation	0.9422*	**0.9648***	0.8812	**0.8877**
Chess	0.9681*	**0.9827***	0.9023	**0.9185**
Mushroom	1*	1*	0.9995	*0.9988*
Nursery	0.9829*	*0.9743**	0.9411	0.9396
Connect4	0.8199*	**0.8399***	0.7042	**0.7451**
Synthetic-1	0.7217*	**0.7326***	0.6514	**0.6854**
Synthetic-2	0.8564*	**0.8584***	0.8345	**0.8508**
Synthetic-3	0.9824*	0.9824	0.9811	**0.9824**
Census-Income	0.9519*	**0.9533***	0.9487	0.9487
Forest Covertype	0.7573*	**0.7787***	0.6974	**0.7052**

user enters various attributes of a meeting to be set up (e.g., the number of attendees), the software may, for example, suggest to the user that the duration of the meeting should be 30 minutes. The user may then either accept that suggestion or may enter a different duration. The user's selection is used to make a training example that is then used to update the software's current hypothesis of what the user's preferred duration is as a function of the other meeting attributes. The software maintains one hypothesis for each value to be predicted (meeting time, date, location, and duration). In CAP, the hypothesis is a set of rules extracted from a decision tree constructed by a procedure similar to ID3. Every night, a new decision tree is constructed using 120 randomly-selected appointments from the user's previous 180 appointments (this number was chosen through trial and error). A set of rules is extracted from this decision tree by converting each path in the decision tree into an if-then rule, where the "if" portion is the conjunction of the tests at each nonterminal node along the path, and the "then" portion is the class at the leaf node of the path. Preconditions of these rules are removed if the removal does not result in reduced accuracy over the previous 180 appointments. Any new rules that are duplicates of previously-generated rules

Table 3.8: Running Times (seconds): batch and online bagging (with Decision Trees)

Dataset	Decision Tree	Bagging	Online Bagging
Promoters	0.16	1.82	1.88
Balance	0.18	1.66	1.96
Breast Cancer	0.18	1.88	2.28
German Credit	0.22	8.98	9.68
Car Evaluation	0.3	4.28	4.28
Chess	0.98	20.4	20.22
Mushroom	1.12	22.6	23.68
Nursery	1.22	29.68	32.74

are removed. The remaining new rules are then sorted with the previously-generated rules based on their accuracy over the previous 180 appointments. For the entire next day, the software uses these rules to give the user advice. Specifically, in the sorted list of rules, the first rule whose preconditions match the known attributes of the meeting to be scheduled is used to give the user a suggestion.

The dataset that we experimented with contains one user's 1790 appointments over a two-year period. Every appointment has 32 input attributes and four possible target classes. We ran our base model learning algorithms and our online ensemble algorithms on this dataset with the aim of predicting two different targets: the desired duration of the meeting (of which there are 13 possible values) and the day of the week (of which there are six possible values—the user chose to never schedule meetings for Sunday)[3]. The basic structure of our online method of learning is given in Figure 3.11. That is, for each training example, we first predict the desired target using the current hypothesis. This prediction is the suggestion that would be supplied to the user if our learning algorithm was incorporated in the calendar software. We then obtain the desired value of the target—in our case, we obtain it from the dataset but in general this would be obtained from the user. We then check whether our current hypothesis correctly predicted the target and update our running total of the number of errors made so far if necessary. Then we use the online learning algorithm to update our current hypothesis with this new example.

[3]We ran every algorithm separately for each target.

Table 3.9: Running Times (seconds): batch and online bagging (with Naive Bayes)

Dataset	Naive Bayes	Bagging	Online Bagging
Promoters	0.02	0.2	0.22
Balance	<0.02	0.1	0.1
Breast Cancer	0.02	0.14	0.32
German Credit	<0.02	0.14	0.38
Car Evaluation	0.04	0.34	0.44
Chess	0.42	1.02	1.72
Mushroom	0.38	2.14	3.28
Nursery	0.86	1.82	3.74
Connect-4	6.92	33.98	42.04
Synthetic-1	7.48	45.6	64.16
Synthetic-2	5.94	44.78	74.84
Synthetic-3	4.58	44.98	56.2
Census-Income	56.6	131.8	157.4
Forest Covertype	106.0	371.8	520.2

Table 3.10: Running Times (seconds): batch and online bagging (with decision stumps)

Dataset	Decision Stump	Bagging	Online Bagging
Promoters	0.2	0.2	0.3
Balance	0.14	0.14	0.2
Breast Cancer	0.1	0.28	0.3
German Credit	0.36	0.46	0.56
Car Evaluation	0.1	0.14	0.28
Chess	1.46	15	2.98
Mushroom	0.8	2.04	2.68
Nursery	0.26	19.64	5.1
Connect-4	5.94	53.66	33.72
Synthetic-1	3.8	26.6	28.02
Synthetic-2	3.82	30.26	28.34
Synthetic-3	4.06	29.36	36.04
Census-Income	51.4	124.2	131.8
Forest Covertype	176.64	594.34	510.58

Table 3.11: Running times (seconds): batch and online bagging (with neural networks). (1) indicates one update per training example and (10) indicates 10 updates per training example.

Dataset	Neural Network	Bagging	Online Net(1)	Online Bag(1)	Online Net(10)	Online Bag(10)
Promoters	2.58	442.74	0.1	32.42	2.34	334.56
Balance	0.12	12.48	0.02	1.48	0.14	11.7
Breast Cancer	0.12	8.14	0.06	0.94	0.18	6.58
German Credit	0.72	73.64	0.1	7.98	0.68	63.5
Car Evaluation	0.6	36.86	0.1	3.92	0.46	36.82
Chess	1.72	166.78	0.38	20.86	1.92	159.8
Mushroom	7.68	828.38	1.2	129.18	6.64	657.48
Nursery	9.14	1118.98	1.54	140.02	9.22	1004.8
Connect-4	2337.62	156009.3	356.12	28851.42	1133.78	105035.76
Synthetic-1	142.02	15449.58	17.38	2908.48	149.34	16056.14
Synthetic-2	300.96	24447.2	17.74	3467.9	124.22	13327.66
Synthetic-3	203.82	17672.84	24.12	2509.6	117.54	12469.1
Census-Income	4221.4	201489.4	249	23765.8	1405.6	131135.2
Forest Covertype	2071.36	126518.76	635.48	17150.24	805.04	73901.86

The results of predicting the day of the week and the desired meeting duration using the base classifiers in isolation and as part of online bagging are shown in Table 3.12. They are the results of running the algorithm shown in Figure 3.11 with L_o replaced by the base model learning algorithms and the online bagging algorithms with these base models. The accuracy on the nth example is measured to be 1 if the hypothesis learned online using the previous $n-1$ examples classifies the nth example correctly, and 0 otherwise. The values given in the table are the average accuracies over all the examples in the dataset.

To provide a basis for comparison, the learning algorithm used in the CAP project had an average accuracy of around 0.50 for day of the week and around 0.63 for meeting duration. Decision stumps, Naive Bayes classifiers, and neural networks as well as the bagging algorithms with these as base models were not competitive. Decision tree learning was competitive (average accuracy 0.5101 for day of the week and 0.6905 for meeting duration) and online bagging with decision trees performed relatively well (0.5536 and 0.7453) on these tasks. This is consistent with the CAP designers' decision to use decision

Initial condition: $errors = 0$.

OnlineLearning(h, x)

Give suggestion $\hat{y} = h(x)$.

Obtain the desired target value y.

if $y \neq \hat{y}$, then $errors \leftarrow errors + 1$.

$h \leftarrow L_o(h, (x, y))$

Figure 3.11: Basic structure of online learning algorithm used to learn the calendar data. h is the current hypothesis, x is the latest training example to arrive, and L_o is the online learning algorithm.

tree learning in their program. Our online algorithm clearly has a major benefit over their method: we are able to learn from all the training data rather than having to use trial and error to select a window of past examples to learn from, which is the only way to make most batch algorithms practical for this type of problem in which data is continually being generated.

3.6 Summary

In this chapter, we first reviewed the bagging algorithm and discussed the conditions under which it tends to work well relative to single models. We then derived an online bagging algorithm. We proved the convergence of the ensemble generated by the online bagging algorithm to that of batch bagging subject to certain conditions. Finally we compared the two algorithms empirically on several "batch" datasets of various sizes and illustrated the performance of online bagging in a domain in which data is generated continuously.

Table 3.12: Results (fraction correct) on Calendar Apprentice Dataset

Decision Trees		
Target	Single Model	Online Bagging
Day	0.5101	0.5536
Duration	0.6905	0.7453
Naive Bayes		
Target	Single Model	Online Bagging
Day	0.4520	0.3777
Duration	0.1508	0.1335
Decision Stumps		
Target	Single Model	Online Bagging
Day	0.1927	0.1994
Duration	0.2626	0.2553
Neural Networks (one update per example)		
Target	Single Model	Online Bagging
Day	0.3899	0.2972
Duration	0.5106	0.4615
Neural Networks (ten updates per example)		
Target	Single Model	Online Bagging
Day	0.4028	0.4380
Duration	0.5196	0.5240

Chapter 4

Boosting

In this chapter, we first describe the boosting algorithm AdaBoost (Freund & Schapire, 1996, 1997) and some of the theory behind it. We then derive our online boosting algorithm. Finally, we compare the performances of the two algorithms theoretically and experimentally.

4.1 Earlier Boosting Algorithms

The first boosting algorithm (Schapire, 1990) was designed to convert a weak PAC-learning algorithm into a strong PAC-learning algorithm (see (Kearns & Vazirani, 1994) for a detailed explanation of the PAC learning model and Schapire's original boosting algorithm). In the PAC (Probably Approximately Correct) model of learning, a learner has access to a set of labeled examples of the form $(x, c(x))$ where each x is chosen randomly from a fixed but unknown probability distribution $\mathcal{D}$ over the domain X, and c is the target concept that the learner is trying to learn. The concept c is drawn from some concept class $\mathcal{C}$ and $c(x) = 1$ if the example x is in the concept and $c(x) = 0$ if not. The learner is expected to return a hypothesis $h : X \to \{0, 1\}$ which hopefully has small error, which is defined to be $P_{x \in \mathcal{D}}(h(x) \neq c(x))$—this is the probability that the hypothesis and true concept disagree on an example drawn randomly from the same distribution used to generate the training examples. A strong PAC-learning algorithm is an algorithm that, given any $\epsilon > 0$, $\delta > 0$, and access to training examples drawn randomly from $\mathcal{D}$, returns a

hypothesis having error at most ϵ with probability at least $1-\delta$. The algorithm must do this in a running time polynomial in $1/\epsilon$, $1/\delta$, the complexity of the target concept, and some appropriate measure of the dimensionality of the example space. A weak PAC-learning algorithm is the same as a strong PAC-learning algorithm except that the set of acceptable values for ϵ may be restricted. More precisely, there must exist some $\gamma > 0$ such that any $\epsilon \in [1/2 - \gamma, 1/2]$ may be selected.

A boosting algorithm is formally defined to be an algorithm that converts a weak learning algorithm into a strong learning algorithm. That is, it *boosts* a learning algorithm that performs slightly better than random chance on a two-class problem into an algorithm that performs very well on that problem. Schapire's original boosting algorithm (Schapire, 1990) is a recursive algorithm. At the bottom of the recursion, it combines three hypotheses generated by the weak learning algorithm. The error of this combination is provably lower than the errors of the weak hypotheses. Three of these combinations are then constructed and combined to form a combination with still lower error. Additional levels in the recursion are constructed until the desired error bound is reached. Freund (Freund, 1995) later devised the "boost-by-majority" algorithm which, like the original algorithm, combines many weak hypotheses, but it combines them all at the same level, i.e., there is no multi-level hierarchy as there is for the original boosting algorithm. However, this algorithm has one practical deficiency: the value of γ as defined above (the amount by which the weak learning algorithm performs better than random chance) has to be known in advance. The AdaBoost algorithm eliminates this requirement. In fact, it is called Adaboost because it adapts to the actual errors of the hypotheses returned by the weak learning algorithm. As explained in Chapter 2, AdaBoost is an ensemble learning algorithm that combines a set of base models made diverse by presenting the base model learning algorithm with different distributions over the training set. We explain how this happens now.

4.2 The AdaBoost Algorithm

The AdaBoost algorithm, which is the boosting algorithm that we use, generates a sequence of base models with different weight distributions over the training set. The Ad-

AdaBoost$(\{(x_1, y_1), \ldots, (x_N, y_N)\}, L_b, M)$

Initialize $D_1(n) = 1/N$ for all $n \in \{1, 2, \ldots, N\}$.

For $m = 1, 2, \ldots, M$:

$h_m = L_b(\{(x_1, y_1), \ldots, (x_N, y_N)\}, D_m)$.

Calculate the error of $h_m : \epsilon_m = \sum_{n:h_m(x_n) \neq y_n} D_m(n)$.

If $\epsilon_m \geq 1/2$ then,

set $M = m - 1$ and abort this loop.

Update distribution D_m: for all n,

$$D_{m+1}(n) = D_m(n) \times \begin{cases} \frac{1}{2(1-\epsilon_m)} & \text{if } h_m(x_n) = y_n \\ \frac{1}{2\epsilon_m} & \text{otherwise} \end{cases}$$

Output the final hypothesis:

$h_{fin}(x) = \text{argmax}_{y \in Y} \sum_{m:h_m(x)=y} log\frac{1-\epsilon_m}{\epsilon_m}$.

Figure 4.1: AdaBoost algorithm: $\{(x_1, y_1), \ldots, (x_N, y_N)\}$ is the set of training examples, L_b is the base model learning algorithm, and M is the number of base models to be generated.

aBoost algorithm is shown in Figure 4.1. Figure 4.2 depicts AdaBoost in action. Its inputs are a set of N training examples, a base model learning algorithm L_b, and the number M of base models that we wish to combine. AdaBoost was originally designed for two-class classification problems; therefore, for this explanation we will assume that there are two possible classes. However, AdaBoost is regularly used with a larger number of classes. The first step in Adaboost is to construct an initial distribution of weights D_1 over the training set which assigns equal weight to all N training examples. For example, a set of 10 training examples with weight $1/10$ each is depicted in Figure 4.2 in the leftmost column where every box, which represents one training example, is of the same size. We now enter the loop in the algorithm. To construct the first base model, we call L_b with the training set weighted by D_1[1]. After getting back a hypothesis h_1, we calculate its error ϵ_1 on the training set itself, which is just the sum of the weights of the training examples that h_1 misclassifies. We require that $\epsilon_1 < 1/2$ (this is the *weak learning* assumption—the error

[1] If L_b cannot take a weighted training set, then one can call it with a training set generated by sampling with replacement from the original training set according to the distribution D_m.

should be less than what we would achieve through randomly guessing the class). If this condition is not satisfied, then we stop and return the ensemble consisting of the previously-generated base models. If this condition is satisfied, then we calculate a new distribution D_2 over the training examples as follows. Examples that were correctly classified by h_1 have their weights multiplied by $\frac{1}{2(1-\epsilon_1)}$. Examples that were misclassified by h_1 have their weights multiplied by $\frac{1}{2\epsilon_1}$. Note that, because of our condition $\epsilon_1 < 1/2$, correctly classified examples have their weights reduced and misclassified examples have their weights increased. Specifically, examples that h_1 misclassified have their total weight increased to $1/2$ under D_2 and examples that h_1 correctly classified have their total weight reduced to $1/2$ under D_2. In our example in Figure 4.2, the first base model misclassified the first three training examples and correctly classified the remaining ones; therefore, $\epsilon_1 = 3/10$. The three misclassified examples' weights are increased from $1/10$ to $1/6$ (the heights of the top three boxes have increased in the figure from the first column to the second column to reflect this), which means the total weight of the misclassified examples is now $1/2$. The seven correctly classified examples' weights are decreased from $1/10$ to $1/14$ (the heights of the remaining seven boxes have decreased in the figure), which means the total weight of the correctly classified examples is now also $1/2$. Returning to our algorithm, after calculating D_2, we go into the next iteration of the loop to construct base model h_2 using the training set and the new distribution D_2. The point of this weight adjustment is that base model h_2 will be generated by a weak learner (i.e., the base model will have error less than $1/2$); therefore, at least some of the examples misclassified by h_1 will have to be learned. We construct M base models in this fashion.

