An Efficient Boosting Algorithm for Combining Preferences

by

Raj Dharmarajan Iyer Jr.

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1999

Author . . . .
Department of Electrical Engineering and Computer Science
August 24, 1999

Certified by . . . .
David R. Karger
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

An Efficient Boosting Algorithm for Combining Preferences

by

Raj Dharmarajan Iyer Jr.

Submitted to the Department of Electrical Engineering and Computer Science
on August 24, 1999, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

The problem of combining preferences arises in several applications, such as combining the results of different search engines. This work describes an efficient algorithm for combining multiple preferences. We first give a formal framework for the problem. We then describe and analyze a new boosting algorithm for combining preferences called RankBoost. We also describe an efficient implementation of the algorithm for certain natural cases. We discuss two experiments we carried out to assess the performance of RankBoost. In the first experiment, we used the algorithm to combine different WWW search strategies, each of which is a query expansion for a given domain. For this task, we compare the performance of RankBoost to the individual search strategies. The second experiment is a collaborative-filtering task for making movie recommendations. Here, we present results comparing RankBoost to nearest-neighbor and regression algorithms.

Thesis Supervisor: David R. Karger
Title: Associate Professor

Acknowledgments

I thank my advisor, David Karger, for wonderfully challenging me and teaching me how to think. I thank Rob Schapire of AT&T Labs for being a shining light of wisdom and kindness ever since I have known him. Thanks to Yoram Singer of AT&T Labs for working side by side with me on this work. Thanks also to Yoav Freund, who patiently taught me during my first summer at AT&T. I would not be where I am without the guidance and nurturing of my undergraduate advisors, Max Mintz and Sampath Kannan of University of Pennsylvania, and Joan Feigenbaum of AT&T Labs. And of course, I would not be here at all if it weren't for the enduring love my parents, Raj Sr. and Patricia, and my brother, Nathan. Finally, I thank my Heavenly Advisor, to whom I am eternally grateful.

Contents

Chapter 1

Introduction

It's Sunday night, and Alice wants to rent a movie. Tired of taking risks with movies that she's never heard of, Alice seeks suggestions of movies that she will enjoy. She calls up her friends, and each one rattles off the list of his of her top ten favorite movies. How should Alice combine their recommendations so that she can determine which movie to rent tonight?

This problem is not as fanciful as it may seem. It is a collaborative-filtering problem [39, 67] and Internet websites such as Movie Critic [4] and MovieFinder [5] are devoted to providing the same service as Alice's friends. When asked to make recommendations, the system uses the preferences of previous users to produce a ranked list of movie suggestions. The question is, how should the system combine these preferences into a single list that will make good recommendations?

This is an example of the *ranking problem*: given a set of instances and a collection of functions that rank the instances, how can we combine the functions to produce an ordering of the instances that approximates their true order? In this dissertation we provide a formal model for the ranking problem as well an algorithm to solve it.

Returning to Alice's dilemma, we see that one way she can combine her friends' movie rankings is to first ask them to rank movies that she has already seen. Once she sees how well each friend agrees with her preferences, she can determine whose suggestions to follow regarding movies she hasn't seen. That is, based on how well her friends rate movies that she has ranked, she can *learn* how to combine her friends' advice for movies that she hasn't ranked. Better yet, Alice can submit a list of her favorite movies to the movie recommendation website and let it do the work of learning how to combine its users' rankings to suggest good movies for her. This is the approach of *machine learning*, which is the basis of our model and algorithm for the ranking problem.

The ranking problem is not specific to Alice; it arises it many places. One example is searching the Internet using multiple web search engines [2, 3]. Suppose that, instead of submitting a query to a single search engine, we submit the query to ten search engines, each of which returns a ranked lists of web pages relevant to our query. How can we combine their ranked lists?

This raises a question about how we represent the ranked lists. Many search engines rank web pages according to a query by computing a numeric similarity score for each web page and then ordering the pages by score. Thus we might hope to combine these rankings by averaging the scores of the pages. Indeed, this is a commonly taken approach [6, 14, 39], which we will explore in Section 2.1. However, this approach can encounter a number of difficulties. First, some search engines such as AltaVista [1] do not reveal their computed similarity scores. Second, different search engines may use different numeric ranges for their scores, rendering meaningless any straightforward combination. Third, and more significant, even if the scores are in the same numeric range, different search engines may use different numbers in the range to express identical measures of quality, or they may use the same numbers to express different measures of quality.

This is easily illustrated in the movie recommendation example: when asked to score movies from one to five stars, some people are extremists and assign movies either one star or five stars, only rarely giving out three stars. Others like every movie they see and give all movies four or five stars. Still others are wishy-washy and give most movies three stars, with an occasional four or two. In such a situation, knowing that a movie received mostly four star ratings is not very informative as an absolute score: we can interpret a person's scoring of a movie only *relative* to the scores that the person assigned to other movies. Our work is specifically designed to address this problem of combining relative preferences instead of absolute scores.

The ranking problem, and the difficulties associated with it, arises in other applications as well. The Internet search engine example is problem from the field of *information retrieval*. One of the primary tasks of information retrieval is searching a large collection of documents for those relevant to particular query: a popular and effective way to present the returned documents to a user is to order the documents according to relevance. Here we use the term "document" rather loosely, as it can mean text, images, audio, or video.

Another application is making predictions based on a ranked list of probable outcomes. For example, automatic speech recognizers receive as input a recorded clip of spoken words and return a list of textual transcriptions ranked according to likelihood. We would like to combine the lists of various recognizers with the goal of increasing their collective accuracy.

Despite the wide range of applications that use and combine rankings, this problem has received relatively little attention in the machine-learning community. The few methods that have been devised for combining rankings are usually based on nearest-neighbor methods [56, 67], regression methods [39] or optimization techniques such as gradient descent [6, 14]. In these cases, the rankings are viewed as real-valued scores and the problem of combining different rankings reduces to a numerical search for a set of parameters that will minimize the disparity between the combined scores and the feedback of a user. As already discussed, these approaches do not guarantee that the combined system will match the user's preference when we view the scores as a means to express preferences.

In this dissertation we introduce and analyze an efficient algorithm called RankBoost for combining multiple rankings. The main difference between the majority of previous work and our work is that we do not use absolute numeric scores to combine the ranking functions. This algorithm is based on Freund and Schapire's [31] AdaBoost algorithm and its recent successor developed by Schapire and Singer [65]. Similar to other boosting algorithms, RankBoost works by combining many rankings of the given instances. Each of these may be only weakly correlated with the target ranking that we are attempting to approximate. We show how to combine such weak rankings into a single highly accurate ranking, and we prove a bound on the quality of this final ranking in terms of the quality of the weak rankings.