The ensemble returned by AdaBoost is a function that takes a new example as input and returns the class that gets the maximum weighted vote over the M base models, where each base model's weight is $log(\frac{1-\epsilon_m}{\epsilon_m})$, which is proportional to the base model's accuracy on the weighted training set presented to it. According to Freund and Schapire, this method of combining is derived as follows. If we have a two-class problem, then given an instance x and base model predictions $h_m(x)$ for $m \in \{1, \ldots, M\}$, by the Bayes optimal decision rule we should choose the class y_1 over y_2 if

$$P(Y = y_1 | h_1(x), \ldots, h_M(x)) > P(Y = y_2 | h_1(x), \ldots, h_M(x)).$$

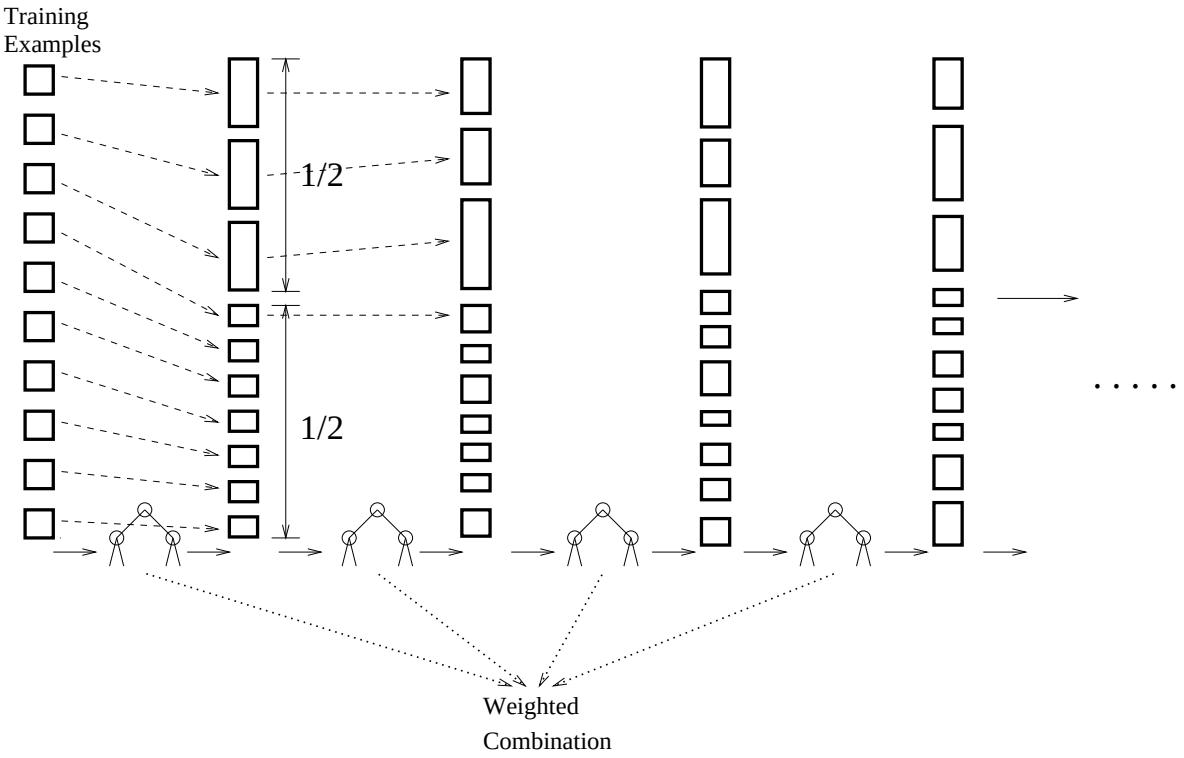

Figure 4.2: The Batch Boosting Algorithm in action.

By Bayes's rule, we can rewrite this as

$$\frac{P(Y=y_1)P(h_1(x),\ldots,h_M(x)|Y=y_1)}{P(h_1(x),\ldots,h_M(x))} > \frac{P(Y=y_2)P(h_1(x),\ldots,h_M(x)|Y=y_2)}{P(h_1(x),\ldots,h_M(x))}.$$

Since the denominator is the same for all classes, we disregard it. Assume that the errors of the different base models are independent of one another and of the target concept. That is, assume that the event $h_m(x) \neq y$ is conditionally independent of the actual label y and the predictions of the other base models. Then, we get

$$P(Y=y_1) \prod_{m:h_m(x)\neq y_1} \epsilon_m \prod_{m:h_m(x)=y_1} (1-\epsilon_m) >$$
$$P(Y=y_2) \prod_{m:h_m(x)\neq y_2} \epsilon_m \prod_{m:h_m(x)=y_2} (1-\epsilon_m)$$

where $\epsilon_m = P(h_m(x) \neq y)$ and y is the actual label. This intuitively makes sense: we want to choose the class y that has the best combination of high prior probability (the $P(Y=y)$ factor), high accuracies $(1-\epsilon_m)$ of models that vote for class y (those for which

$h_m(x) = y$), and high errors (ϵ_m) for models that vote against class y (those for which $h_m(x) \neq y$). If we add the trivial base model h_o that always predicts class y_1, then we can replace $P(Y = y_1)$ with $1 - \epsilon_0$ and $P(Y = y_2)$ with ϵ_0. Dividing by the ϵ_m's, we get

$$\prod_{m:h_m(x)=y_1} \frac{1-\epsilon_m}{\epsilon_m} > \prod_{m:h_m(x)=y_2} \frac{1-\epsilon_m}{\epsilon_m}.$$

Taking logarithms and replacing logs of products with sums of logs, we get

$$\sum_{m:h_m(x)=y_1} \log\left(\frac{1-\epsilon_m}{\epsilon_m}\right) > \sum_{m:h_m(x)=y_2} \log\left(\frac{1-\epsilon_m}{\epsilon_m}\right).$$

If there are more than two classes, one can simply choose the class y that maximizes

$$\sum_{m:h_m(x)=y} \log\left(\frac{1-\epsilon_m}{\epsilon_m}\right).$$

which is the method that AdaBoost uses to choose the classification of a new example.

4.3 Why and When Boosting Works

The question of why boosting performs as well as it has in experimental studies performed so far (e.g., (Freund & Schapire, 1996; Bauer & Kohavi, 1999)) has not been precisely answered. However, there are characteristics that appear systematically in experimental studies and can be seen just from the algorithm.

Unlike bagging, which is largely a variance reduction method, boosting appears to reduce both bias and variance. After a base model is trained, misclassified training examples have their weights increased and correctly classified examples have their weights decreased for the purpose of training the next base model. Clearly, boosting attempts to correct the bias of the most recently constructed base model by focusing more attention on the examples that it misclassified. This ability to reduce bias enables boosting to work especially well with high-bias, low-variance base models such as decision stumps and Naive Bayes classifiers.

One negative aspect of boosting is the difficulty that it has with noisy data. Noisy data is generally difficult to learn; therefore, in boosting, many base models tend to misclassify

noisy training examples, causing their weights to increase too much. This causes the base model learning algorithm to focus too much on the noisy examples at the expense of the remaining examples and overall performance.

There is a well-known theorem by Freund and Schapire proving that the training error of the boosted ensemble decreases exponentially in the number of base models.

Theorem 3 *Suppose the weak learning algorithm* ***WeakLearn****, when called by* ***AdaBoost****, generates hypotheses with errors* $\epsilon_1, \epsilon_2, \ldots, \epsilon_M$*. Then the error* $\epsilon = P_{i \sim D}[h_f(x_i) \neq y_i]$ *of the final hypothesis* h_{fin} *returned by* ***AdaBoost*** *is bounded above by* $2^M \prod_{m=1}^{M} \sqrt{\epsilon_m(1-\epsilon_m)}$.

One would think that increasing the number of base models so much that the training error goes to zero would lead to overfitting due to the excessive size of the ensemble model. However, several experiments have demonstrated that adding more and more base models even after the training error has gone to zero continues to reduce the test error (Drucker & Cortes, 1996; Quinlan, 1996; Breiman, 1998). There has been some more recent work (Schapire, Freund, Bartlett, & Lee, 1997, 1998) that attempts to explain this phenomenon in terms of the distribution of *margins* of the training examples, where the margin of an example is the total weighted vote for the correct class minus the maximum total weighted vote received by any incorrect class. That is, it is claimed that adding more base models to the ensemble tends to increase the distribution of margins over the training set. For the training error to reach zero, one only needs a positive margin on all the training examples; however, even after this is achieved, boosting continues to increase the margins, thereby increasing the separation between the examples in the different classes. However, this explanation has been shown experimentally to be incomplete (Breiman, 1997). A theoretical explanation for boosting's seeming immunity to overfitting has not yet been obtained and is an active area of research. However, this immunity and boosting's good performance in experiments makes it one of the most popular ensemble methods, which is why we now devise an online version.

Initial conditions: For all $m \in \{1, 2, \ldots, M\}$, $\lambda_m^{sc} = 0, \lambda_m^{sw} = 0$.

OnlineBoosting($\mathbf{h}$, L_o, (x, y))

Set the example's "weight" $\lambda = 1$.

For each base model h_m, ($m \in \{1, 2, \ldots, M\}$) in $\mathbf{h}$,

Set k according to $Poisson(\lambda_d)$.

Do k times

$h_m = L_o(h_m, (x, y))$

If $y = h_m(x)$

then

$\lambda_m^{sc} \longleftarrow \lambda_m^{sc} + \lambda$

$\epsilon_m \longleftarrow \frac{\lambda_m^{sw}}{\lambda_m^{sc} + \lambda_m^{sw}}$

$\lambda \longleftarrow \lambda \left(\frac{1}{2(1-\epsilon_m)}\right)$

else

$\lambda_m^{sw} \longleftarrow \lambda_m^{sw} + \lambda$

$\epsilon_m \longleftarrow \frac{\lambda_m^{sw}}{\lambda_m^{sc} + \lambda_m^{sw}}$

$\lambda \longleftarrow \lambda \left(\frac{1}{2\epsilon_m}\right)$

To classify new examples:

Return $h(x) = \text{argmax}_{c \in Y} \sum_{m: h_m(x) = c} log \frac{1-\epsilon_m}{\epsilon_m}$.

Figure 4.3: Online Boosting Algorithm: $\mathbf{h}$ is the set of M base models learned so far, (x, y) is the latest training example to arrive, and L_o is the online base model learning algorithm.

4.4 The Online Boosting Algorithm

Just like bagging, boosting seems to require that the entire training set be available at all times for every base model to be generated. In particular, at each iteration of boosting, we call the base model learning algorithm on the entire weighted training set and calculate the error of the resulting base model on the entire training set. We then use this error to adjust the weights of all the training examples.

However, we have devised an online version of boosting that is similar in principle to our online bagging algorithm. Recall that the online bagging algorithm assigns each train-

ing example a Poisson parameter $\lambda = 1$ to correspond to the weight $1/N$ assigned to each training example by the batch bagging algorithm. When the batch boosting algorithm generates the first base model, every training example is assigned a weight $1/N$ just like batch bagging, so the online boosting algorithm assigns each example a Poisson parameter $\lambda = 1$ just like online bagging. For subsequent base models, online boosting updates the Poisson parameter for each training example in a manner very similar to the way batch boosting updates the weight of each training example—increasing it if the example is misclassified and decreasing it if the example is correctly classified.

The pseudocode of our online boosting algorithm is given in Figure 4.3. Because our algorithm is an online algorithm, its inputs are the current set of base models $\mathbf{h} = \{h_1, \ldots, h_M\}$ and the associated parameters $\lambda^{\mathbf{sc}} = \{\lambda_1^{sc}, \ldots, \lambda_M^{sc}\}$ and $\lambda^{\mathbf{sw}} = \{\lambda_1^{sw}, \ldots, \lambda_M^{sw}\}$ (these are the sums of the weights of the correctly classified and misclassified examples, respectively, for each of the M base models), as well as an online base model learning algorithm L_o and a new labeled training example (x, y). The algorithm's output is a new classification function that is composed of updated base models $\mathbf{h}$ and associated parameters $\lambda^{\mathbf{sc}}$ and $\lambda^{\mathbf{sw}}$. The algorithm starts by assigning the training example (x, y) the "weight" $\lambda = 1$. Then the algorithm goes into a loop, in which one base model is updated in each iteration. For the first iteration, we choose k according to the $Poisson(\lambda)$ distribution, and call L_o, the online base model learning algorithm, k times with base model h_1 and example (x, y). We then see if the updated h_1 has learned the example, i.e., whether h_1 classifies it correctly. If it does, we update λ_1^{sc}, which is the sum of the weights of the examples that h_1 classifies correctly. We then calculate ϵ_1 which, just like in boosting, is the weighted fraction of the total examples that h_1 has misclassified. We then update λ by multiplying it by the same factor $\frac{1}{2(1-\epsilon_1)}$ that we do in AdaBoost. On the other hand, if h_1 misclassifies example x, then we increment λ_1^{sw}, which is the sum of the weights of the examples that h_1 misclassifies. Then we calculate ϵ_1 and update λ by multiplying it by $\frac{1}{2\epsilon_1}$, which is the same factor that is used by AdaBoost for misclassified examples. We then go into the second iteration of the loop to update the second base model h_2 with example (x, y) and its new updated weight λ. We repeat this process for all M base models. The final ensemble returned has the same form as in AdaBoost, i.e., it is a function that takes a new example and returns the class that gets the maximum weighted vote over all the base models, where

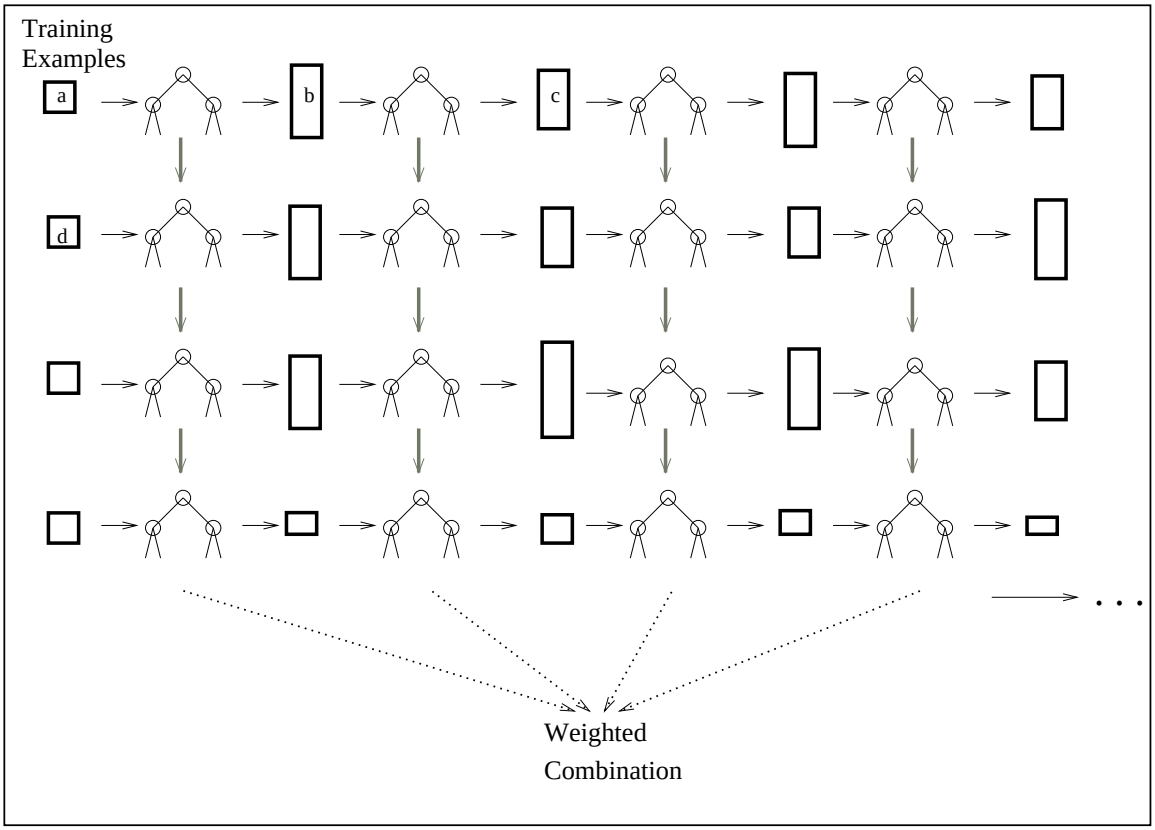

Figure 4.4: Illustration of online boosting in progress. Each row represents one example being passed in sequence to all the base models for updating; time runs down the diagram. Each base model (depicted here as a tree) is generated by updating the base model above it with the next weighted training example. In the upper left corner (point "a" in the diagram) we have the first training example. This example updates the first base model but is still misclassified after training, so its weight is increased (the rectangle "b" used to represent it is taller). This example with its higher weight updates the second base model which then correctly classifies it, so its weight decreases (rectangle "c").

each base model's vote is $log(\frac{1-\epsilon_m}{\epsilon_m})$, which is proportional to the base model's accuracy on the weighted training set presented to it.