For the movie task, we use very simple rankings that partition all movies into only two equivalence sets, those that are more preferred and those that are less preferred. For instance, we might use another user's ranked list of movies partitioned according to whether or not he prefers them to some particular movie that appears on his list. Such partitions of the data have the advantage that they only depend on the relative ordering defined by the given rankings rather than absolute ratings. Despite their apparent weakness, their combination using RankBoost performs quite well experimentally.

Besides giving a theoretical analysis of the quality of the ranking produced by RankBoost, we also analyze its complexity and show how it can be implemented efficiently. We discuss further improvements in efficiency that are possible in certain natural cases.

We report the results of experimental tests of our approach on two different tasks. One is the movie-recommendation problem described above. For this problem, there exists a large publicly available dataset which contains ratings of movies by many different people. We compared RankBoost to nearest-neighbor and regression algorithms that have been previously studied for this application, and we used several measures to evaluate their performance. RankBoost was the clear winner in these experiments.

The other task is the meta-searching problem [24]. In a meta-search application, the goal is to combine the rankings of several WWW search strategies. Examples of meta-search engines include MetaCrawler [3] and Dogpile [2]. We simulated a meta-search engine by creating search strategies, each of which is an operation that takes a query as input, performs some simple transformation of the query (such as adding search directives such as "AND", or search tokens such as "homepage") and sends it to a particular search engine. The outcome of using each strategy is an ordered list of URL's that are proposed as answers to the query. The goal is to combine the strategies that work best for a given set of queries.

The remainder of this dissertation is organized as follows. In Chapter 2 we discuss previous approaches to the ranking problem. In Chapter 3 we introduce the basic ideas of machine learning and then present a history of boosting. Finally, in Chapter 4 we explain in detail our ranking model, boosting algorithm, and theoretical and experimental results.

Chapter 2

The Ranking Problem

In this chapter we explore the background that sets the stage for our boosting algorithm for ranking problems. We begin by introducing the ranking problem and the applications where it arises. We then survey existing approaches to solving the problem.

2.1 The Ranking Problem

As described in the previous chapter, the *ranking problem* asks, given a set of instances and a collection of functions that rank the instances, how we can combine the functions to produce an ordering of the instances that approximates their true ordering. We introduced the ranking problem in the context of the collaborative-filtering [39, 67] task of making movie recommendations. The problem also arises in a number of other areas and applications, including information retrieval [57, 58], automatic speech recognition [55], and human voting. In this section we survey past approaches to the problem and their relationship to our work, as described in Chapter 1.

Despite the wide range of applications that use and combine rankings, this problem has received relatively little attention from the machine-learning community. The methods that have been proposed try to approximate the true ordering of a set of items by assigning scores to each item (the predicted ordering is the order induced by the scores). They do this by searching for the best combination of ranking functions. These methods can be grouped into three general categories based on how they define "best" and how they perform the search. Nearest-neighbor methods define a distance measure between two ranking functions and find the closest one (or ones) by enumerative search. Regression methods view the ranking functions as vectors over the space of instances and then find, by linear algebra, the straight-line fit that best approximates them. Finally, more general methods define some correlation measure between the predicted order and the true order and then search for the best prediction via numerical methods such as gradient descent. We will briefly summarize examples of work based on these methods.

As discussed in the previous chapter, these approaches do not guarantee that the combined system will match the user's preference when we view the scores as a means to express preferences. The main difference between the majority of previous work and our work is that we do not use the absolute numeric scores to combine the ranking functions.

2.2 Nearest-Neighbor Methods

Some researchers [56, 67] have reported results using nearest-neighbor methods to rank items. Shardanand and Maes [67] built a collaborative-filtering system that utilized a database of user preferences to recommend music albums to a new user. Their system received a ranked list of the new user's preferences on various music albums and then found the set of nearest-neighbors to the user and averaged their predictions.

The authors experimented with four user similarity measures based on mean-squared difference and the Pearson r correlation coefficient. The latter measure ignores the scores used in ranking and is proportional to our measure of disagreement. They found that this was the best of the four measures, a finding that supports our choice of the disagreement measure. Of the three approaches, theirs is the most similar in spirit to ours.

2.3 Regression Methods

Regression methods have also been used combine user preferences. Hill et al. [39] presented the earliest work on the collaborative filtering movie recommendation task described in Chapter 1: given Alice's preferences on movies she has seen, examine a database of other users' preferences and recommend movies that Alice hasn't seen. In this task, each user is treated as a ranking function. To create a good combination of these users, Hill et al. first found a subset of users whose preferences are correlated with the target user's preferences. They then combined the scores to make predictions using a regression equation (for more details see Section 5.2.3). This approach differs from ours since regression methods depend on the actual numeric scores used in ranking, and using these scores may not be a good idea, as explained in Chapter 1. We present experiments comparing our algorithm to one similar to theirs in Chapter 5.

2.4 More General Methods

Regression seeks to maximize a certain utility function (Euclidean distance) by setting its parameters via linear algebra. A more general approach is to choose a different utility function and maximize via numerical search. Such an approach to the ranking problem views the set of scores assigned by a ranking function as a vector in a high-dimensional vector space and defines a utility function that takes as input a target vector, a set of vectors that will approximate the target, and some free parameters such as weights to form a linear combination of the set of vectors. Thus the problem of combining different rankings reduces to a numerical search for a set of parameters that will maximize the utility function. [6, 14]. Bartell, Cottrell, and Belew [6] developed a method for automatically combining multiple ranking functions with the goal of ranking a set of documents in reference to a user query. Each ranking function, for example a keyword search, orders the documents it returns according to their predicted relevance to the query. The combined classifier output by this system is a weighted combination of the individual ranking functions; a set of documents is ordered according to the score assigned by the combined classifier. The task of their system is to estimate the optimal weight of each function so that the combined classifier will produce a good ranking of the documents. The system estimates the weights on a training set of queries (and documents) via a gradient descent method to maximize a utility function that is proportional to the average (over all training queries) sum over all pairs of documents (a, b), where a is preferred to b, of the difference between the combined score of a and the combined score of b. They justify the choice of this utility function, explaining that is a variant of Guttman's Point Alienation [37], which is a statistical measure of rank correlation.

Their approach is similar to ours in considering pairwise preferences of one document over another. However, unlike our approach, in their utility function they use the actual numeric scores assigned by the ranking functions. Such an approach may work in a specific domain where all the ranking functions scores have the same range and interpretation, but in a more general context it may suffer from the difficulties described in Chapter 1.

2.5 Other Related Work

Kantor [42] also considered the task of combining multiple ranking functions in the context of information retrieval. Like us, he ignored the numeric scores assigned by each ranking function, viewing each function as an ordering of the documents (he restricted each function to be a total order, whereas we allow partial orders). He combined the ranking functions using maximum, minimum, median, and summation operators. For example, the minimum operator assigns to a document the minimum of the ranks assigned to it by the ranking functions. Such a combination rule is easy to implement, but its justification is not clear. Indeed, Kantor's experiments showed that maximum, minimum, and median performed poorly compared to the single best ranking function, but summation performed the same as the best ranking function. Our work provides a combination rule with a theoretical basis and analysis.

One of the tasks encompassed by our ranking model is the meta-search task of combining the results of multiple Web search engines, described in Chapter 1. Etzioni et al. [24] also proposed a formal model of this task. They assumed that each search engine has a cost associated with querying it; their goal is then to answer the query while minimizing the cost of the meta-search. They further assume that each query has a sole relevant document, rendering ordering issues unimportant. Thus their approach is only superficially related to ours.

Chapter 3

Boosting

As described in Chapter 1, our solution to the ranking problem is based on machine learning. We begin this chapter with an introduction to the basic ideas of machine learning and pattern recognition (Section 3.1). We then present the history and development of a particular machine learning method called boosting. We first examine the theoretical origins of boosting that led to the discovery of the AdaBoost algorithm, the direct predecessor of our boosting algorithm (Section 3.2). We then survey some of the experiments that demonstrated the ability of AdaBoost to produce highly accurate prediction rules (Section 3.3). These experiments raised theoretical questions about the algorithm, such as why boosting produces good prediction rules, and why AdaBoost tends to resist overfitting (described in Section 3.1). We discuss these questions and conclude with a summary of recent work that is attempting to provide answers (Section 3.4).

Although our boosting algorithm for the ranking problem makes heavy use of the ideas developed in Sections 3.2-3.4, our presentation of it in the next chapter is self-contained and can be read after Section 3.1.

3.1 Machine Learning for Pattern Recognition

One of the primary motivations for the invention of the computer was the need to store and manipulate large amounts of information. Now that people have this tool for information processing, they are accumulating more and more data. Although it is relatively easy to gather large quantities data, analyzing and interpreting the data remain consistent challenges. One can ask simple questions about the data that may be difficult to answer. For example, a credit card company might want to know, given a list of all purchases made by its customers today, which purchases were made using stolen credit cards. As another example, a software company's customer service division may receive 900 email messages a day and would like to separate the messages into categories such as service requests from Macintosh users, service requests from PC users, and inquiries from prospective customers. Finally, when Alice is searching for a movie to watch this evening, she can look up a movie database on the Internet and ask, "which movies would I like?"

In order answer a question about a set of data, a person might go through the data in search of a *pattern*. Let's say that person is Bob. The credit card company hires Bob to find stolen credit cards by giving him examples of purchases made with stolen cards and then asking him to go through today's logs and report any suspicious transactions. In order to do this, Bob needs to detect a pattern to get an idea of what a suspicious transaction looks like. If Bob discovers a pattern that allows him to correctly identify newly stolen credit cards more often than not, the credit company will handsomely reward him. When Bob finishes that job, the software company hires him to organize its customer service email messages. Looking at examples of correctly classified messages, Bob searches for patterns and formulates rules to categorize incoming messages. One of his rules might be, "if the message contains the phrase 'please help' then it is probably a service request." Finally, at the end of the day Alice tells Bob the names of three movies that she loved and three movies that she hated and asks him to recommend movies. Bob consults the list of movies that he has seen and, if his tastes agree with Alice's, offers his suggestions.

The task that Bob is repeatedly performing is called *pattern recognition.* Given examples of what to look for, Bob formulates a rule to find new examples of the same kind. Although Bob may easily recommend a handful of movies to Alice, he will quickly tire of processing the software company's 900 email messages. Rather than going through each message himself, Bob can program his computer to do it for him (so that he can spend more time with Alice).

This is the approach of *machine learning* for pattern recognition. Let's consider the email message classification task. As in any classification task, the goal is take an *instance* (email message) and correctly predict its *class* (service category). Bob writes a *learning algorithm*, to detect patterns just like he did. He first trains the algorithm, by giving it instances labeled with the correct answers, so that it can formulate a prediction rule. He then tests the algorithm's prediction rule on unlabeled data. More specifically, Bob first gives the algorithm a *training set* of data that consists of instances labeled with their correct classification, also called training *examples.* The algorithm uses the training set to produce a classification rule that, given an instance, predicts the class of that instance. For example, one of the parts of the rule might be:[1]

if "please" appears in the message
  then the message is a request for service
  else the message is from a Macintosh user

Once constructed, the prediction rule is applied to a disjoint *test set* of data that consists of unlabeled instances. The rule predicts the class of each of the test instances, and then its predictions are compared to the correct answers (often obtained from a human). The error of the rule is usually measured as the percentage of misclassifications it made. If the error is small, then the learning algorithm is declared to be a good one and its rule is used to classify future data.

The prediction rule needs to be evaluated on a test set to make sure that it *generalizes* beyond the training set: just because a rule performs well on the training set, where it has access to the correct classification, does not mean that it will perform well on new data. For example, a rule that simply stores the correct classification of every training instance will

[1] This example is an actual rule that was produced by a learning algorithm run on the customer service email data of a well-known software company.

make perfect predictions on the training set but will be unable to make any predictions on a test set. Such a rule is said to *overfit* the training data. Also, a rule might not generalize well if the training set is not representative of the kinds of examples that the rule will encounter in the future. Similarly, if the test set is not representative of future examples, then it will not accurately measure the generalization of the rule.

At this point we need to construct a mathematical model of learning so that we can ask and answer questions about the process. The model we use, a *probabilistic* model of machine learning for pattern recognition, has been introduced and well-studied by various researchers [17, 69, 71, 72]. In this model we assume that there is a fixed and unknown probability distribution over the space of all instances. Similarly, there is a fixed and unknown classification function that takes an instance as input and outputs the correct class of the instance. The goal of a learning algorithm is to produce a rule that approximates the classification function.

We assume that the training set and test set each consist of instances that are chosen randomly and independently according to the unknown distribution (these sets differ in that the classification function is used to correctly label the training instances, whereas the test instances remain unlabeled). We consider a learning algorithm to be successful if it takes a training set as input and outputs a prediction rule that has low expected classification error on the test set (the expectation is taken over the random choice of the test set). We do not demand that the learning algorithm be successful for every choice of training set, since may be impossible if the training set is not representative of the instance space. Instead we ask that the learning algorithm be successful with high probability (taken over the choice of the training set and any internal random choices made by the algorithm).

This probabilistic model of machine learning for pattern recognition is the basis of the history and development our work. In Section 3.2 we will see how theoretical questions about this model gave rise to the first boosting algorithms, which eventually evolved into powerful and efficient practical tools for machine learning tasks, and in turn raised theoretical questions of their own.