Figure 4.4 illustrates our online boosting algorithm in action. Each row depicts one training example updating all the base models. For example, at the upper left corner (rectangle "a" in the diagram) we have the first training example. The first base model (the tree to the right of the "a" rectangle) is updated with example "a," however the first base model still misclassifies that example; therefore the example's weight is increased. This is

depicted as rectangle “b” which is taller to indicate that the example now has more weight. The second base model is now updated with the same training example but with its new higher weight. The second base model correctly classifies it, therefore its weight is reduced (depicted by rectangle “c”). This continues until all the base models are updated, at which time we throw away this training example and pick up the next example (rectangle “d” in the figure). We then update all the base models with this new example, increasing and decreasing this example’s weight as necessary. Each column of base models (depicted in the diagram as trees) is actually the same base model being incrementally updated by each new training example.

One area of concern is that, in AdaBoost, an example’s weight is adjusted based on the performance of a base model on the entire training set, whereas in online boosting, the weight adjustment is based on the base model’s performance only on the examples seen earlier. To see why this may be an issue, consider running AdaBoost and online boosting on a training set of size 10000. In AdaBoost, the first base model h_1 is generated from all 10000 examples before being tested on, say, the tenth training example. In online boosting, h_1 is generated from only the first ten examples before being tested on the tenth example. Clearly, we may expect the two h_1’s to be very different; therefore, h_2 in AdaBoost and h_2 in online boosting may be presented with different weights for the tenth example. This may, in turn, lead to very different weights for the tenth example when presented to h_3 in each algorithm, and so on.

We will see in Section 4.6 that this is a problem that often leads to online boosting performing worse than batch boosting, and sometimes significantly so. Online boosting is especially likely to suffer a large loss initially when the base models have been trained with very few examples, and the algorithm may never recover. Even when online boosting’s performance ends up comparable to that of batch boosting, one may obtain learning curves such as what is shown in Figure 4.5. The learning curve clearly shows that online boosting performs poorly relative to batch boosting for smaller numbers of examples and then finally catches up to batch boosting by the time the entire training set has been learned.

To alleviate this problem with online boosting, we implemented *primed* online boosting, which trains with some initial part of the training set in batch mode and then trains with the remainder in online mode. The hope is to reduce some of the loss that online boosting

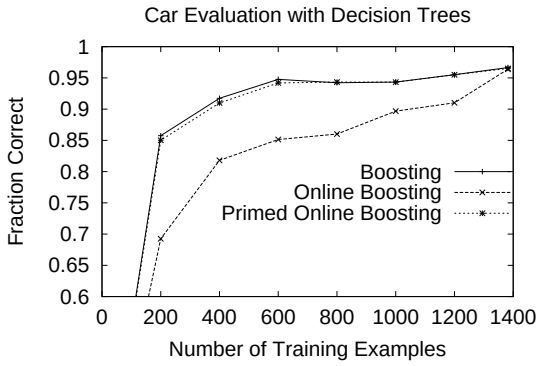

Figure 4.5: Learning curves for Car Evaluation dataset.

suffers initially by training in batch mode with a reasonable initial portion of the training set. We would like to get results such as what is shown for "Primed Online Boosting" in Figure 4.5, which stays quite close to batch boosting for all numbers of training examples and ends with an accuracy better than what online boosting without priming achieves.

4.5 Convergence of Batch and Online Boosting for Naive Bayes

To demonstrate the convergence of online and batch boosting, we do a proof by induction over the base models. That is, for the base case, we demonstrate that if both batch and online boosting are given the same training set, then the first base model returned by online boosting converges to that returned by batch boosting as $N \to \infty$. Then we show that if the mth base models converge then the $m+1$st base models converge. In the process, we show that the base model errors also converge. This is sufficient to show that the classification functions returned by online boosting and batch boosting converge.

We first discuss some preliminary points: definitions and lemmas that will be useful to us. Then we discuss the proof of convergence.

4.5.1 Preliminaries

Lemma 3, Corollary 2, and Corollary 3 are standard results (Billingsley, 1995; Grimmett & Stirzaker, 1992), so we state them without proof.

Lemma 3 *If $X_1, X_2, \ldots$ and $Y_1, Y_2, \ldots$ are sequences of random variables such that $X_n \xrightarrow{P} X$ and $Y_n \xrightarrow{P} Y$, then $f(X_n, Y_n) \xrightarrow{P} f(X, Y)$ for any continuous function $f : \mathcal{R}^2 \to \mathcal{R}$.*

Corollary 2 *If $X_n \xrightarrow{P} X$ and $Y_n \xrightarrow{P} Y$, then $X_n Y_n \xrightarrow{P} XY$.*

Corollary 3 *If $X_n \xrightarrow{P} X$, $Y_n \xrightarrow{P} Y$, and $Y_n > 0$ for all n, then $\frac{X_n}{Y_n} \xrightarrow{P} \frac{X}{Y}$.*

Lemma 4 *If $X_1, X_2, \ldots$ and X are discrete random variables and $X_n \xrightarrow{P} X$, then $I(X_n = x) \xrightarrow{P} I(X = x)$ for all possible values x.*

Proof: We have $X_n \xrightarrow{P} X$, which implies that, for all $\epsilon > 0$, $P(|X_n - X| > \epsilon) \to 0$ as $n \to \infty$. Since the variables X and X_n for all n are discrete-valued, we have that $P(X_n - X = 0) \to 1$ as $n \to \infty$. This implies that $P(I(X_n = x) - I(X = x) = 0) \to 1$ as $n \to \infty$, which implies the statement of the theorem. ∎

Lemma 5 *For all N, define $a_N(n)$ and $b_N(n)$ over the integers $n \in \{1, 2, \ldots, N\}$ such that $0 \leq a_N(n) \leq 1$, $0 \leq b_N(n) \leq 1$, $a_N(n) \xrightarrow{P} b_N(n)$ (as $N \to \infty$), and $\sum_{n=1}^{N} a_N(n) = \sum_{n=1}^{N} b_N(n) = 1$. If $X_1, X_2, \ldots$ and $Y_1, Y_2, \ldots$ are uniformly bounded random variables such that $X_N \xrightarrow{P} Y_N$, then $\sum_{n=1}^{N} a_N(n) X_n \xrightarrow{P} b_N(n) Y_n$.*

Proof: We want to show, by the definition of convergence in probability, that for all $\epsilon > 0$, $P(|\sum_{n=1}^{N} a_N(n) X_n - b_N(n) Y_n| > \epsilon) \to 0$ as $N \to \infty$. We have

$$P\left(\left|\sum_{n=1}^{N} a_N(n) X_n - b_N(n) Y_n\right| > \epsilon\right) \quad \leq \quad P\left(\sum_{n=1}^{N} |a_N(n) X_n - b_N(n) Y_n| > \epsilon\right).$$

It is sufficient for us to show that, for all $\epsilon > 0$ and $\delta > 0$, there exists an N_δ such that for all $N \geq N_\delta$, $P(\sum_{n=1}^{N} |a_N(n) X_n - b_N(n) Y_n| > \epsilon) < \delta$.

We already have that $X_N \xrightarrow{P} Y_N$ and $a_N(n) \xrightarrow{P} b_N(n)$; therefore, $a_N(n)X_N \xrightarrow{P} b_N(n)Y_N$. Since $\sum_{n=1}^{N} a_N(n) = \sum_{n=1}^{N} b_N(n) = 1$, by Corollary 1, $\sum_{n=1}^{N} |a_N(n) - b_N(n)| \to 0$. Therefore, we can choose a constant d such that $|a_N(n) - b_N(n)| \leq d/N$. This, combined with the uniform boundedness of X_n and Y_n, means that, for any $\epsilon_1 > 0$ and $\epsilon_2 > 0$, we can choose N_o such that for all $N > N_o$, $P(|a_N(n)X_N - b_N(n)Y_N| > \epsilon_1/N) < \epsilon_2$. We will specify further restrictions on ϵ_1 and ϵ_2 later, but for now, it is sufficient that they be positive.

Have $N > N_o$ so that we can write

$$
\begin{aligned}
&P\left(\sum_{n=1}^{N} |a_N(n)X_n - b_N(n)Y_n| > \epsilon\right) = \\
&P\left(\sum_{n=1}^{N_o} |a_N(n)X_n - b_N(n)Y_n| + \sum_{n=N_o+1}^{N} |a_N(n)X_n - b_N(n)Y_n| > \epsilon\right) \leq \\
&P\left(\sum_{n=1}^{N_o} |a_N(n)X_n - b_N(n)Y_n| > \frac{\epsilon}{2}\right) + \\
&P\left(\sum_{n=N_o+1}^{N} |a_N(n)X_n - b_N(n)Y_n| > \frac{\epsilon}{2}\right). \qquad (4.1)
\end{aligned}
$$

We can simply constrain each of the last two terms to be less than $\delta/2$ and we will be finished. Let us look at the first term first. We want

$$P\left(\sum_{n=1}^{N_o} |a_N(n)X_n - b_N(n)Y_n| > \frac{\epsilon}{2}\right) < \frac{\delta}{2}.$$

Since the X_n's and Y_n's are uniformly bounded and $|a_N(n) - b_N(n)| \leq d/N$ as we mentioned earlier, we know that there exists an M such that $|a_N(n)X_n - b_N(n)Y_n| < M/N$ for all n. So we have

$$P\left(\sum_{n=1}^{N_o} |a_N(n)X_n - b_N(n)Y_n| > \frac{\epsilon}{2}\right) \quad < \quad P\left(\frac{N_o M}{N} > \frac{\epsilon}{2}\right).$$

It is sufficient for us to have

$$\frac{N_o M}{N} < \frac{\epsilon}{2} \Longrightarrow N > \frac{2N_o M}{\epsilon}.$$

Let us define c such that $N = cN_o$. This means that it is sufficient to have $c > \frac{2M}{\epsilon}$ in order to satisfy the constraint on the first term of Equation 4.1.

Now let us look at the second term. We want

$$P\left(\sum_{n=N_o+1}^{N} |a_N(n)X_n - b_N(n)Y_n| > \frac{\epsilon}{2}\right) < \frac{\delta}{2}. \tag{4.2}$$

Recall that we have, for $n > N_o$, $P(|a_N(n)X_n - b_N(n)Y_n| > \epsilon_1/N) < \epsilon_2$. We now use the following "coin tossing" argument.

Let us say that the ith coin is heads if $|a_N(i)X_i - b_N(i)Y_i| > \epsilon_1/N$ and tails otherwise for $i \in \{N_o+1, N_o+2, \ldots, N\}$. Therefore, the probability that the ith coin is heads is less than ϵ_2. We can upper bound the probability of having more than some number k heads using Markov's Inequality:

$$P(H > k) < \frac{E(H)}{k}$$

where H is a random variable representing the number of heads in the $N - N_o$ tosses. Clearly, $E(H) < (N - N_o)\epsilon_2$. We now choose some $\gamma > 0$ such that, by Markov's Inequality,

$$P(H > (N - N_o)(1+\gamma)\epsilon_2) < \frac{(N - N_o)\epsilon_2}{(N - N_o)(1+\gamma)\epsilon_2} = \frac{1}{1+\gamma}. \tag{4.3}$$

Now we translate from the realm of coins back to the realm of our original random variables—specifically to our sum $\sum_{n=N_o+1}^{N} |X_n - X|$. Recall that $M/N > |a_N(i)X_i - b_N(i)Y_i| > \epsilon_1/N$ if the ith coin is heads, and $|a_N(i)X_i - b_N(i)Y_i| \leq \epsilon_1/N$ otherwise. So for each head, we add at most M/N to our sum, and for each tail we add at most ϵ_1/N to our sum. So if we have less than $(N - N_o)(1+\gamma)\epsilon_2$ heads, then our sum is at most $(N - N_o)(1+\gamma)\epsilon_2 M/N + (N - N_o - (N - N_o)(1+\gamma)\epsilon_2)\epsilon_1/N = (N - N_o)(1+\gamma)\epsilon_2 M/N + (N - N_o)(1 - (1+\gamma)\epsilon_2)\epsilon_1/N$. Therefore, we can state the contrapositive, which is that if the sum is at least $(N - N_o)(1+\gamma)\epsilon_2 M/N + (N - N_o)(1 - (1+\gamma)\epsilon_2)\epsilon_1/N$, then we have at least $(N - N_o)(1+\gamma)\epsilon_2$ heads. This means that the probability of achieving at least the given sum is less than the probability of achieving at least the given number of heads. That is,

$$P(H > (N - N_o)(1+\gamma)\epsilon_2) \geq$$
$$P\left(\sum_{n=N_o+1}^{N} |a_N(n)X_n - b_N(n)Y_n| > (N - N_o)\left[(1+\gamma)\epsilon_2\frac{M}{N} + (1 - (1+\gamma)\epsilon_2)\frac{\epsilon_1}{N}\right]\right) =$$

$$P\left(\sum_{n=N_o+1}^{N}|a_N(n)X_n - b_N(n)Y_n| > (c-1)N_o\left[(1+\gamma)\epsilon_2\frac{M}{N} + (1-(1+\gamma)\epsilon_2)\frac{\epsilon_1}{N}\right]\right) \geq$$
$$P\left(\sum_{n=N_o+1}^{N}|a_N(n)X_n - b_N(n)Y_n| > cN_o\left[(1+\gamma)\epsilon_2\frac{M}{N} + \frac{\epsilon_1}{N} - (1+\gamma)\frac{\epsilon_1}{N}\epsilon_2\right]\right) \geq$$
$$P\left(\sum_{n=N_o+1}^{N}|a_N(n)X_n - b_N(n)Y_n| > cN_o\left[(1+\gamma)\epsilon_2\frac{M}{N} + (1+\gamma)\epsilon_1\frac{M}{N}\right]\right) =$$
$$P\left(\sum_{n=N_o+1}^{N}|a_N(n)X_n - b_N(n)Y_n| > cN_o(1+\gamma)\frac{M}{N}(\epsilon_1+\epsilon_2)\right).$$

Putting the last inequality together with Equation 4.3 yields

$$P\left(\sum_{n=N_o+1}^{N}|a_N(n)X_n - b_N(n)Y_n| > cN_o(1+\gamma)\frac{M}{N}(\epsilon_1+\epsilon_2)\right) < \frac{1}{1+\gamma}.$$

Comparing this to Equation 4.2, which is what we want, we need to have $\frac{1}{1+\gamma} < \frac{\delta}{2}$ and $cN_o(1+\gamma)\frac{M}{N}(\epsilon_1+\epsilon_2) < \frac{\epsilon}{2}$. The first constraint requires us to choose γ such that

$$\gamma > \frac{2}{\delta} - 1.$$

The second requirement gives us a constraint on ϵ_1 and ϵ_2 as follows:

$$cN_o(1+\gamma)M(\epsilon_1+\epsilon_2) < \frac{N\epsilon}{2} \Longrightarrow (\epsilon_1+\epsilon_2) < \frac{\epsilon}{2(1+\gamma)M}.$$

For example, choosing both ϵ_1 and ϵ_2 less than $\frac{\epsilon}{4(1+\gamma)M}$ satisfies the constraint.

We are given δ and ϵ. Given these, we have described how to choose γ which, together with our known bound M allows us to choose c, ϵ_1, and ϵ_2. These allow us to choose N_o and hence the minimum N needed to satisfy all the constraints, which is sufficient to complete the proof. ∎

Lemma 6 *If $X_1, X_2, \ldots$ and $Y_1, Y_2, \ldots$ are sequences of random variables and X and Y are random variables such that $X_n \xrightarrow{P} X$ and $Y_n \xrightarrow{P} Y$ and if there exists a $d > 0$ such that $|X - Y| \geq d$, then $I(X_n > Y_n) \xrightarrow{P} I(X > Y)$.*

Proof: We will prove this using the definition of convergence in probability. That is, we will prove that for all $\epsilon > 0$ and $\delta > 0$, there exists an N_o such that for all $n > N_o$, $P(|I(X_n > Y_n) - I(X > Y)| > \epsilon) < \delta$.