3.2 Origins of Boosting

Given a training set of data, a learning algorithm will generate a rule that classifies the data. This rule may or may not be accurate, depending on the quality of the learning algorithm and the inherent difficulty of the particular classification task. Intuitively, if the rule is even slightly better than randomly guessing the class of an instance, the learning algorithm has found some structure in the data to achieve this edge. *Boosting* is a method that boosts the accuracy of the learning algorithm by capitalizing on its edge. Boosting uses the learning algorithm as a subroutine in order to produce a prediction rule that is guaranteed to be highly accurate on the training set. Boosting works by running the learning algorithm on the training set multiple times, each time focusing the learner's attention on different training examples. After the boosting process is finished, the rules that were output by the learner are combined into a single prediction rule which is provably accurate on the training set. This combined rule is usually also highly accurate on the test set, which has been verified both theoretically and experimentally.

In this section we outline the history and development of the first boosting algorithms that culminated in the popular AdaBoost algorithm.

3.2.1 The PAC Model

In 1982, Leslie Valiant introduced a computational model of learning known as the *probably approximately correct* (PAC) model of learning [69]. The PAC model differs slightly from the probabilistic model for pattern recognition described in Section 3.1 in that it explicitly considers the computational costs of learning (for a thorough presentation of the PAC model, see, for instance, Kearns and Vazirani [44]). A PAC learning problem is specified by an instance space and a *concept*, a boolean function defined over the instance space, that represents the information to be learned. In the email classification task described in Section 1, the instance space consists of all email messages and a concept is "a service request." The goal of a PAC learning algorithm is to output a boolean prediction rule called a *hypothesis* that approximates the concept.

The algorithm has access to an *oracle* which is a source of examples (instances with their correct label according to the concept). When the algorithm requests an example, the oracle chooses an instance at random according to a fixed probability distribution $\mathcal{D}$ that is unknown to the algorithm. (The notion of an examples oracle is an abstract model of a set of training examples. If the algorithm makes m calls to the oracle, this is equivalent to the algorithm receiving as input a set of m training examples.)

In addition to the examples oracle, the algorithm receives an *error* parameter ϵ, a *confidence* parameter δ, and other parameters that specify the respective "sizes" of the instance space and the concept. After running for a polynomial amount of time[2], the learning algorithm must output a hypothesis that, with probability $1 - \delta$, has expected error less than ϵ. That is, the algorithm must output a hypothesis that is probably approximately correct. (The probability $1 - \delta$ is taken over all possible sets of examples returned by the oracle, as well as any random decisions made by the learning algorithm, and the expectation is taken with respect to the unknown distribution $\mathcal{D}$.)

The PAC model has many strengths and received intense study after Valiant introduced it. The model proved to be quite robust: researchers proposed numerous extensions that were shown to be equivalent to the original definition. Kearns and Valiant [43] proposed one such extension by defining *strong* and *weak* learning algorithms. A strong learning algorithm runs in polynomial time and outputs a hypothesis that is probably approximately correct as just described. A weak learning algorithm runs in polynomial time and outputs a hypothesis that is probably *barely* correct, meaning that its accuracy is slightly better than the strategy that randomly guesses the label of an instance by predicting 1 with probability $\frac{1}{2}$ and 0 with probability $\frac{1}{2}$. More precisely, a weak learner receives the same inputs as a strong learner, except for the error parameter ϵ, and it outputs a hypothesis that, with probability $1 - \delta$, has expected error less than $\frac{1}{2} - \gamma$ for a fixed $\gamma > 0$. The constant γ measures the edge of the weak learning algorithm over random guessing; it is not an input to the algorithm.

Kearns and Valiant raised the question of whether or not a weak learning algorithm

[2]The algorithm is required to run in time that is polynomial in $1/\epsilon$, $1/\delta$, and the two size parameters.

could be converted into a strong learning algorithm. They referred to this problem as the *hypothesis boosting problem* since, in order to show that a weak learner is equivalent to a strong learner, one must boost the accuracy of the hypothesis output by the weak learner. When considering this problem, they provided some evidence that these notions might not be equivalent: assuming a uniform distribution over the instance space, they gave a weak learning algorithm for concepts that are monotone boolean functions, but they showed that there exists no strong learning algorithm for these functions. This showed that when restrictions are placed on the unknown distribution, the two notions of learning are not equivalent, and it seemed that this inequivalence would apply to the general case as well. Thus it came as a great surprise when Robert E. Schapire demonstrated that strong and weak learning actually are equivalent by providing an algorithm for converting a weak learner into strong learner. His was the first boosting algorithm.

3.2.2 Schapire's Algorithm

Schapire [60] constructed a brilliant method for converting a weak learning algorithm into a strong learning algorithm. Although the main idea of the algorithm is easy to grasp, the proofs that the algorithm is correct and that it runs in polynomial time are somewhat involved. The following presentation of the algorithm is from Schapire's Ph.D. thesis [61] which the reader should consult for the details.

The core of the algorithm is a method for boosting the accuracy of a weak learner by a small but significant amount. This method is applied recursively to achieve the desired accuracy.

Consider a weak learning algorithm A that with high probability outputs a hypothesis with an error rate of α with respect to a target concept c. The key idea of the boosting algorithm B is to simulate A on three different distributions over the instance space X in order to produce a new hypothesis with error significantly less than α. This simulation of A on different distributions fully exploits the property that A outputs a weak hypothesis with error slightly better than random guessing with respect to *any* distribution over X.

Let Q be the given examples oracle, and let $\mathcal{D}$ be the unknown distribution over X. Algorithm B begins by simulating A on the original distribution $D_1 = \mathcal{D}$ using oracle $Q_1 = Q$. Let h_1 be the hypothesis output by A.

Intuitively, A has found some weak advantage on the original distribution; this advantage is expressed by h_1. To force A to learn more about the "harder" parts of the distribution, B must somehow destroy this advantage. To do so, B creates a new distribution D_2 over X. An instance chosen according to D_2 has an equal chance of being correctly or incorrectly classified by h_1 (so h_1 is no better than random guessing when it receives examples drawn from D_2). The distribution D_2 is simulated by *filtering* the examples chosen according to $\mathcal{D}$ by Q. To simulate D_2, a new examples oracle Q_2 is constructed. When asked for an instance, Q_2 first flips a fair coin: if the result is heads then Q_2 requests examples from Q until one is chosen for which $h_1(x) = c(x)$; otherwise, Q_2 waits for an instance to be chosen for which $h_1(x) \neq c(x)$. (Schapire shows how to prevent Q_2 from having to wait too long in either of these loops for a desired instance, which is necessary for algorithm B to run in polynomial time). Algorithm B simulates A again, this time providing A with examples chosen by Q_2

according to D_2. Let h_2 be the resulting output hypothesis.

Finally, D_3 is constructed by filtering out from $\mathcal{D}$ those instances on which h_1 and h_2 agree. That is, a third oracle Q_3 simulates the choice of an instance according to D_3 by requesting instances from Q until one is found for which $h_1(x) \neq h_2(x)$. (Again Schapire shows how to limit the time spent waiting in this loop for a desired instance.) Algorithm A is simulated a third time, now with examples drawn from Q_3, producing hypothesis h_3.

At last, B outputs its hypothesis h, defined as follows. Given an instance x, if $h_1(x) = h_2(x)$ then h predicts the agreed upon value; otherwise h predicts $h_3(x)$ (h_3 serves as the tie breaker). In other words, h takes the majority vote of h_1,h_2, and h_3. Schapire is able to prove that the error of h is bounded by $g(\alpha) = 3\alpha^2 - 2\alpha^3$, which is significantly smaller than the original error α.

Algorithm B serves as the core of the boosting algorithm and is called recursively to improve the accuracy of the output hypothesis. The boosting algorithm takes as input a desired error bound ϵ and a confidence parameter δ, and the algorithm constructs a hypothesis with error less than ϵ from weaker, recursively computed hypotheses.

In summary, Schapire's algorithm boosts the accuracy of a weak learner by efficiently simulating the weak learner on multiple distributions over the instance space and taking the majority vote of the resulting output hypotheses. Schapire's paper was rightly hailed as ingenious, both in the algorithm it presented and the elegant handling of the proof technicalities. The equivalence of strong and weak learnability settled a number of open questions in computational learning theory, and Schapire used the boosting algorithm to derive tighter bounds on various resources used in the PAC model. His algorithm also had implications in the areas of computational complexity theory and data compression.

3.2.3 The Boost-By-Majority Algorithm

Schapire's boosting algorithm was certainly a theoretical breakthrough, but the algorithm and its analysis are quite complicated. And although the algorithm runs in polynomial time, it is inefficient and impractical because of its repeated recursive calls. In addition, the output final hypothesis is complex due to its recursive construction.

A much simpler and more efficient algorithm was constructed by Yoav Freund one year after Schapire's original paper. Freund's algorithm, called the Boost-By-Majority algorithm [25, 26], also works by constructing many different distributions over the instance space. These constructed distributions are presented to the weak learner in order to focus the learner's attention on "difficult" regions of the unknown distribution. The weak learner outputs a weak hypothesis for each distribution it receives; intuitively, these hypotheses perform well on different portions of the instance space. The boosting algorithm combines these hypotheses into a final hypothesis using a single majority vote; this final hypothesis has provably low expected error on the instance space.

Freund elegantly presents the main idea of his boosting algorithm by abstracting the hypothesis boosting problem as a game, which he calls the majority-vote game. The majority-vote game is played by two players, the *weightor* and the *chooser.* The weightor corresponds to the boosting algorithm and the chooser corresponds to the weak learner. The game is played over a finite space S.[3] A parameter $0 < \gamma < \frac{1}{2}$ is fixed before the game. The game proceeds for T rounds (T is chosen by the weightor), where each round consists of the following steps:

1. The weightor picks a *weight* measure D on S. The weight measure is a probability distribution over S, and the weight of a subset A is $D(A) = \sum_{x \in A} D(x)$.
2. The chooser selects a set $U \subseteq S$ such that $D(U) \geq \frac{1}{2} + \gamma$ and marks all of the points in U.

The game continues until the weightor decides to stop, at which point it suffers a *loss*, calculated as follows. Let $L \subseteq S$ be the set of points that were marked less than or equal to $T/2$ times. The weightor's loss is $|L|/|S|$, the relative size of L. The goal of the weightor is minimize its loss and the goal of the chooser to maximize it. (In the language of game theory, this is a complete information, zero-sum game.)

We now illustrate the correspondence between the majority-vote game and the hypothesis boosting problem. The weightor is the boosting algorithm and the chooser is the weak learner. The space S is the training set, and the fixed parameter γ is the edge of the weak learner. During each round t, the weightor's weight measure D on round t is a probability distribution over the training set. Given the training set weighted by distribution D, the weak learner produces a weak hypothesis. The points marked by the chooser are the training examples that the weak hypothesis classifies correctly. After T rounds of the game, T weak hypotheses have been generated by the weak learner. These are combined into a final hypothesis H using a majority vote. H is then used to classify the training instances. The points that are marked more than $T/2$ times are instances that are correctly classified by more than $T/2$ weak hypotheses; thus, these instances are also correctly classified by H. The points in L (those that are marked less than or equal to $T/2$ times), are misclassified by H (we are making the pessimistic assumption that, if ties are broken randomly, the outcomes are always decided incorrectly). Thus the error of H on the training set is $|L|/|S|$. The boosting algorithm's goal is to minimize this error.

Freund showed that there exists a weighting strategy for the weightor, meaning an algorithm for choosing D on each round of the game, that guarantees that its loss will be small after a few rounds, *regardless* of the behavior of the chooser. More precisely, he gave a strategy such that for any S, $\epsilon > 0$, and $\delta > 0$, the weightor can guarantee that its loss is less than ϵ after $T \leq \frac{1}{2}(1/\gamma)^2 \ln(1/(2\epsilon))$ rounds, no matter what the chooser does.

Although the weighting strategy is not too complicated, we choose not to present it here since it is superceded by the method of the AdaBoost algorithm, presented in the next section. Freund gives an explicit algorithm for his strategy, which iteratively updates the weight of the point x on round t as a function of t, T, γ, and how many times x has been marked already. He also proves a tight bound $F(\gamma, \epsilon)$ on T, the number of rounds in the majority-vote game required to bring the training error below ϵ. He proves that this bound is optimal by giving a second weighting strategy that uses $F(\gamma, \epsilon)$ rounds. Freund used his

[3] Freund proves his results for the game defined over an arbitrary probability space. The case we consider where the space is finite and the distribution is uniform is all that is needed to derive the Boost-By-Majority algorithm.

algorithm and the methods used to construct it to prove tighter bounds on a number of different problems from the PAC learning model, complexity theory, and data compression.