Clearly there are two relevant cases: the case where $X > Y$ and the case where $X < Y$. Let us first assume that $X > Y$ so that $I(X > Y) = 1$. This means there exists some $d > 0$ such that $X - Y \geq d$. Since we have $X_n \xrightarrow{P} X$ and $Y_n \xrightarrow{P} Y$, we have that, for some $\epsilon_x > 0$, $\delta_x > 0$, $\epsilon_y > 0$, and $\delta_y > 0$, there exist N_x and N_y such that for all $n_x > N_x$ and $n_y > N_y$,

$$P(|X_{n_x} - X| > \epsilon_x) < \delta_x$$
$$P(|Y_{n_y} - Y| > \epsilon_y) < \delta_y.$$

If we choose $\delta_x = \delta/2$, $\delta_y = \delta/2$, $\epsilon_x = d/4$, and $\epsilon_y = d/4$, then

$$P(X_n - Y_n > d - \epsilon_x - \epsilon_y = d/2) \geq (1 - \delta_x)(1 - \delta_y) = 1 - \delta + \delta^2/4$$

for $n > max(n_x, n_y)$. This means that $P(I(X_n > Y_n) = 1 | I(X > Y) = 1) \geq 1 - \delta + \delta^2/4$. For the case where $X < Y$, repeat the above derivation with X and Y reversed. In this case, we get $P(I(X_n > Y_n) = 0 | I(X > Y) = 0) \geq 1 - \delta + \delta^2/4$. Putting it all together, we get

$$\begin{aligned} &P(I(X_n > Y_n) = 1 | I(X > Y) = 1) P(I(X > Y) = 1) + \\ &\quad P(I(X_n > Y_n) = 0 | I(X > Y) = 0) P(I(X > Y) = 0) \geq 1 - \delta + \frac{\delta^2}{4} \Longrightarrow \\ &\quad P(|I(X_n > Y_n) - I(X > Y)| > 0) \leq \delta - \frac{\delta^2}{4} \end{aligned}$$

which is stronger than what is needed to prove the desired statement. ∎

Define $\mathbf{D_m^b}$ to be a vector of N weights $D_m^b(n)$—one for each training example—used in the batch boosting algorithm. This is the same set of weights $D_m(n)$ shown in Figure 4.1, but we add the superscript "b" to indicate that these are weights used in batch boosting rather than online boosting. The variable m indexes over the base models 1 through M. Define $\mathbf{D_m^o}$ to be the normalized version of the corresponding vector of weights used in the

online boosting algorithm. Recall that, whereas AdaBoost uses a vector of weights normalized to make it a true probability distribution over the training examples, our online boosting algorithm uses Poisson parameters as weights. For our analysis, it will be helpful to also use the normalized version of the parameters used in online boosting. Define $P_{\mathbf{D^o_m}}(Z)_N$ and $P_{\mathbf{D^b_m}}(Z)_N$ to be the probabilities of event Z in N training examples under the distributions described by $\mathbf{D^o_m}$ and $\mathbf{D^b_m}$, respectively. That is, $P_{\mathbf{D^o_m}}(Z)_N = \sum_{n=1}^{N} D^o_m(n) I(X_n \in Z)$ and $P_{\mathbf{D^b_m}}(Z)_N = \sum_{n=1}^{N} D^b_m(n) I(X_n \in Z)$. Recall that X_n is the nth training example. Define $\mathbf{X_N}$ to be the set of all N training examples.

Lemma 7 *For any event Z defined as a set of attribute and class values, if $D^o_m(n) \xrightarrow{P} D^b_m(n)$ for all n (as $N \to \infty$), then $P_{\mathbf{D^o_m}}(Z)_N \xrightarrow{P} P_{\mathbf{D^b_m}}(Z)_N$.*

Proof: Since $D^o_m(n) \xrightarrow{P} D^b_m(n)$ (as $N \to \infty$) and, clearly $P_{\mathbf{D^o_m}}(\mathbf{X_N})_N = P_{\mathbf{D^b_m}}(\mathbf{X_N})_N = 1$, by Corollary 1 we get $\sum_{n=1}^{N} |D^o_m(n) - D^b_m(n)| I(X_n \in Z) \xrightarrow{P} 0$, which implies that $\sum_{n=1}^{N} D^o_m(n) I(X_n \in Z) \xrightarrow{P} \sum_{n=1}^{N} D^b_m(i) I(X_n \in Z)$. Therefore, $P_{\mathbf{D^o_m}}(Z)_N \xrightarrow{P} P_{\mathbf{D^b_m}}(Z)_N$, which is what we wanted to show. ∎

4.5.2 Main Result

In this section, we prove that, given the same training set, the classification function returned by online boosting with Naive Bayes base models converges to that returned by batch boosting with Naive Bayes base models. However, we first define some of the terms we use in the proof. Define $h^b_m(x)$ as the mth base model returned by AdaBoost and define $h^o_m(x)$ to be the mth base model returned by the online boosting algorithm. Define $\epsilon^b_{m,n}$ to be ϵ_m in AdaBoost (Figure 4.1) after training with n training examples—recall that this is the weighted error of the mth base model on the training set. Define $\epsilon^o_{m,n}$ to be ϵ_m (also the weighted error of the mth base model) in the online boosting algorithm (Figure 4.3) after training with n training examples. The classification function returned by AdaBoost can be written as $h^b(x) = \text{argmax}_{c \in Y} \sum_{m: h^b_m(x) = c} log \frac{1 - \epsilon^b_{m,N}}{\epsilon^b_{m,N}}$. The classification function returned by online boosting can be written as $h^o(x) = \text{argmax}_{c \in Y} \sum_{m: h^o_m(x) = c} log \frac{1 - \epsilon^o_{m,N}}{\epsilon^o_{m,N}}$.

It helps us to write down the classification functions returned by a Naive Bayes classifier learning algorithm when called from the batch boosting and online boosting algorithms. We defined a generic Naive Bayes classifier in Section 2.1.2. However, this definition assumes that the training set is unweighted. For now, we only consider the two-class problem for simplicity; however, we generalize to the multi-class case toward the end of this section. The mth Naive Bayes classifier in an ensemble constructed by batch boosting using n training examples is

$$h^b_{m,n}(x) = I(P_{\mathbf{D^b_m}}(Y=1)_n P_{\mathbf{D^b_m}}(X=x|Y=1)_n > P_{\mathbf{D^b_m}}(Y=0)_n P_{\mathbf{D^b_m}}(X=x|Y=0)_n) \quad (4.4)$$

For example, $P_{\mathbf{D^b_m}}(Y=1)_n$ is the sum of the weights ($D^b_m(i)$) of those among the n training examples $\{(x_1,y_1),(x_2,y_2),\ldots,(x_n,y_n)\}$ whose class values are 1. That is,

$$P_{\mathbf{D^b_m}}(Y=1)_n = \sum_{i=1}^{n} D^b_m(i) I(y_i = 1).$$

We use $P_{\mathbf{D^b_m}}(X=x|Y=1)_n$ as shorthand for $P_{\mathbf{D^b_m}}(X_1=x_{(1)}|Y=1)_n P_{\mathbf{D^b_m}}(X_2=x_{(2)}|Y=1)_n \cdots P_{\mathbf{D^b_m}}(X_{|A|}=x_{|A|}|Y=1)_n$, where $x_{(a)}$ is example x's value for attribute number a and A is the set of attributes. For example, $P_{\mathbf{D^b_m}}(X_1=x_{(1)}|Y=1)_n$ is the sum of the weights of those examples among the n training examples whose class values are 1 and whose values for the first attribute are the same as that of example x, divided by the sum of the weights of the class 1 examples. That is,

$$P_{\mathbf{D^b_m}}(X_1=x_{(1)}|Y=1)_n = \frac{\sum_{i=1}^{n} D^b_m(i) I(x_{i1}=x_{(1)} \wedge y_i = 1)}{\sum_{i=1}^{n} D^b_m(i) I(y_i = 1)}$$

where x_{ij} is the jth attribute value of example i. The mth Naive Bayes classifier in an ensemble returned by online boosting using n training examples is written in a manner similar to the corresponding one of batch boosting. The only difference is that we replace the weights $\mathbf{D^b_m}$ with $\mathbf{D^o_m}$.

$$h^o_{m,n}(x) = I(P_{\mathbf{D^o_m}}(Y=1)_n P_{\mathbf{D^o_m}}(X=x|Y=1)_n > P_{\mathbf{D^o_m}}(Y=0)_n P_{\mathbf{D^o_m}}(X=x|Y=0)_n) \quad (4.5)$$

We prove two lemmas essential to our final theorem. The first lemma states that if the vector of weights for the mth base model under online boosting converges to the corresponding vector under batch boosting, then the online base model itself converges to the batch base model.

Lemma 8 *If* $\mathbf{D^o_m} \xrightarrow{P} \mathbf{D^b_m}$, *then* $h^o_{m,N}(x) \xrightarrow{P} h^b_{m,N}(x)$.

Proof: By Lemma 7, each probability of the form $P_{\mathbf{D^o_m}}(Z)_n$ in the online classifier converges to the corresponding probability $P_{\mathbf{D^b_m}}(Z)_n$ in the batch classifier. For example, $P_{\mathbf{D^o_m}}(Y = 1)_n \xrightarrow{P} P_{\mathbf{D^b_m}}(Y = 1)_n$. By Corollary 2 and Lemma 6, we have $h^o_{m,N}(x) \xrightarrow{P} h^b_{m,N}(x)$. ∎

The next lemma states that if the mth online base model converges to the mth batch base model, then the mth online base model's training error $\epsilon^o_{m,N}$ converges to the mth batch base model's training error $\epsilon^b_{m,N}$.

Lemma 9 *If* $\mathbf{D^o_m} \xrightarrow{P} \mathbf{D^b_m}$ *and* $h^o_{m,N}(x) \xrightarrow{P} h^b_{m,N}(x)$ *then* $\epsilon^o_{m,N} \xrightarrow{P} \epsilon^b_{m,N}$.

Proof: To do this, we must first write down suitable expressions for $\epsilon^o_{m,N}$ and $\epsilon^b_{m,N}$.

In batch boosting, the mth base model's error on example i is the error of the Naive Bayes classifier constructed using the entire training set: $D^b_m(i)|y_i - h^b_{m,N}(x_i)|$. So clearly, the total error on N training examples is

$$\epsilon^b_{m,N} = \sum_{n=1}^{N} D^b_m(n)|y_n - h^b_{m,N}(x_n)|.$$

In online boosting, the mth base model's error on example i is the error of Naive Bayes classifier constructed using only the first i training examples: $D^o_m(n)|y_i - h^o_{m,i}(x_i)|$. So the total error on N training examples is

$$\epsilon^o_{m,N} = \sum_{n=1}^{N} D^o_m(n)|y_n - h^o_{m,n}(x_n)|.$$

We are now ready to prove that $\epsilon^o_{m,N} \xrightarrow{P} \epsilon^b_{m,N}$. Since we have $D^o_m(n) \xrightarrow{P} D^b_m(n)$ and $\sum_{n=1}^{N} D^o_m(n) = \sum_{n=1}^{N} D^b_m(n) = 1$, by Lemma 5, we only need to have $h^o_{m,n}(x_n) \xrightarrow{P} h^b_{m,N}(x_n)$ in order to have $\epsilon^o_{m,N} \xrightarrow{P} \epsilon^b_{m,N}$. We have already established $h^o_{m,N}(x_n) \xrightarrow{P} h^b_{m,N}(x_n)$ for all examples x_n. So clearly as $N \to \infty$, $n \to \infty$, and $n \leq N$, the sequence $h^o_{m,n}(x_n)$ (for $n = 1, 2, \ldots$) converges in probability to the sequence $h^b_{m,N}(x_n)$, which is the condition we want. Hence $\epsilon^o_{m,N} \xrightarrow{P} \epsilon^b_{m,N}$. ∎

Theorem 4 *Given the same training set, if $h^o_{m,N}(x)$ and $h^b_{m,N}(x)$ for all $m \in \{1, 2, \ldots, M\}$ are Naive Bayes classifiers, then $h^o(x) \xrightarrow{P} h^b(x)$.*

Proof: We first prove the convergence of the base models and their errors by induction on m. For the base case, we show that $D^o_1 \xrightarrow{P} D^b_1$. This lets us show that $h^o_{1,N}(x) \xrightarrow{P} h^b_{1,N}(x)$ and $\epsilon^o_{1N} \xrightarrow{P} \epsilon^b_{1N}$ as $N \to \infty$. For the inductive part, we show that if $D^o_m \xrightarrow{P} D^b_m$, then $h^o_{m,N}(x) \xrightarrow{P} h^b_{m,N}(x)$ and $\epsilon^o_{m,N} \xrightarrow{P} \epsilon^b_{m,N}$. From these facts, we get $D^o_{m+1} \xrightarrow{P} D^b_{m+1}$, which lets us show that $h^o_{m+1,N}(x) \xrightarrow{P} h^b_{m+1,N}(x)$ and $\epsilon^o_{(m+1)N} \xrightarrow{P} \epsilon^b_{(m+1)N}$ as $N \to \infty$. All of these facts are sufficient to show that the classification functions $h^b(x)$ and $h^o(x)$ converge.

We already have $D^o_1 \xrightarrow{P} D^b_1$ by Lemma 2 (recall that the first training set distributions in the boosting algorithms are the same as the training set distributions in the bagging algorithms). By Lemma 8, we have $h^o_{1,N}(x) \xrightarrow{P} h^b_{1,N}(x)$. That is, the first online Naive Bayes classifier converges to the first batch Naive Bayes classifier. By Lemma 9, $\epsilon^o_{1N} \xrightarrow{P} \epsilon^b_{1N}$. We have thus proven the base case.

Now we prove the inductive portion. That is, we assume that $D^o_m \xrightarrow{P} D^b_m$, which means that $h^o_{m,N}(x) \xrightarrow{P} h^b_{m,N}(x)$ (by Lemma 8) and $\epsilon^o_{m,N} \xrightarrow{P} \epsilon^b_{m,N}$ (by Lemma 9). Given D^o_m and $\epsilon^o_{m,N}$, we can calculate D^o_{m+1}. Given D^b_m and $\epsilon^b_{m,N}$, we can calculate D^b_{m+1}.

By the way the algorithms are set up, we have

$$D^o_{m+1}(n) = D^o_m(n) \left[\left(\frac{1}{2\epsilon^o_{m,n}} \right)^{|y_n - h^o_{m,n}(x_n)|} \left(\frac{1}{2(1-\epsilon^o_{m,n})} \right)^{1-|y_n - h^o_{m,n}(x_n)|} \right],$$

$$D^b_{m+1}(n) = D^b_m(n) \left[\left(\frac{1}{2\epsilon^b_{m,N}} \right)^{|y_n - h^b_{m,N}(x_n)|} \left(\frac{1}{2(1-\epsilon^b_{m,N})} \right)^{1-|y_n - h^b_{m,N}(x_n)|} \right].$$

Because $\epsilon^o_{m,N} \xrightarrow{P} \epsilon^b_{m,N}$, $h^o_{m,n}(x_n) \xrightarrow{P} h^b_{m,N}(x_n)$, and our inductive assumption $D^o_m(n) \xrightarrow{P} D^b_m(n)$, and both $D^o_{m+1}(n)$ and $D^b_{m+1}(n)$ are continuous functions in these quantities that converge, we have that $D^o_{m+1}(n) \xrightarrow{P} D^b_{m+1}(n)$ as long as $\epsilon^o_{m,N}$ and $\epsilon^b_{m,N}$ are bounded away from 0 and 1. This is a reasonable assumption because, if the error is ever 0 or 1, then the algorithm stops and returns the ensemble constructed so far, so the algorithm would never even calculate the training set distribution for the next base model.