Generalization Error

We now return to the point mentioned earlier, that producing a classifier with low error on a training sample S implies that the classifier will have low expected error on instances outside S. This result comes from the notion of VC-dimension and uniform convergence theory [71, 72]. Roughly, the VC-dimension of a space of classifiers captures their complexity; the higher the VC-dimension, the more complex the classifier. Vapnik [71] proved a precise bound on the difference between the training error and generalization error of a classifier. Specifically, let h be a classifier that comes from a space of binary functions with VC-dimension d. Its generalization error is $\Pr_{\mathcal{D}}[h(x) \neq y]$ where the probability is taken with respect to the unknown distribution $\mathcal{D}$ over the instance space. Its empirical error is $\Pr_S[h(x) \neq y]$, the empirical probability on a set S of m training examples chosen independently at random according to $\mathcal{D}$. Vapnik proved that, with high probability (over the choice of training set),

$$\Pr_{\mathcal{D}}[h(x) \neq y] \leq \Pr_S[h(x) \neq y] + \tilde{O}\left(\sqrt{\frac{d}{m}}\right), \tag{3.1}$$

($\tilde{O}(\cdot)$ is the same as $O(\cdot)$ ignoring log factors). Thus, if an algorithm outputs classifiers from a space of sufficiently small VC-dimension that have zero error on the training set, then it can produce a classifier with arbitrarily small generalization error by training on a sufficiently large number of training examples.

Although useful for proving theoretical results, the above bound is not predictively accurate in practice. Also, typical learning scenarios involve a fixed set of training data on which to build the classifier. In this situation Vapnik's theorem agrees with the intuition that if the output classifier is sufficiently simple and is accurate on the training data, then its generalization error will be small.

It can be proved that the VC-dimension of the majority vote classifier generated by the Boost-By-Majority algorithm is $\tilde{O}(Td)$, where T is the number of rounds of boosting and d is the VC-dimension of the space of hypotheses generated by the weak learner [31]. Thus, given a large enough training sample, Boost-By-Majority is able to produce an arbitrarily accurate combined hypothesis.[4]

Summary

In summary, Freund's Boost-By-Majority algorithm uses the weak learner to create a final hypothesis that is highly accurate on the training set. Similar in spirit to Schapire's algorithm, Boost-By-Majority achieves this by presenting the weak learner with different distributions over the training set, which forces the weak learner to output hypotheses that are accurate on different parts of the training set. However, Boost-By-Majority is major

[4]If the desired generalization error is $\epsilon > 0$, the number of training examples required is d/ϵ^2, a polynomial in $1/\epsilon$ and d, which is required by the PAC model (Section 3.2.1).

improvement over Schapire's algorithm because it is much more efficient and its final hypothesis is merely a majority vote over the weak hypotheses, which is much simpler than the recursive final hypothesis produced by Schapire's algorithm.

3.2.4 The AdaBoost Algorithm

The need for a better boosting algorithm

So far we've seen two boosting algorithms for increasing the accuracy of a base learning algorithm. The goal of the boosting algorithms is to output a combined hypothesis, which is a majority vote of barely accurate weak hypotheses generated by the base learning algorithm, that is accurate on the training data. Using Vapnik's theorem (Eq. (3.1)), this implies that the combined hypothesis is highly likely to be accurate on the entire instance space.

Schapire's recursive algorithm constructs different distributions over the training data in order to focus the base learner on "harder" parts of the unknown distribution. Freund's Boost-By-Majority algorithm constructs different distributions by maintaining a weight for each training example and updating the weights on each round of boosting. This algorithm reduces training error much more rapidly, and its output hypothesis is simpler, being a single majority vote over the weak hypotheses.

Although Boost-By-Majority is very efficient (it is optimal in the sense described in the previous section), it has two practical deficiencies. First, the weight update rule depends on γ, the worst case edge of the base learner's weak hypotheses over random guessing (recall that the base learner outputs hypotheses whose expected error with respect to any distribution over the data is less than $\frac{1}{2} - \gamma$). In practice γ is usually unknown, and estimating it requires either knowledge of the underlying distribution of the data (also usually unknown) or repeated experiment. Secondly, Freund proved that Boost-By-Majority requires approximately $1/\gamma^2$ rounds in order to reduce the training error to zero. Thus if $\gamma = 0.001$, one million rounds of boosting may be needed. During the boosting process a weak hypothesis may be generated whose error is much less than $\frac{1}{2} - \gamma$, but Boost-By-Majority is unable to use this advantage to speed up the boosting process.

For these reasons, Freund and Schapire joined forces to develop a more practical boosting algorithm. The algorithm they discovered, AdaBoost, came from a unexpected connection to *on-line learning*.

The on-line learning model

In the on-line learning model, introduced by Littlestone [47], learning takes place in a sequence of trials. During each trial, an on-line learning algorithm is given an unlabelled instance (such as an email message) and asked to predict the label of the instance (such as "service request"). After making its prediction, the algorithm receives the correct answer and suffers some loss depending on whether or not its prediction was correct. The goal of the algorithm is to minimize its cumulative loss over a number of such trials.

One kind of on-line learning algorithm, called a *voting algorithm*, makes its predictions by employing an input set of prediction rules called *experts*. The algorithm maintains a real-valued *weight* for each expert that represents its confidence in the expert's advice. When given an instance, the voting algorithm shows the instance to each expert and asks for its vote on its label. The voting algorithm chooses as its prediction the weighted majority vote of the experts. When the correct label of the instance is revealed, both the voting algorithm and each expert may suffer some loss. Indeed, we can view this process as the voting algorithm first receiving and instance and then receiving a vector of losses for each expert. After examining the loss of each expert on the instance, the voting algorithm may increase or decrease the weight of an expert according to whether or not the expert predicted the correct label.

The Hedge learning algorithm

Freund and Schapire were working on a particular voting algorithm called Hedge [31] which led to the discovery of the new boosting algorithm. The Hedge algorithm[5] receives as input a set of N experts and a learning rate parameter $\beta \in [0, 1]$. It initializes the weight vector $\mathbf{p}^1 = \langle p_1^1, \ldots, p_N^1 \rangle$ to be a uniform probability distribution over the experts. (The initial weight vector can be initialized according to a prior distribution if such information is available). During learning trial t, the algorithm receives an instance and the corresponding loss vector $\ell^t = \langle \ell_1^t, \ldots, \ell_N^t \rangle$, where $\ell_i^t \in [0, 1]$ is the loss of expert i on the instance. The loss Hedge suffers is $\mathbf{p}^t \cdot \ell^t$, the expected loss of its prediction according to its current distribution over the experts. Hedge updates the distribution according to the rule

$$p_i^{t+1} = p_i^t \, \beta^{\ell_i^t}$$

which has the effect of decreasing the weight an expert if its prediction was incorrect ($\mathbf{p}^{t+1}$ is renormalized to make it a probability distribution.) Freund and Schapire proved that the cumulative loss of the Hedge algorithm over T trials is almost as good as that of the best expert, meaning the expert with loss $\min_i L_i$ where $L_i = \sum_{t=1}^{T} \ell_i^t$. Specifically, they proved that the cumulative loss of Hedge is bounded by $c \min_i L_i + a \ln N$, where the constants c and a turn out to be the best achievable by any on-line learning algorithm [73].

Application to boosting: AdaBoost

Using the Hedge algorithm and the bounds on its performance, Freund and Schapire derived a new boosting algorithm. The natural application of Hedge to the boosting problem is to consider a fixed set of weak hypotheses as experts and the training examples as trials. If it makes an incorrect prediction, the weight of a hypothesis is decreased, via multiplication by a factor $\beta \in [0, 1]$. The problem with this boosting algorithm is that, in order to output a highly accurate prediction rule in a reasonable amount of time, the weight update factor β must depend on the worst-case edge γ. This is exactly the dependence they were trying to avoid. Freund and Schapire in fact used the *dual* application: the experts correspond to training examples and trials correspond to weak hypotheses. The weight update rule is similarly reversed: the weight of an example is *increased* if the current weak hypothesis

[5]The Hedge algorithm and its analysis are direct generalizations of the "weighted majority" algorithm of Littlestone and Warmuth [48].

Algorithm **AdaBoost**
Given: training examples $(x_1, y_1), \ldots, (x_m, y_m)$ where $x_i \in X$, $y_i \in Y = \{+1, -1\}$
Initialize: $D_1(i) = 1/m$.
For $t = 1, \ldots, T$:

- Train weak learner using distribution D_t.
- Get weak hypothesis $h_t : X \to [-1, +1]$ and its error $\epsilon_t = \sum_{i=1}^m D_t(i) y_i h_t(x_i)$.
- Set
$$\alpha_t = \tfrac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right).$$
- Update:
$$D_{t+1}(i) = \frac{D_t(i) \exp\left(-\alpha_t y_i h_t(x_i)\right)}{Z_t}$$
where Z_t is a normalization factor (chosen so that D_{t+1} will be a distribution).

Output the final hypothesis:

$$H(x) = \operatorname{sign}(f(x)) = \operatorname{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x) \right).$$

Figure 3-1: The AdaBoost algorithm.

predicts its label incorrectly. Also, the parameter β is no longer fixed; it is β_t set as a function of the error of the weak hypothesis on that round. This is the AdaBoost algorithm.

We present pseudocode for the AdaBoost algorithm in Figure 3-1. We use the more convenient notation of the recent generalization of AdaBoost by Schapire and Singer [65]. The first notational change is that positive examples are labelled $+1$ and negative examples are labelled -1 (instead of 1 and 0, respectively). Accordingly, the weak hypotheses h_t now map the instance space X to $[-1, +1]$. Note that this means that $h_t(x_i) \neq y_i$ if and only if $y_i h_t(x_i) < 0$. The second change is that the weight update rule uses the constant α_t instead of β_t. The relationship between the two is $\beta_t = e^{-2\alpha_t}$ or $\alpha_t = -\frac{1}{2} \ln \beta_t$. Despite these differences in notation, the algorithm in Figure 3-1 is identical to Freund and Schapire's original algorithm.

AdaBoost behaves similarly to the other boosting algorithms we've seen so far in that weak hypotheses are generated successively and the weight of each training example is increased if that example is "hard". The main difference between AdaBoost and Boost-By-Majority is the weight update rule: AdaBoost uses a multiplicative update rule that depends on the loss of the current weak hypothesis, not its worst case edge γ. Another difference is that each weak hypothesis receives a weight α_t when it is generated; AdaBoost's combined hypothesis is a weighted majority vote of the weak hypotheses rather than a simple majority vote.

Training error

The effect of the weight update rule is to reduce the training error. It is relatively easy to show that the training error drops exponentially rapidly:

$$\frac{1}{m}\left|\{i : H(x_i) \neq y_i\}\right| \leq \frac{1}{m}\sum_{i=1}^{m} \exp(-y_i f(x_i)) = \prod_{t=1}^{T} Z_t. \tag{3.2}$$

The inequality follows from the fact that $\exp(-y_i f(x_i)) \geq 1$ if $y_i \neq H(x_i)$, and the equality can be seen by unraveling the recursive definition of D_t [65].

In order to rapidly minimize training error, Eq. (3.2) suggests that α_t and h_t should be chosen on round t to minimize the normalization factor

$$Z_t = \sum_{i=1}^{m} D_t(i) \exp(-\alpha_t y_i h_t(x_i)). \tag{3.3}$$

Of course, our learning model assumes that the weak learner is a subroutine to the boosting algorithm and is not required to choose its weak hypotheses to minimize Eq. (3.3). In practice, however, one often designs and implements the weak learner along with the boosting algorithm, depending on the application, and thus has control over which hypothesis is output as h_t. If the weak hypotheses h_t are binary, then using the setting for α_t in Fig. (3-1), the bound on the training error simplifies to

$$\prod_{t=1}^{T} 2\sqrt{\epsilon_t(1-\epsilon_t)} = \prod_{t=1}^{T} \sqrt{1-4\gamma_t^2} \leq \exp\left(-2\sum_{t=1}^{T} \gamma_t^2\right)$$

where γ_t is the empirical edge of h_t over random guessing, that is, $\epsilon_t = \frac{1}{2} - \gamma_t$. Note that this means that AdaBoost is able to improve in efficiency if *any* of the weak hypotheses have an error rate lower than the worst case error $\frac{1}{2} - \gamma$. This is a desirable property not enjoyed by the Boost-By-Majority algorithm; in practice, AdaBoost reduces the training error to zero very rapidly, as we will see in Section 3.3.

In addition, Eq. (3.2) indicates that AdaBoost is essentially a greedy method for finding a linear combination f of weak hypotheses which attempts to minimize

$$\sum_{i=1}^{m} \exp(-y_i f(x_i)) = \sum_{i=1}^{m} \exp\left(-y_i \sum_{t=1}^{T} \alpha_t h_t(x_i)\right). \tag{3.4}$$

On each round t, AdaBoost receives h_t from the weak learner and then sets α_t to add one more term to the accumulating weighted sum of weak hypotheses in such a way that Eq. (3.4) will be maximally reduced. In other words, AdaBoost is performing a kind of steepest descent search to minimize Eq. (3.4), where each step is constrained to be along the coordinate axes (we identify coordinate axes with the weights assigned to the weak hypotheses).

Generalization error

Freund and Schapire proved that as the number of boosting rounds T increases, the training error of the combined classifier AdaBoost drops to zero exponentially fast. Using techniques of Baum and Haussler [8] and Vapnik's theorem (Eq. (3.1)), they showed that, if the weak learner has a hypothesis space of VC-dimension d, then with high probability the generalization error of the combined classifier H is bounded:

$$\Pr_{\mathcal{D}}\left[H(x) \neq y\right] \leq \Pr_{S}\left[H(x) \neq y\right] + \tilde{O}\left(\sqrt{\frac{Td}{m}}\right), \tag{3.5}$$

where $\Pr_S[\cdot]$ denotes the empirical probability on the training sample S. This implies that the generalization error of H can be made arbitrarily small by training on a large enough number of examples. It also suggests that H will overfit a fixed training sample as the number of rounds of boosting T increases.