$D^o_{m+1}(n) \xrightarrow{P} D^b_{m+1}(n)$ implies that $h^o_{m+1}(x) \xrightarrow{P} h^b_{m+1}(x)$ (by Lemma 8), which in turn implies that $\epsilon^o_{(m+1)N} \xrightarrow{P} \epsilon^b_{(m+1)N}$ (by Lemma 9). So by induction, we have that $h^o_{m,N}(x) \xrightarrow{P}$

$h^b_{m,N}(x)$ and $\epsilon^o_{m,N} \xrightarrow{P} \epsilon^b_{m,N}$ for all $m \in \{1, 2, \ldots, \}$. By Lemma 4, we have $I(h^o_{m,N}(x) = c) \xrightarrow{P} I(h^b_{m,N}(x) = c)$ for all $c \in Y$. This means that $\sum_{m\in\{1,2,\ldots,M\}} I(h^o_{m,N}(x) = c) log\frac{1-\epsilon^o_{m,N}}{\epsilon^o_{m,N}} \xrightarrow{P} \sum_{m\in\{1,2,\ldots,M\}} I(h^b_{m,N}(x) = c) log\frac{1-\epsilon^b_{m,N}}{\epsilon^b_{m,N}}$. Therefore, the order of the terms $\sum_{m\in\{1,2,\ldots,M\}} I(h^o_{m,N}(x) = c) log\frac{1-\epsilon^o_{m,N}}{\epsilon^o_{m,N}}$ and $\sum_{m\in\{1,2,\ldots,M\}} I(h^b_{m,N}(x) = c) log\frac{1-\epsilon^b_{m,N}}{\epsilon^b_{m,N}}$ for each class c is preserved. the final functions $h^o(x)$ and $h^b(x)$ converge. ∎

We have shown that online boosting with Naive Bayes base models produces a classification function that converges to that produced by batch boosting with Naive Bayes base models. In particular, we rely on showing that convergence of the weights of the training examples implies convergence of the base models. This requires that the online base model learning algorithm be lossless, which is true for Naive Bayes classifiers. In our proofs we used the normalized version of the weights used in online boosting. We got away with this because, as we saw in Chapter 3, the Naive Bayes classifier has proportional online and batch learning algorithms, i.e., scaling the weight vector does not change a Naive Bayes classifier. We also used the fact that the Naive Bayes classifier has a functional form that is relatively easy to write down and does not change significantly with additional training examples. In contrast, a decision tree classifier would be very difficult to write down because, at each node of the decision tree, there are many attributes to choose from and a relatively complicated method of choosing them. Additionally, the structure of a decision tree can change substantially with the addition of new examples.

4.6 Experimental Results

In this section, we discuss the results of experiments to compare the performances of boosting, online boosting, and the base model learning algorithms on the same datasets as in the previous chapter. We allow batch and online boosting to generate a maximum of 100 base models.

Table 4.1: Results (fraction correct): batch and online boosting (with Decision Trees). Boldfaced/italicized results are significantly better/worse than single decision trees.

Dataset	Decision Tree	Bagging	Online Bagging	Boosting†	Online Boosting	Primed Online Boosting
Promoters	0.7837	**0.8504**	**0.8613**	**0.9097**	0.7671†	0.7920†
Balance	0.7664	**0.8161**	**0.8160**	*0.7354*	*0.7467*	0.7651
Breast Cancer	0.9531	**0.9653**	**0.9646**	**0.9729**	**0.9679**	**0.9679**
German Credit	0.6929	**0.7445**	**0.7421**	**0.7396**	*0.6773*†	0.6933†
Car Evaluation	0.9537	**0.9673**	**0.9679**	**0.9664**	**0.9639**	**0.9651**
Chess	0.9912	**0.9938**	**0.9936**	**0.9950**	*0.9860*†	*0.9868*†
Mushroom	1.0	1.0	1.0	1.0	0.9999	1
Nursery	0.9896	**0.9972**	**0.9973**	*0.9743*	*0.9821*†	*0.9817*

4.6.1 Accuracy Results

The results of our experiments are shown in Tables 4.1, 4.2, 4.3, 4.4, 4.5, and 4.6. Many column headings in these tables have the names of algorithms with symbols such as "†" and "‡" next to them. Results for a dataset that have such symbols next to them are significantly different from the result on the same dataset of the algorithm associated with that symbol. For example, in Table 4.1, any result with a "†" next to it is significantly different from boosting's result on the same dataset. With decision trees (Table 4.1), online boosting performed significantly worse than batch boosting on the Promoters, German Credit, and Chess datasets. For the remaining datasets, batch and online boosting performed comparably. In case of the Balance and Nursery datasets, both boosting algorithms performed significantly worse than a single decision tree; on Mushroom, they performed comparably; while on Breast Cancer and Car Evaluation they performed significantly better. Figure 4.6 gives a scatterplot comparing the errors of batch and online boosting on the different datasets.

The results of running with Naive Bayes classifiers are shown in Table 4.2. A scatterplot comparing the test errors of batch and online boosting is shown in Figure 4.7. Batch boosting significantly outperforms online boosting in many cases—especially the smaller datasets. However, the performances of boosting and online boosting relative to a single

Table 4.2: Results (fraction correct): batch and online boosting (with Naive Bayes). Bold-faced/italicized results are significantly better/worse than single Naive Bayes classifiers.

Dataset	Naive Bayes	Bagging	Online Bagging	Boosting†	Online Boosting‡	Primed Online Boosting
Promoters	0.8774	*0.8354*	*0.8401*	*0.8455*	*0.7136*†	*0.8218*‡
Balance	0.9075	0.9067	0.9072	*0.8754*	*0.8341*†	*0.8451*†
Breast Cancer	0.9647	**0.9665**	**0.9661**	*0.9445*	*0.9573*†	*0.9546*†
German Credit	0.7483	0.748	0.7483	*0.735*	*0.6879*†	*0.6911*†
Car Evaluation	0.8569	*0.8532*	*0.8547*	**0.9017**	**0.8967**†	**0.8931**†
Chess	0.8757	0.8759	*0.8749*	**0.9517**	**0.9476**†	**0.9490**
Mushroom	0.9966	0.9966	0.9966	**0.9999**	**0.9987**†	**0.9993**†‡
Nursery	0.9031	0.9029	*0.9027*	**0.9163**	**0.9118**†	**0.9144**‡
Connect4	0.7214	0.7212	**0.7216**	*0.7197*	*0.7209*	0.7209
Synthetic-1	0.4998	0.4996	0.4997	**0.5068**	0.5007†	0.4996†
Synthetic-2	0.7800	0.7801	0.7800	**0.8446**	**0.8376**†	**0.8366**†
Synthetic-3	0.9251	0.9251	0.9251	**0.9680**	**0.9688**	**0.9720**
Census-Income	0.7630	**0.7637**	**0.7636**	**0.9365**	**0.9398**	**0.9409**
Forest Covertype	0.6761	0.6762	0.6762	0.6753	0.6753	0.6753

Naive Bayes classifier agree to a remarkable extent, i.e., when one of them is significantly better or worse than a single Naive Bayes classifier, the other one tends to be the same way.

The results of our experiments with decision stumps are shown in Table 4.3. Batch boosting significantly outperforms online boosting on some of the smaller datasets but otherwise their performances are comparable, as seen in Figure 4.8. Both batch boosting and online boosting do not improve upon decision stumps as much as they do upon Naive Bayes—especially on the larger datasets.

Table 4.4 contains the results of running the batch algorithms with neural networks. Recall that, in these algorithms, each neural network is trained by running through the training set ten times (epochs). Table 4.5 and Table 4.6 show the results of running our online algorithms with one and ten updates per training example, respectively, but the algorithms pass through the training set only once. We can see from the tables and from the scatterplots of batch and online boosting (Figure 4.9 and Figure 4.10) that online boosting performs worse than batch boosting. Online boosting with ten updates per training example

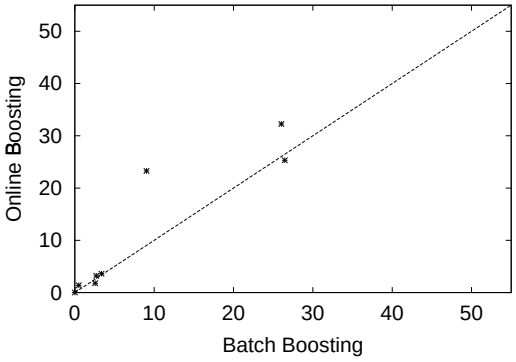

Figure 4.6: Test Error Rates: Batch Boosting vs. Online Boosting with decision tree base models.

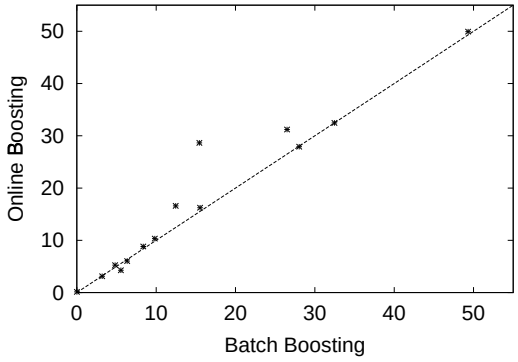

Figure 4.7: Test Error Rates: Batch Boosting vs. Online Boosting with Naive Bayes base models.

performs a little bit better than the one update version. However, in terms of performance relative to its corresponding single model, online boosting with one update performs better. As we discussed in Section 4.4, there is loss associated both with passing through the training set only once and performing online updates with each training example based on the performances of the base models trained on only the examples seen earlier. We can examine the loss associated with each of these factors through our experiments with batch boosting with neural network base models trained using our online method—that is, using only one pass through the dataset, but with one or ten updates per training example. The results are given in the last columns of Table 4.5 and Table 4.6. As expected, most of these performances are between those of batch boosting, which trains each neural network base models with ten passes through the dataset instead of just one, and online boosting, which does not generate each base model with the entire training set before generating the next model. Overall, the loss due to online learning is higher than the loss associated with training each neural network using fewer passes through the training set. In the next section we explore a way of alleviating some of the loss suffered by online boosting.

4.6.2 Priming

In this section, we discuss our experiments with priming the online boosted ensemble by training with some initial part of the training set in batch mode and then training

Table 4.3: Results (fraction correct): batch and online boosting (with decision stumps). Boldfaced/italicized results are significantly better/worse than single decision stumps.

Dataset	Decision Stump	Bagging	Online Bagging	Boosting†	Online Boosting‡	Primed Online Boosting
Promoters	0.7710	**0.8041**	**0.8113**	**0.8085**	*0.7380*†	*0.7118*†
Balance	0.5989	**0.7170**	**0.7226**	**0.7354**	**0.7114**	**0.6595**†‡
Breast Cancer	0.8566	0.8557	0.8564	0.8573	*0.7861*†	*0.834*†‡
German Credit	0.6862	0.6861	0.6862	**0.7291**	*0.6613*†	**0.6961**†‡
Car Evaluation	0.6986	0.6986	0.6986	0.6986	0.6986	0.6986
Chess	0.6795	0.6798	0.6792	**0.8017**	**0.7597**†	**0.7422**†
Mushroom	0.5617	0.5617	0.5617	0.5604	*0.5423*†	*0.5507*†‡
Nursery	0.4184	0.4185	0.4177	*0.3329*	*0.3329*	*0.3329*
Connect4	0.6581	0.6581	0.6581	0.6581	0.6581	0.6581
Synthetic-1	0.5002	0.4996	0.4994	0.4993	0.4987	0.5016
Synthetic-2	0.8492	0.8492	0.8492	0.8492	0.8492	0.8492
Synthetic-3	0.9824	0.9824	0.9824	0.9824	0.9824	0.9824
Census-Income	0.9380	0.9380	0.9380	0.9380	0.9380	0.9380
Forest Covertype	0.6698	0.6698	0.6698	0.6698	0.6698	0.6698

with the remainder of the training set in online mode. The results of running primed online boosting with decision trees, Naive Bayes, decision stumps, and neural networks are shown in Tables 4.1, 4.2, 4.3, 4.5, and 4.6. In our experiments, we trained in batch mode using the *lesser* of the first 20% of the training set or the first 10000 examples. Comparing the error scatterplots in Figure 4.6 and Figure 4.11, we can see that priming slightly helped online boosting with decision trees perform more like batch boosting. Figure 4.12 shows a scatterplot that gives a direct comparison between the unprimed and primed versions and confirms the primed version's slightly better performance. Figures 4.7, 4.13, and 4.14 show that priming helped online boosting with Naive Bayes to some extent as well. Online boosting with decision stumps was not helped by priming as much: Figure 4.8 and Figure 4.15 look quite similar, and Figure 4.16 confirms that priming did not help decision stumps so much. Priming improved online boosting with neural networks tremendously. Figures 4.9, 4.17, and 4.18 show that primed online boosting improved substantially over the original online boosting with neural networks trained using one update per training example. Fig-

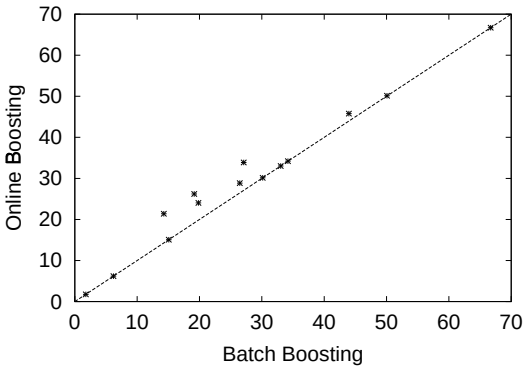

Figure 4.8: Test Error Rates: Batch Boosting vs. Online Boosting with decision stump base models.

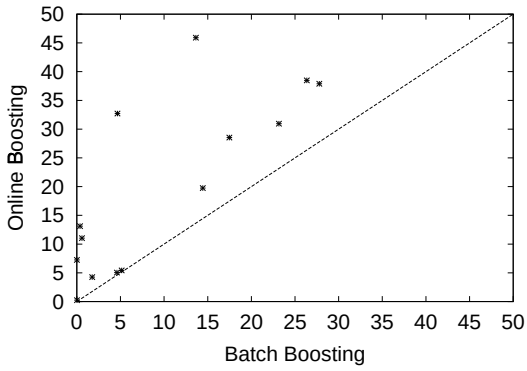

Figure 4.9: Test Error Rates: Batch Boosting vs. Online Boosting (one update per example) with neural network base models.

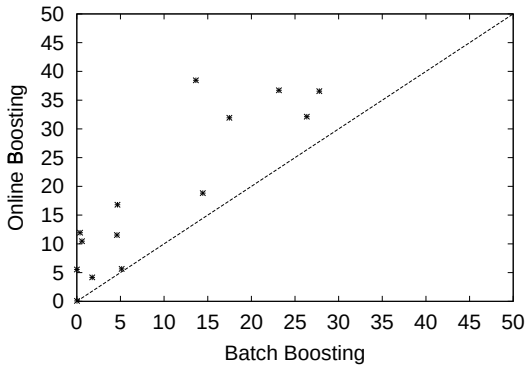

Figure 4.10: Test Error Rates: Batch Boosting vs. Online Boosting (10 updates per example) with neural network base models.

ures 4.9, 4.19, and 4.20 demonstrate the same for the primed and original online boosting algorithms with ten-update neural networks. Thus, by priming, we are able to overcome a large portion of the loss due to online learning.

4.6.3 Base Model Errors

The errors of the base models, which are the errors $\epsilon_1, \epsilon_2, \ldots$ in the boosting algorithms (Figure 4.1 and Figure 4.3), determine much of the algorithms' behavior. These errors not only reflect the accuracies of the base models, but they affect the weights of the training

Table 4.4: Results (fraction correct): batch algorithms (with neural networks). Bold-faced/italicized results are significantly better/worse than single neural networks.

Dataset	Neural Network¶	Bagging	Boosting†
Promoters	0.8982	0.9036	*0.8636*
Balance	0.9194	**0.9210**	**0.9534**
Breast Cancer	0.9527	**0.9561**	0.9540
German Credit	0.7469	0.7495	*0.7365*
Car Evaluation	0.9422	**0.9648**	**0.9963**
Chess	0.9681	**0.9827**	**0.9941**
Mushroom	1.0	1.0	0.9998
Nursery	0.9829	*0.9743*	**0.9999**
Connect4	0.8199	**0.8399**	**0.8252**
Synthetic-1	0.7217	**0.7326**	0.7222
Synthetic-2	0.8564	**0.8584**	0.8557
Synthetic-3	0.9824	0.9824	0.9824
Census-Income	0.9519	**0.9533**	0.9486
Forest Covertype	0.7573	**0.7787**	**0.7684**

examples and the weights of the base models themselves in the final ensemble. Therefore, the closer these errors are in batch and online boosting, the more similar the behaviors of these algorithms. In this section, we compare the errors of the base models generated by the boosting algorithms for some of our experiments. Specifically, for our five largest datasets (the three synthetic datasets, Census Income, and Forest covertype), we compare the errors of batch, online, and primed online boosting with Naive Bayes and neural network base models (with one update per training example for the online algorithms).

Figures 4.21, 4.22, 4.23, 4.24, and 4.25 show the average errors on the training sets of the consecutive base models in batch, online, and primed online boosting with Naive Bayes base models for the three synthetic datasets, Census Income, and Forest Covertype, respectively. We depict the errors for the maximum number of base models generated by the batch boosting algorithm. For example, on the Census Income dataset, no run of batch boosting ever generated more than 22 base models. This happens because of the condition given in step 3 of Figure 4.1—if the next base model that is generated has error greater than 0.5, then the algorithm stops. Note that our online boosting algorithm always

Table 4.5: Results (fraction correct): online algorithms (with neural networks trained with one update step per training example). Boldfaced/italicized results are significantly better/worse than single online neural networks. Results marked "¶" are significantly different from single batch neural networks (Table 4.4). Results marked "†" are significantly different from batch boosting (Table 4.4).

Dataset	Online Neural Net	Online Bagging	Online Boosting‡	Primed Online Boosting§	Boosting One Update
Promoters	0.5018¶	0.5509	0.5409¶†	**0.7791**¶†‡	**0.7018**¶†‡§
Balance	0.6210¶	**0.6554**	0.6728¶†	**0.8845**¶†‡	0.5894¶†‡§
Breast Cancer	0.9223¶	0.9221	**0.9496**	0.9266¶†‡	**0.9373**¶†‡
German Credit	0.7479¶	*0.6994*	*0.6152*¶†	**0.7074**¶†‡	*0.6969*¶†‡
Car Evaluation	0.8452¶	0.8461	**0.8688**¶†	**0.9327**¶†‡	**0.9153**¶†‡§
Chess	0.9098¶	**0.9277**	0.8896¶†	**0.9410**¶†‡	**0.9451**¶†‡
Mushroom	0.9965¶	0.9959	**0.9976**¶†	**0.9996**¶‡	**1**‡
Nursery	0.9220¶	**0.9188**	**0.9275**¶†	**0.9807**†‡	**0.9999**¶‡§
Connect4	0.7717¶	**0.7691**	*0.7147*¶†	**0.7587**¶†‡	0.7693¶†‡
Synthetic-1	0.6726¶	**0.6850**	*0.6211*¶†	0.6709¶†‡	**0.6850**¶†‡§
Synthetic-2	0.8523¶	0.8524	*0.8026*¶†	**0.8298**¶†‡	0.8522¶†‡§
Synthetic-3	0.9824	0.9824	*0.9574*¶†	**0.9652**¶†	0.9824‡§
Census-Income	0.9520	0.9514	*0.9458*	0.9466	0.9471
Forest Covertype	0.7381¶	**0.7416**	*0.6905*¶†	**0.7187**¶†‡	0.7399¶†‡§

generates the full set of 100 base models because, during training, we do not know how the base model errors will fluctuate; however, to classify a new example, we only use the first L base models such that model $L + 1$ has error greater than 0.5. Our primed online boosting algorithm may generate fewer than 100 base models because it essentially runs batch boosting with an initial portion of the training set.

The base model errors of online and batch boosting are quite similar for the synthetic datasets. On Synthetic-1, the base model errors are all close to 0.5, i.e., they perform only slightly better than chance, which explains why boosting does not perform much better than the base models. On Synthetic-2 and Synthetic-3, the first base model performs quite well in both batch and online boosting. Both algorithms then follow the pattern of having subsequent base models perform worse, which is typical because subsequent base models are presented with previously misclassified examples having higher weight, which makes

Table 4.6: Results (fraction correct): online algorithms (with neural networks trained with 10 update steps per training example). Boldfaced/italicized results are significantly better/worse than single online neural networks. Results marked "¶" are significantly different from single batch neural networks (Table 4.4). Results marked "†" are significantly different from batch boosting (Table 4.4).

Dataset	Online Neural Net	Online Bagging	Online Boosting‡	Primed Online Boosting§	Boosting Ten Updates
Promoters	0.8036¶	**0.7691**	*0.6155*¶†	*0.7600*¶†‡	**0.8855**¶†‡§
Balance	0.8965¶	**0.9002**	*0.8320*¶†	*0.8622*¶†‡	0.8970¶†‡§
Breast Cancer	0.9020¶	0.8987	*0.8847*¶†	*0.8810*¶†	**0.9197**¶†‡§
German Credit	0.7062¶	**0.7209**	*0.6788*¶†	*0.6821*¶†	**0.7298**¶‡§
Car Evaluation	0.8812¶	**0.8877**	0.8806¶†	**0.9148**¶†‡	**0.9895**¶†‡§
Chess	0.9023¶	**0.9185**	0.8954¶†	**0.9216**¶†‡	**0.9616**†‡§
Mushroom	0.9995¶	**0.9988**	0.9993¶†	**0.9998**¶‡	**0.9999**‡
Nursery	0.9411¶	0.9396	0.9445¶†	**0.9878**¶†‡	**0.9999**¶‡§
Connect4	0.7042¶	**0.7451**	0.6807¶†	0.7059¶†‡	**0.7330**¶†‡§
Synthetic-1	0.6514¶	**0.6854**	*0.6344*¶†	*0.6333*¶†	**0.6865**¶†‡§
Synthetic-2	0.8345¶	**0.8508**	*0.8117*¶†	*0.8127*¶†	0.8397¶†‡§
Synthetic-3	0.9811¶	**0.9824**	*0.9583*¶†	*0.9638*¶†	**0.9822**¶†‡§
Census-Income	0.9487¶	0.9487	0.9435¶	0.9471¶	0.9465
Forest Covertype	0.6974¶	**0.7052**	*0.6329*¶†	*0.6645*¶†‡	0.7044¶†‡§

their learning problems more difficult. Primed online boosting follows boosting's pattern more closely than the unprimed version does, especially on Synthetic-3. In case of the Census Income dataset, the performances of the base models also follow the general trend of increasing errors for consecutive base models, although more loosely. On the Forest Covertype dataset, the base model errors are quite similar in all three algorithms.

Figures 4.26, 4.27, 4.28, 4.29, and 4.30 show the average errors in batch and online boosting with neural network base models. On the Forest Covertype dataset, primed online boosting never produced more than nine base models, but the performances of those base models mirrored those of the unprimed version. For the other four datasets, the primed version's graph lies between those of batch and online boosting. The reader may have noticed that the base model errors of the boosting algorithms have larger differences for Synthetic-3 and Census Income than for the other three datasets even though the test set

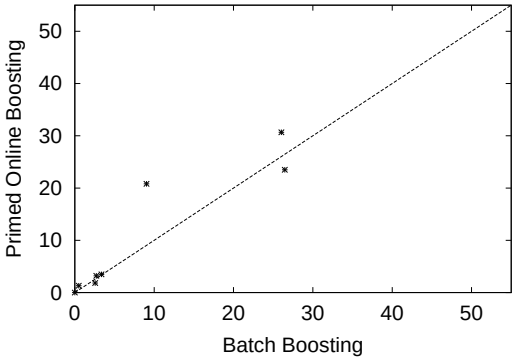

Figure 4.11: Test Error Rates: Batch Boosting vs. Primed Online Boosting with decision tree base models.

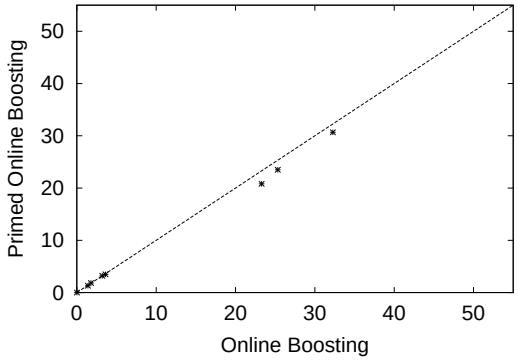

Figure 4.12: Test Error Rates: Online Boosting vs. Primed Online Boosting with decision tree base models.

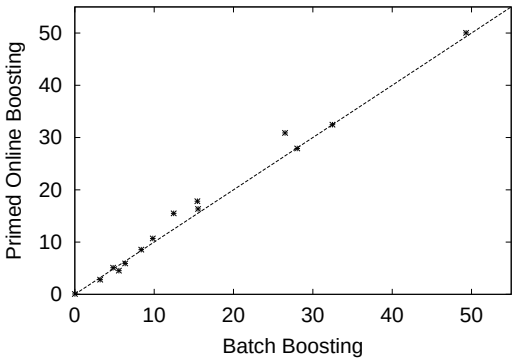

Figure 4.13: Test Error Rates: Batch Boosting vs. Primed Online Boosting with Naive Bayes base models.

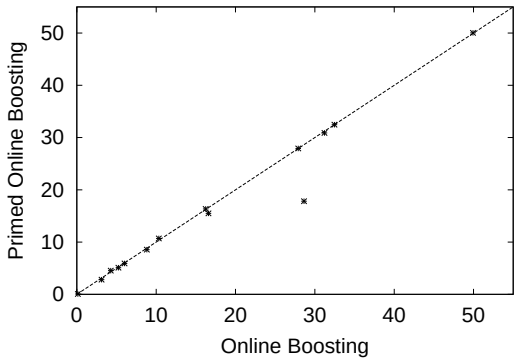

Figure 4.14: Test Error Rates: Online Boosting vs. Primed Online Boosting with Naive Bayes base models.

performances are more similar for Synthetic-3 and Census Income. This is largely because of the similar and very good performances of the first few base models on Synthetic-3 and Census Income—because their errors are very low, their weights in the ensembles are quite high, so the remaining base models get relatively low weight. Therefore, the fact that the remaining base models' errors are very different under the different boosting algorithms does not affect the ensembles' classifications of test cases.

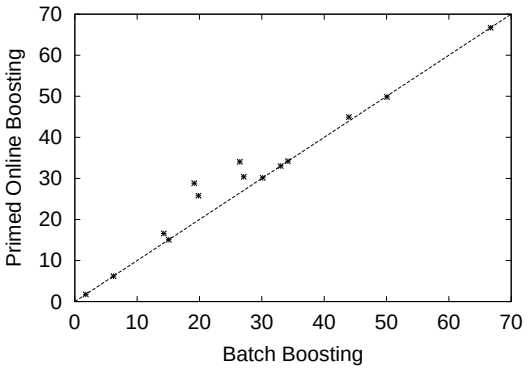

Figure 4.15: Test Error Rates: Batch Boosting vs. Primed Online Boosting with decision stump base models.

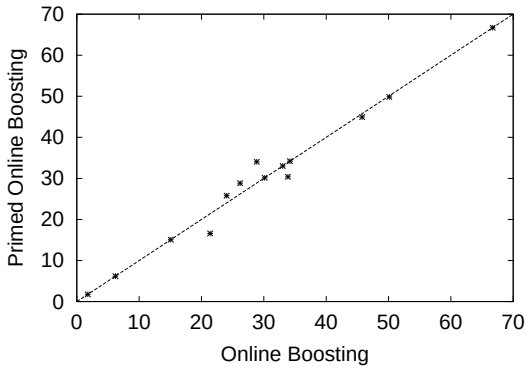

Figure 4.16: Test Error Rates: Online Boosting vs. Primed Online Boosting with decision stump base models.

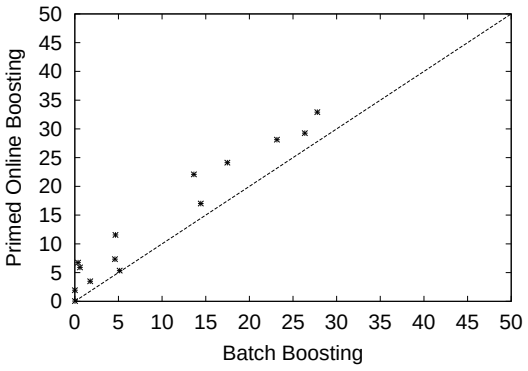

Figure 4.17: Test Error Rates: Batch Boosting vs. Primed Online Boosting with neural network base models (one update per example).

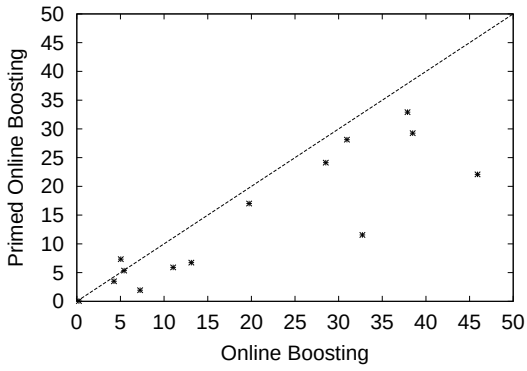

Figure 4.18: Test Error Rates: Online Boosting vs. Primed Online Boosting with neural network base models (one update per example).

4.6.4 Running Times

The running times of all the algorithms that we experimented with are shown in Tables 4.7, 4.8, 4.9, 4.10, 4.11, and 4.12. There are several factors that give batch boosting, online boosting, and primed online boosting advantages in terms of running time.

Batch boosting has the advantage that it can generate fewer than the 100 base models that we designate as the maximum. Recall that, if a base model has error more than 0.5 on the current weighted training set after learning, then boosting stops. We do not have this luxury with online boosting because initially, many of the base models may have high

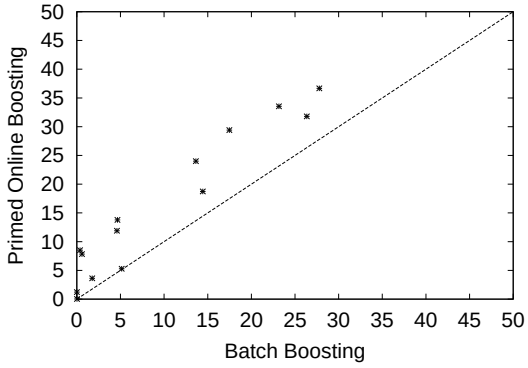

Figure 4.19: Test Error Rates: Batch Boosting vs. Primed Online Boosting with neural network base models (10 updates per example).

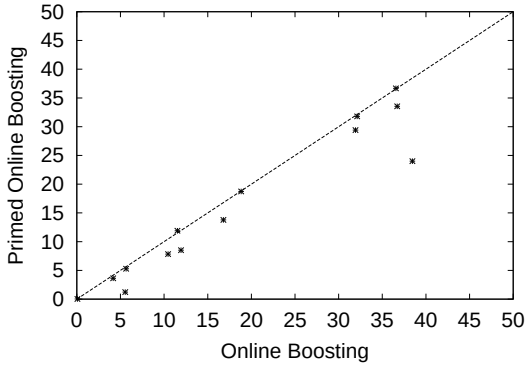

Figure 4.20: Test Error Rates: Online Boosting vs. Primed Online Boosting with neural network base models (10 updates per example).

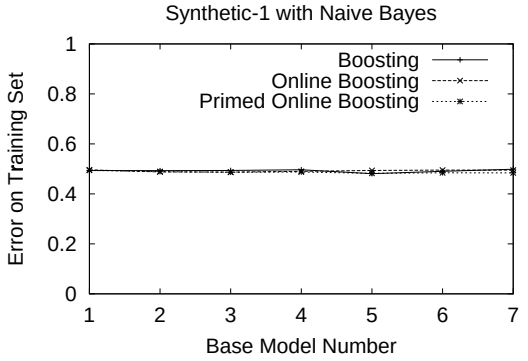

Figure 4.21: Naive Bayes Base Model Errors for Synthetic-1 Dataset

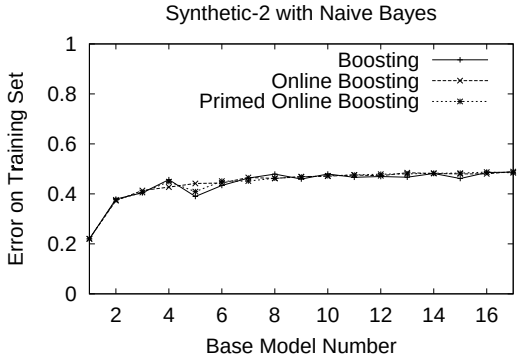

Figure 4.22: Naive Bayes Base Model Errors for Synthetic-2 Dataset

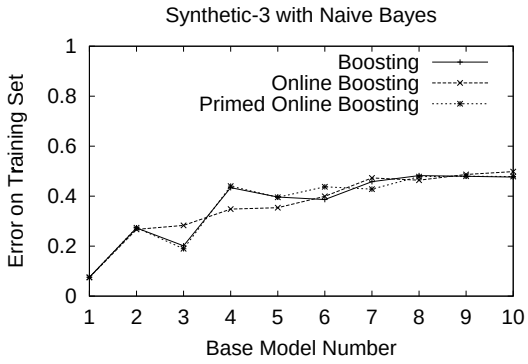

Figure 4.23: Naive Bayes Base Model Errors for Synthetic-3 Dataset

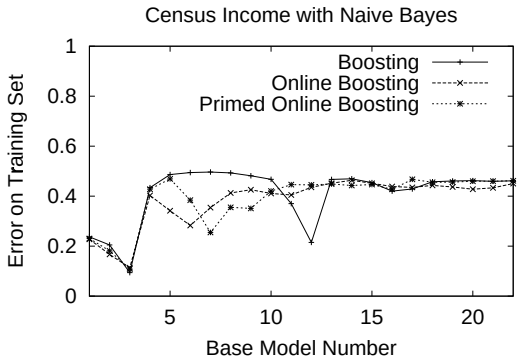

Figure 4.24: Naive Bayes Base Model Errors for Census Income Dataset

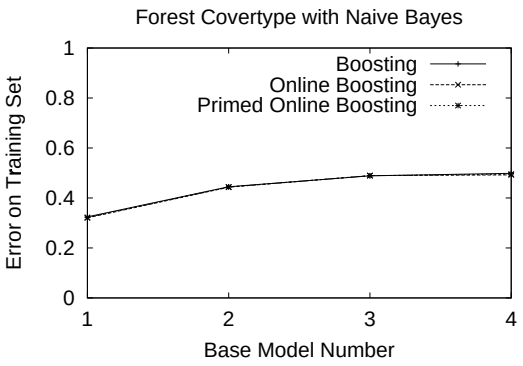

Figure 4.25: Naive Bayes Base Model Errors for Forest Covertype Dataset

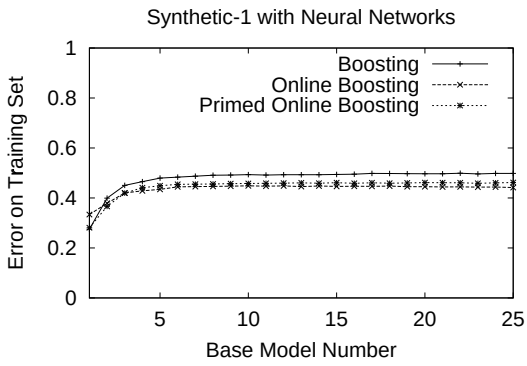

Figure 4.26: Neural Network Base Model Errors for Synthetic-1 Dataset

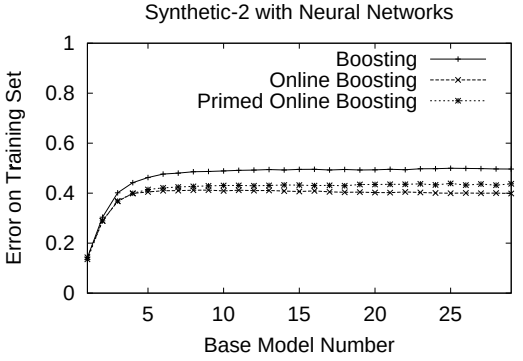

Figure 4.27: Neural Network Base Model Errors for Synthetic-2 Dataset

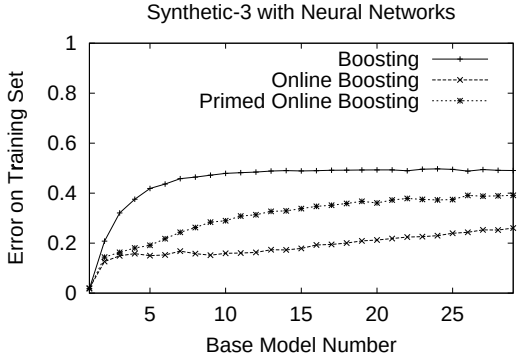

Figure 4.28: Neural Network Base Model Errors for Synthetic-3 Dataset

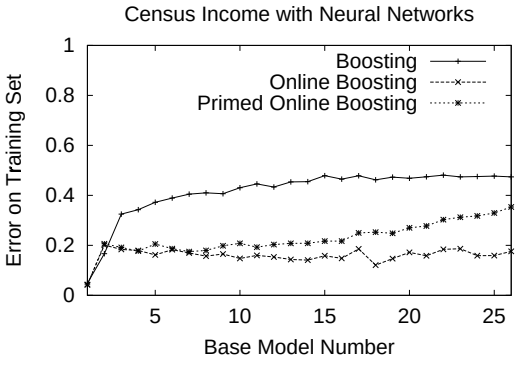

Figure 4.29: Neural Network Base Model Errors for Census Income Dataset

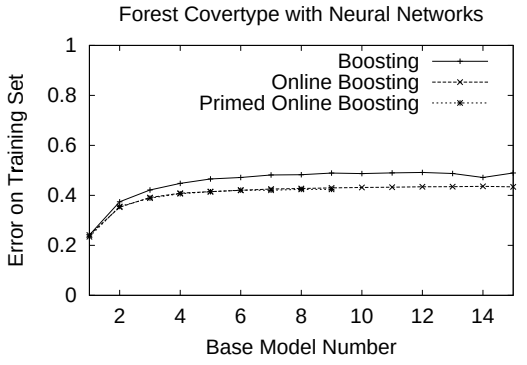

Figure 4.30: Neural Network Base Model Errors for Forest Covertype Dataset

Table 4.7: Running Times (seconds): batch and online boosting (with Decision Trees)

Dataset	Decision Tree	Bagging	Online Bagging	Boosting	Online Boosting	Primed Online Boosting
Promoters	0.16	1.82	1.88	3.56	3.5	0.71
Balance	0.18	1.66	1.96	4.76	25.28	37.64
Breast Cancer	0.18	1.88	2.28	6.12	7.24	7.55
German Credit	0.22	8.98	9.68	17.86	312.52	448.33
Car Evaluation	0.3	4.28	4.28	14.9	53.56	55.09
Chess	0.98	20.4	20.22	82.7	176.92	244.94
Mushroom	1.12	22.6	23.68	162.42	30.12	80.63
Nursery	1.22	29.68	32.74	343.28	364.76	371.92

errors but these errors may decrease with additional training examples. With primed online boosting, we regain this luxury because, when the algorithm is running in batch mode, it may choose to generate fewer base models for the same reason that batch boosting might, in which case the subsequent online learning is also faster because fewer base models need to be updated. Therefore primed online boosting has the potential to be faster than the unprimed version.

Online boosting clearly has the advantage that it only needs to sweep through the training set once, whereas batch boosting needs to cycle through it $M(T+1)$ times, where M is the number of base models, T is the number of times the base model learning algorithm needs to pass through the data to learn it, and one additional pass is needed for each base model to test itself on the training set—recall that boosting needs this step to calculate each base model's error on the training set ($\epsilon_1, \epsilon_2, \ldots$) which is used to calculate the new weights of the training examples and the weight of each base model in the ensemble. Primed online boosting has a slight disadvantage to the unprimed version in this regard because the primed version performs batch training with an initial portion of the training set, which means that the primed version needs more passes through that initial portion than the unprimed version.

The running times of the base model learning algorithms also clearly affect the running times of the ensemble algorithms. Given a fixed training set, online decision tree and online

Table 4.8: Running Times (seconds): batch and online boosting (with Naive Bayes)

Dataset	Naive Bayes	Bagging	Online Bagging	Boosting	Online Boosting	Primed Online Boosting
Promoters	0.02	0	0.22	0.44	0.72	0.26
Balance	0	0.1	0.1	0.26	0.06	0.24
Breast Cancer	0.02	0.14	0.32	1.32	0.66	0.42
German Credit	0	0.14	0.38	0.7	1.5	0.86
Car Evaluation	0.04	0.34	0.44	0.88	1.72	0.06
Chess	0.42	1.02	1.72	9.42	7.46	2.94
Mushroom	0.38	2.14	3.28	114.04	11.08	18.48
Nursery	0.86	1.82	3.74	31.4	20.74	6.12
Connect-4	6.92	33.98	42.04	646.84	465.28	53
Synthetic-1	7.48	45.6	64.16	1351.76	394.08	21.22
Synthetic-2	5.94	44.78	74.84	5332.52	342.86	93.1
Synthetic-3	4.58	44.98	56.2	3761.92	283.52	78.06
Census-Income	56.6	131.8	157.4	25604.6	1199.8	354
Forest Covertype	106	371.8	520.2	67610.8	15638.2	1322.2

decision stump learning are slower than their batch counterparts because, after each online update, the online versions have to recheck that the attribute tests at each node are still the best ones and alter parts of the tree if they are not. Naive Bayes classifier learning is essentially the same in online and batch mode, so online and batch learning given a fixed training set take the same time. Online neural network learning is clearly faster than batch learning due to fewer passes through the dataset, but as we discussed earlier, this speed can come at the cost of reduced accuracy.

We can see from the tables (Tables 4.10, 4.11, and 4.12) that online boosting and primed online boosting with neural networks are clearly faster than batch boosting, which is consistent with the lower running time of online neural network learning relative to batch network learning. Online boosting with Naive Bayes classifiers (Table 4.8) is much faster than batch boosting—the fewer passes required through the dataset clearly outweigh batch boosting's advantage of generating fewer base models. Primed online boosting uses this advantage of generating fewer base models to make itself much faster than online boosting. On the other

Table 4.9: Running Times (seconds): batch and online boosting (with decision stumps)

Dataset	Decision Stump	Bagging	Online Bagging	Boosting	Online Boosting	Primed Online Boosting
Promoters	0.2	0.2	0.3	0.28	1.8	0.30
Balance	0.14	0.14	0.2	0.12	2.74	0.16
Breast Cancer	0.1	0.28	0.3	0.22	4.24	1
German Credit	0.36	0.46	0.56	1.14	11.18	1.52
Car Evaluation	0.1	0.14	0.28	0.5	13.56	0.48
Chess	1.46	15	2.98	62.26	30.6	7.6
Mushroom	0.8	2.04	2.68	25.52	217.46	19.44
Nursery	0.26	19.64	5.1	18.88	236.1	1.42
Connect-4	5.94	53.66	33.72	404.6	2101.56	94.16
Synthetic-1	3.8	26.6	28.02	1634.68	552.46	101.08
Synthetic-2	3.82	30.26	28.34	3489.36	478.5	183.48
Synthetic-3	4.06	29.36	36.04	7772.7	353.76	407.66
Census-Income	51.4	124.2	131.8	231904.2	9071.4	10721.2
Forest Covertype	176.64	594.34	510.58	37510.9	343210.04	10764.04

Table 4.10: Running times (seconds): batch algorithms (with neural networks)

Dataset	Neural Network	Bagging	Boosting	Boosting one updates	Boosting ten updates
Promoters	2.58	442.74	260.86	2.92	335.12
Balance	0.12	12.48	1.96	0.12	1.7
Breast Cancer	0.12	8.14	2.56	0.34	0.62
German Credit	0.72	73.64	11.86	0.6	9.06
Car Evaluation	0.6	36.86	44.04	1.22	41.7
Chess	1.72	166.78	266.74	4.78	25.6
Mushroom	7.68	828.38	91.72	144.08	447.28
Nursery	9.14	1118.98	1537.44	331.26	1369.04
Connect-4	2337.62	156009.3	26461.28	1066.64	6036.94
Synthetic-1	142.02	15449.58	6431.48	1724.96	5211.3
Synthetic-2	300.96	24447.2	10413.66	1651.6	4718.98
Synthetic-3	203.82	17672.84	9261.86	1613.36	5619.82
Census-Income	4221.4	201489.4	89608.4	31587.2	25631.8
Forest Covertype	2071.36	126518.76	141812.4	70638.92	100677.42

Table 4.11: Running times (seconds): online algorithms (with neural networks–one update per example)

Dataset	Online Neural Net	Online Bagging	Online Boosting	Primed Online Boosting
Promoters	<0.02	32.42	26.62	64.46
Balance	0.02	1.48	2.46	0.28
Breast Cancer	0.06	0.94	0.9	0.68
German Credit	0.74	7.98	11.4	2.24
Car Evaluation	0.1	3.92	3.88	10.62
Chess	0.38	20.86	19.1	44.52
Mushroom	1.2	129.18	60.46	152.2
Nursery	1.54	140.02	87.96	297.46
Connect-4	356.12	28851.42	33084.5	2837
Synthetic-1	17.38	2908.48	4018.94	640.34
Synthetic-2	17.74	3467.9	4738.6	559.4
Synthetic-3	24.12	2509.6	1716.38	509.76
Census-Income	249	23765.8	16843.8	10551
Forest Covertype	635.48	17150.24	20662.78	965.26

Table 4.12: Running times (seconds): online algorithms (with neural networks–ten updates per example)

Dataset	Online Neural Net	Online Bagging	Online Boosting	Primed Online Boosting
Promoters	2.34	334.56	83.18	183.82
Balance	0.14	11.7	4.18	0.4
Breast Cancer	0.18	6.58	2.28	1.08
German Credit	0.68	63.5	23.76	5.18
Car Evaluation	0.46	36.82	9.2	15.74
Chess	1.92	159.8	32.42	62.88
Mushroom	6.64	657.48	53.28	171.36
Nursery	9.22	1004.8	160.18	493.46
Connect-4	1133.78	105035.76	58277.4898	6760.76
Synthetic-1	149.34	16056.14	8805.58	1556.38
Synthetic-2	124.22	13327.66	5643.82	1272.32
Synthetic-3	117.54	12469.1	1651.5	832.88
Census-Income	1405.6	131135.2	52362	11009.6
Forest Covertype	805.04	73901.86	74662.66	4164.6

hand, with decision trees (Table 4.7) and decision stumps (Table 4.9), the running times of batch and online boosting are not consistent relative to each other. In those cases where batch boosting generates a small number of base models, batch boosting runs faster. Otherwise, online boosting's advantage of fewer passes through the dataset demonstrates itself with a lower average running time than batch boosting. For example, with decision stumps, online boosting is considerably faster than batch boosting on the Census Income datasets, but the opposite is true on the Forest Covertype dataset. On the Census Income dataset, all the runs of batch boosting generated the full 100 base models just like online boosting always does; therefore, online boosting's advantage of fewer passes through the dataset is demonstrated. However, on the Forest Covertype dataset, the boosting runs generated an average of only 2.76 base models, giving it a significant running time advantage over online boosting. Primed online boosting appears to have the best of both worlds, generating fewer base models where possible and passing through the dataset fewer times. Continuing with the same example, whereas primed online boosting with decision stumps generated an average of 97 base models on the Census Income dataset, it generated only 2.68 base models on the Forest Covertype dataset, giving us batch boosting's advantage of generating fewer base models.

4.6.5 Online Dataset

Table 4.13 shows the results of running all of our algorithms on the Calendar APprentice (CAP) dataset (Mitchell et al., 1994) described in the previous chapter. The accuracies of all the algorithms except primed online boosting are measured over all 1790 appointments in the dataset, where the accuracy on the nth appointment is measured using the hypothesis constructed over the previous $n - 1$ examples. The accuracy of primed online boosting is measured over the last 80% of the data—not the first 20% which was processed in batch mode. Once again, our algorithms perform best with decision tree base models, which is consistent with the choice of learning algorithm in the CAP project. Primed online boosting achieved large improvements over online boosting with decision trees for predicting the day of the week but especially for predicting the meeting duration. Priming also greatly helped with Naive Bayes base models for predicting the day of the week. Just as

with online bagging, our online boosting algorithm has the advantage over the previous method (Mitchell et al., 1994) of not having to select just the right window of past training examples to train with and throw away the results of past learning.

4.7 Summary

In this chapter, we discussed several boosting algorithms including AdaBoost, which is the latest boosting algorithm in common use. We discussed the conditions under which boosting tends to work well relative to single models. We then derived an online version of AdaBoost. We proved the convergence of the ensemble generated by the online boosting algorithm to that of batch boosting for Naive Bayes classifiers. We experimentally compared the two algorithms using several base model types on several "batch" datasets of various sizes and illustrated the performance of online boosting in a calendar scheduling domain—a domain in which data is generated continually. We showed that online boosting and especially primed online boosting often yield combinations of good classification performance and low running times, which make them practical algorithms for use with large datasets.

Table 4.13: Results (fraction correct) on Calendar Apprentice Dataset

Decision Trees				
Target	Single Model	Online Bagging	Online Boosting	Primed Online Boosting
Day	0.5101	0.5536	0.5123	0.5608
Duration	0.6905	0.7453	0.6436	0.7781
Naive Bayes				
Target	Single Model	Online Bagging	Online Boosting	Primed Online Boosting
Day	0.4520	0.3777	0.4665	0.5733
Duration	0.1508	0.1335	0.1626	0.1899
Decision Stumps				
Target	Single Model	Online Bagging	Online Boosting	Primed Online Boosting
Day	0.1927	0.1994	0.1950	0.1899
Duration	0.2626	0.2553	0.1637	0.1750
Neural Networks (one update per example)				
Target	Single Model	Online Bagging	Online Boosting	Primed Online Boosting
Day	0.3899	0.2972	0.2603	0.3736
Duration	0.5106	0.4615	0.4369	0.5433
Neural Networks (ten updates per example)				
Target	Single Model	Online Bagging	Online Boosting	Primed Online Boosting
Day	0.4028	0.4380	0.3575	0.3272
Duration	0.5196	0.5240	0.5330	0.5070

Chapter 5

Conclusions

5.1 Contributions of this Dissertation

The two primary contributions of this thesis are online versions of two of the most popular ensemble learning algorithms: bagging and boosting. To produce an online version of bagging, we had to devise a way to generate bootstrapped training sets in an online manner, i.e., without needing to have the entire training set available to us at all times the way bagging does. We proved that the distribution over bootstrapped training sets under our online scheme converges to that of the batch scheme. We then proved that the online bagged ensemble converges to the batch bagged ensemble when the batch base model learning algorithm and its corresponding online algorithm are proportional learning algorithms that return base classifiers that converge toward each other as the training set grows. We demonstrated this convergence through some experiments with three base models that satisfy the condition of proportionality (decision trees, decision stumps, Naive Bayes classifiers). We also experimented with neural networks, which do not satisfy this condition and for which learning is lossy. We saw that the loss experienced by the online base model learning algorithm leads to loss in online bagging. However, we observed that, for some larger datasets, this loss was rather low and may, in fact, be acceptable given the lower running time of online bagging relative to batch bagging with neural networks.

The running times of batch and online bagging with the other base models were seen to be close to one another. However, the main running time advantage of our online algo-

rithms and of online algorithms in general relative to batch algorithms is in situations in which data is arriving continuously. In these situations, batch algorithms would have to be rerun using the entire previously-used training set plus the new examples, while online algorithms could simply perform incremental updates using the new examples without examining the previously-learned data. Clearly, incremental updating would be much faster than rerunning a batch algorithm on all the data seen so far, and may even be the only possibility if all the data seen so far cannot be stored or if we need to perform online prediction and updating in real time or, at least, very quickly. We experimented with a domain of this type—the calendar scheduling domain. Using online bagging, we were able to obtain improved accuracy over single decision trees which have been used in the past, and we were able to do this through straightforward online learning without having to explicitly forget past examples or use trial and error to choose windows of past examples with which to train.

We then devised an online version of AdaBoost, which is a batch boosting algorithm. We did this by first noting that batch boosting generates the first base model using the same distribution over the training set used in bagging, but then calculates a new distribution over the training set based on the performance of the first base model and uses this new distribution to create a second base model. This process of calculating new distributions over the training set and using them to generate additional base models is repeated as desired. We devised an online boosting algorithm that generates the first base model using the training set distribution used by online bagging, and then calculates a new distribution in a manner much like that used in batch boosting. We explained why online boosting performs differently from batch boosting even when online boosting is able to use a lossless online algorithm to generate its base models. However, we proved that, for Naive Bayes base models, the online boosted ensemble converges to that generated by batch boosting. We presented experiments showing that when we had lossless online algorithms to create the base models (decision trees, decision stumps, and Naive Bayes classifiers), the performances of online boosting were not far off from batch boosting. However, with a lossy online base model learning algorithm (neural networks), online boosting sometimes experienced substantial loss. We devised a version of online boosting that primes the ensemble by training in batch mode using some initial part of the training set and then updates that ensemble online using

the remainder of the training set. This algorithm performed better than the original online boosting algorithm and also ran faster in most cases. We observed the running times of all the algorithms that we experimented with and noted the situations under which online boosting ran faster than batch boosting and vice versa. We experimented with the calendar scheduling domain and obtained even more improvement relative to decision trees using our online boosting algorithms with decision trees than we did with online bagging.

5.2 Future Work

Many of the current active areas of research related to batch ensemble learning also apply to online ensemble learning. One area of research is using additional types of base models such as radial basis function networks. This is necessary in order for ensemble learning methods to be used in a greater variety of problem domains where other types of models are considered standard. Experimental and theoretical results with a greater variety of ensemble methods and base models should also enable us to better describe the precise conditions under which ensemble methods work well.

Most ensemble methods, including bagging and boosting, use some method to bring about differences in the training sets used to generate the base models in the hope that these differences are enough to yield diversity in the pool of base models. A very different approach is used by Merz (1998, 1999)—in this approach, given a set of base models, a combining scheme based on Principal Component Analysis (PCA) (Jolliffe, 1986) is used to try to achieve the best performance that can be achieved from that set of models. This method reduces the weights of base models that are redundant even though they may perform well and increases the weights of base models that, in spite of their poor overall performance, perform well on certain parts of the input space where other base models perform poorly. An online version of this method would be very useful.

Boosting is known to perform poorly when there is noisy data because boosting tends to increase the weights of noisy examples too much at the expense of less noisy data. Correcting this is an active area of research within the boosting community. However, this problem can also be seen as a benefit, in that boosting can be used to detect outliers, which

is an important task in many problem domains. Boosting has not been used for this purpose so far.

In this dissertation, we have mostly used our online ensemble algorithms to learn static datasets, i.e., those which do not have any temporal ordering among the training examples. As explained in Chapter 1, this is an important area of application because online learning algorithms are often the only practical way to learn very large batch datasets. However, online learning algorithms are especially important for time series data. Some work has been done on applying batch boosting to time series classification (Diez & Gonzalez, 2000) and regression (Avnimelech & Intrator, 1998). The work on time series classification assumes that each training example is a time series example of one class. There are other possible time series classification problems. For example, there may be just one time series from some continuously running system (e.g., a factory machine or airplane) and one may want to determine which of several modes the system is in at each point in time. Our online ensemble algorithms or variants of them should be especially useful for this type of problem. Similar to what we discussed in the last paragraph, we might be able to use online boosting to detect outliers within a time series whether the problem is time series classification or regression. Much data mining research is concerned with finding methods applicable to the increasing variety of types of data available—time series, spatial, multimedia, world-wide web logs, etc. Using online learning algorithms—both single-model and ensemble methods—on these different types of data is an important area of work.

When applying a learning algorithm to a large dataset, the computational intensiveness of the algorithm becomes important. In this dissertation, we examined the running times of our algorithms and found that they are quite practical. However, we should be able to do even better. Batch and online bagging train each of their base models independently; therefore, they both can be executed on a parallel computer, which would make them approximately as fast as the base model learning algorithm. Batch boosting produces its base models in sequence; therefore, it clearly cannot be parallelized. However, online boosting can be parallelized. In particular, one can set up one base model learning algorithm on each available processor, and have each base model's processor receive an example and weight from the previous base model's processor, update its own base model, and then pass that example and new weight to the next base model's processor. The current base model's

processor would then be immediately ready to receive the next training example. In our experiments with primed online boosting, we primed using the lesser of the first 20% of the training set or the first 10000 training examples. A version of primed online boosting that can adapt to the available hardware and even adapt to the current processor load and available memory would be useful.

This dissertation provides a stepping stone to using ensemble learning algorithms on large datasets. We hope that this research stimulates others to work on the ideas discussed in this chapter and come up with new ideas to make ensemble learning algorithms more practical for modern data mining problems.

Bibliography

Al-Ghoneim, K., & Vijaya Kumar, B. V. K. (1995). Learning ranks with neural networks (Invited paper). In *Applications and Science of Artificial Neural Networks, Proceedings of the SPIE*, Vol. 2492, pp. 446–464.

Armstrong, R., Freitag, D., Joachims, T., & Mitchell, T. (1995). Webwatcher: A learning apprentice for the world wide web. In *AAAI Spring Symposium on Information Gathering from Heterogeneous Distributed Environments*.

Avnimelech, R., & Intrator, N. (1998). Boosting regression estimators. *Neural Computation*, *11*, 491–513.

Battiti, R., & Colla, A. M. (1994). Democracy in neural nets: Voting schemes for classification. *Neural Networks*, *7*(4), 691–709.

Bauer, E., & Kohavi, R. (1999). An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine Learning*, *36*, 105–139.

Bay, S. (1999). The UCI KDD archive.. (URL: http://kdd.ics.uci.edu).

Benediktsson, J., Sveinsson, J., Ersoy, O., & Swain, P. (1994). Parallel consensual neural networks with optimally weighted outputs. In *Proceedings of the World Congress on Neural Networks*, pp. III:129–137. INNS Press.

Billingsley, P. (1995). *Probability and Measure*. John Wiley and Sons, New York.

Blake, C., Keogh, E., & Merz, C. (1999). UCI repository of machine learning databases.. (URL: http://www.ics.uci.edu/∼mlearn/MLRepository.html).

Blum, A. (1996). On-line algorithms in machine learning. In *Dagstuhl Workshop on Online Algorithms*.

Blum, A. (1997). Empirical support for winnow and weighted-majority based algorithms: results on a calendar scheduling domain. *Machine Learning*, *26*, 5–23.

Breiman, L. (1993). Stacked regression. Tech. rep. 367, Department of Statistics, University of California, Berkeley.

Breiman, L. (1994). Bagging predictors. Tech. rep. 421, Department of Statistics, University of California, Berkeley.

Breiman, L. (1996a). Bagging predictors. *Machine Learning*, *24*(2), 123–140.

Breiman, L. (1996b). Bias, variance and arcing classifiers. Tech. rep. 460, Department of Statistics, University of California, Berkeley.

Breiman, L. (1997). Prediction games and arcing algorithms. Tech. rep. 504, Department of Statistics, University of California, Berkeley.

Breiman, L. (1999). Pasting small votes for classification in large databases and on-line. *Machine Learning*, *36*, 85–103.

Breiman, L. (1998). Arcing classifiers. *The Annals of Statistics*, *26*, 801–849.

Bryson, A., & Ho, Y.-C. (1969). *Applied Optimal Control*. Blaisdell Publishing Co.

Dietterich, T., & Bakiri, G. (1995). Solving multiclass learning problems via error-correcting output codes. *Journal of AI Research*, *2*, 263–286.

Dietterich, T. G. (2000). An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine Learning*, *40*, 139–158.

Diez, J. J. R., & Gonzalez, C. J. A. (2000). applying boosting to similarity literals for time series classification. In *First International Workshop on Multiple Classifier Systems*. Springer-Verlag.

Drucker, H., & Cortes, C. (1996). Boosting decision trees. In Touretzky, D. S., Mozer, M. C., & Hasselmo, M. E. (Eds.), *Advances in Neural Information Processing Systems-8*, pp. 479–485. M.I.T. Press.

Fern, A., & Givan, R. (2000). Online ensemble learning: An empirical study. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pp. 279–286. Morgan Kaufmann.

Freund, Y., & Schapire, R. (1996). Experiments with a new boosting algorithm. In *Proceedings of the Thirteenth International Conference on Machine Learning*, pp. 148–156 Bari, Italy. Morgan Kaufmann.

Freund, Y. (1995). Boosting a weak learning algorithm by majority. *Information and Computation*, *121*(2), 256–285.

Freund, Y., & Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, *55*(1), 119–139.

Grimmett, G. R., & Stirzaker, D. R. (1992). *Probability and Random Processes*. Oxford Science Publications, New York.

Hansen, L. K., & Salamon, P. (1990). Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *12*(10), 993–1000.

Hashem, S., & Schmeiser, B. (1993). Approximating a function and its derivatives using MSE-optimal linear combinations of trained feedforward neural networks. In *Proceedings of the Joint Conference on Neural Networks*, Vol. 87, pp. I:617–620 New Jersey.

Ho, T. K., Hull, J. J., & Srihari, S. N. (1994). Decision combination in multiple classifier systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *16*(1), 66–76.

Jacobs, R. (1995). Method for combining experts' probability assessments. *Neural Computation*, *7*(5), 867–888.

Jolliffe, I. (1986). *Principal Component Analysis*. Springer-Verlag.

Jordan, M. I., & Jacobs, R. A. (1994). Hierarchical mixture of experts and the EM algorithm. *Neural Computation*, *6*, 181–214.

Kearns, M. J., & Vazirani, U. V. (1994). *Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA.

Krogh, A., & Vedelsby, J. (1995). Neural network ensembles, cross validation and active learning. In Tesauro, G., Touretzky, D. S., & Leen, T. K. (Eds.), *Advances in Neural Information Processing Systems-7*, pp. 231–238. M.I.T. Press.

Lincoln, W., & Skrzypek, J. (1990). Synergy of clustering multiple back propagation networks. In Touretzky, D. (Ed.), *Advances in Neural Information Processing Systems-2*, pp. 650–657. Morgan Kaufmann.

Littlestone, N. (1988). Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, *2*, 285–318.

Littlestone, N., & Warmuth, M. (1994). The weighted majority algorithm. *Information and Computation*, *108*, 212–261.

Merhav, N., & Feder, M. (1998). Universal prediction. *IEEE Transactions on Information Theory*, *44*, 2124–2147.

Merz, C. J. (1998). *Classification and Regression by Combining Models*. Ph.D. thesis, University of California, Irvine, Irvine, CA.

Merz, C. J. (1999). A principal component approach to combining regression estimates. *Machine Learning*, *36*, 9–32.

Mitchell, T., Caruana, R., Freitag, D., McDermott, J., & Zabowski, D. (1994). Experience with a learning personal assistant. *Communications of the ACM*, *37*(7), 81–91.

Oza, N. C., & Tumer, K. (1999). Dimensionality reduction through classifier ensembles. Tech. rep. NASA-ARC-IC-1999-126, NASA Ames Research Center.

Oza, N. C., & Tumer, K. (2001). Input decimation ensembles: Decorrelation through dimensionality reduction. In *Second International Workshop on Multiple Classifier Systems*. Springer-Verlag.

Perrone, M., & Cooper, L. N. (1993). When networks disagree: Ensemble methods for hybrid neural networks. In Mammone, R. J. (Ed.), *Neural Networks for Speech and Image Processing*, chap. 10. Chapmann-Hall.

Quinlan, J. (1996). Bagging, boosting and C4.5. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence* Portland, OR.

Rogova, G. (1994). Combining the results of several neural network classifiers. *Neural Networks*, *7*(5), 777–781.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representations by error propagation. In Rumelhart, D. E., & McClelland, J. L. (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Bradford Books/MIT Press, Cambridge, Mass.

Schapire, R. (1990). The strength of weak learnability. *Machine Learning*, *5*(2), 197–227.

Schapire, R. E., Freund, Y., Bartlett, P., & Lee, W. S. (1997). Boosting the margin: A new explanation for the effectiveness of voting methods. In *Proceedings of the Fourteenth International Conference on Machine Learning*. Morgan Kaufmann.

Schapire, R. E., Freund, Y., Bartlett, P., & Lee, W. S. (1998). Boosting the margin: A new explanation for the effectiveness of voting methods. *The Annals of Statistics*, *26*(5), 1651–1686.

Singer, A. C., & Feder, M. (1999). Universal linear prediction by model order weighting. *IEEE Transactions on Signal Processing*, *47*, 2685–2699.

Tumer, K., & Ghosh, J. (1996). Error correlation and error reduction in ensemble classifiers. *Connection Science, Special Issue on Combining Artificial Neural Networks: Ensemble Approaches*, *8*(3 & 4), 385–404.

Tumer, K., & Ghosh, J. (1998). Classifier combining through trimmed means and order statistics. In *Proceedings of the International Joint Conference on Neural Networks*, pp. 757–762 Anchorage, Alaska.

Tumer, K., & Oza, N. C. (1999). Decimated input ensembles for improved generalization. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN-99)*.

Tumer, K. (1996). *Linear and Order Statistics Combiners for Reliable Pattern Classification*. Ph.D. thesis, The University of Texas, Austin, TX.

Utgoff, P., Berkman, N., & Clouse, J. (1997). Decision tree induction based on efficient tree restructuring. *Machine Learning*, *29*(1), 5–44.

Wolpert, D. H. (1992). Stacked generalization. *Neural Networks, 5*, 241–259.

Xu, L., Krzyzak, A., & Suen, C. Y. (1992). Methods of combining multiple classifiers and their applications to handwriting recognition. *IEEE Transactions on Systems, Man and Cybernetics*, *22*(3), 418–435.

Yang, J.-B., & Singh, M. G. (1994). An evidential reasoning approach for multiple-attribute decision making with uncertainty. *IEEE Transactions on Systems, Man, and Cybernetics*, *24*(1), 1–19.

Zheng, Z., & Webb, G. (1998). Stochastic attribute selection committees. In *Proceedings of the 11th Australian Joint Conference on AI (AI'98)*, pp. 321–332.