Non-binary classification

Freund and Schapire also generalized the AdaBoost algorithm to handle classification problems with more than two classes. Specifically, they presented two algorithms for *multiclass* problems, where the label space Y is a finite set. They also presented an algorithm for *regression* problems where $Y = [0, 1]$. Schapire [62] used error-correcting codes to produce another boosting algorithm for multiclass problems (see also Dietterich and Bakiri [19] and Guruswami and Sahai [36]). In their generalization of binary AdaBoost, Schapire and Singer [65] proposed another multiclass boosting algorithm as well as an algorithm for *multi-label* problems where an instance may have more than one correct label.

Summary

The AdaBoost algorithm was a breakthrough. Once boosting became practical, the experiments could begin. In the next section we will discuss the empirical evaluation of AdaBoost.

3.3 Experiments with Boosting Algorithms

When the first boosting algorithms were invented they received a small amount of attention from the experimental machine learning community [21, 22]. Then the AdaBoost algorithm arrived with its many desirable properties: a theoretical derivation and analysis, fast running time, and simple implementation. These properties attracted machine learning researchers who began experimenting with the algorithm. All of the experimental studies showed that AdaBoost almost always improves the performance of various base learning algorithms, often by a dramatic amount.

We begin this section by discussing the application of boosting to one kind of base learning algorithm that outputs decision tree classifiers. We then briefly survey other experimental studies. We conclude with a discussion of the questions raised by these experiments with AdaBoost that led to further theoretical study of the algorithm.

3.3.1 Decision Trees

Experiments with the AdaBoost algorithm usually apply it to classification problems. Recall that a classification problem is specified by a space X of instances and a space Y of labels, where each instance x is assigned a label y according to an unknown labelling function $c : X \rightarrow Y$. We assume that the label space Y is finite. The input to a base learning algorithm is a set of training examples $\langle(x_1, y_1), \ldots, (x_m, y_m)\rangle$, where it is assumed that y_i is the correct label of instance x_i (i.e., $y_i = c(x_i)$). The goal of the algorithm is to output a classifier $h : X \rightarrow Y$ that closely approximates the unknown function c.

The first experiments with AdaBoost [20, 29, 53] used it to improve the performance of algorithms that generate decision trees, which are defined as follows. Suppose each instance $x \in X$ is represented as a vector of n *attributes* $\langle a_i, \ldots, a_n\rangle$ that take on either discrete or continuous values. For example, an attribute vector that represents human physical characteristics is $\langle$*height, weight, hair color, eye color, skin color*$\rangle$. The values of these attributes for a particular person might be $\langle$1.85 m, 70.5 kg, *black, dark brown, tan*$\rangle$. A *decision tree* is a hierarchical classifier that classifies instances according the values of their attributes. Each non-leaf node of the decision tree has an associated attribute a (one of the a_i's) and a value v (one of the possible values of a). Each non-leaf node has three children designated as "yes", "no", and "missing." Each leaf node u has an associated label $y \in Y$.

A one node decision tree, called a *stump* [40], consists of one internal node and three leaves. Consider a stump T_1 whose internal node compares the value of attribute a to value v. T_1 classifies instance x as follows. Let $x.a$ be the value of attribute a of x. If a is a discrete-valued attribute then

- if $x.a = v$ then T_1 assigns x the label associated with the "yes" leaf.
- if $x.a \neq v$ then T_1 assigns x the label associated with the "no" leaf.
- if $x.a$ is undefined, meaning x is missing a value for attribute a, then T_1 assigns x the label associated with the "missing" leaf.

If instead a is a continuous-valued attribute, T_1 applies a threshold test $(x.a > v)$ instead of an equality test.

A general decision tree T has many internal nodes with associated attributes. In order to classify instance x, T traces x along the path from the root to a leaf u according to the outcomes at every decision node; T assigns x the label associated with leaf u. A decision tree can be thought of as a partition of the instance space X into pairwise disjoint sets X_u whose union is X, where each X_u has an associated logic expression that expresses the attribute values of instances that fall in that set (for example "*hair color* = *brown* and *height* < 1 m").

The goal of a decision tree learning algorithm is to find a partition of X and an assignment of labels to each set of the partition that minimizes the number of mislabelled instances. Algorithms such as CART [13] and C4.5 and its successors [54] use a greedy strategy to generate a partition and label assignment which has low error on the training set. These algorithms run the risk of overfitting, meaning creating a specialized decision tree that is highly accurate on the training set but performs poorly on the test set. To resist this when growing the tree, the algorithms *prune* the tree of nodes thought to be too specialized.

3.3.2 Boosting Decision Trees

We describe two experiments using AdaBoost to improve the performance of decision tree classifiers. The first experiment [29] used as a base learner a simple algorithm for generating a decision stump; the final hypothesis output by AdaBoost was then a weighted combination of stumps. In this experiment AdaBoost was compared to *bagging* [10], another method for generating and combining multiple classifiers, in order to separate the effects of combining classifiers from the particular merits of the boosting approach. AdaBoost was also compared to C4.5, a standard decision tree learning algorithm. The second experiment [29, 20, 53] used C4.5 itself as the base learner; here also boosting was compared to C4.5 alone and to bagging. Before we report the results of the experiments, we briefly describe bagging, following Quinlan's presentation [53].

Bagging

Invented by Breiman [10], *bagging* ("bootstrap aggregating") is a method for generating and combining multiple classifiers by repeatedly sampling the training data. Given a base learner and a training set of m examples, bagging runs for T rounds and then outputs a combined classifier. For each round $t = 1, 2, \ldots, T$, a training set of size m is sampled (with replacement) from the original examples. This training set is the same size as the original data, but some examples may not appear in it while others may appear more than once. The base learning algorithm generates a classifier C^t from the sample and the final classifier C^* is formed by combining the T classifiers from these rounds. To classify an instance x, a vote for class k is recorded for every classifier for which $C^t(x) = k$, and $C^*(x)$ is then the class with the most votes (with ties broken arbitrarily).

Breiman used bagging to improve the performance of the CART decision tree algorithm on seven moderate-sized datasets. With the number of classifiers T set to 50, he reported that the average error of the bagged classifier C^* ranged from 0.57 to 0.94 of the corresponding error when a single classifier was learned. He noted, "The vital element is the instability of the [base learning algorithm]. If perturbing the [training] set can cause significant changes in the [classifier] constructed, then bagging can improve accuracy."

Bagging and boosting are similar in some respects. Both use a base learner to generate multiple classifiers by training the base learner on different samples of the training data. As a result, both methods require that the base learner be "instable" in that small changes in the training set will lead to different classifiers. However, there are two major differences between bagging and boosting. First, bagging resamples the training set on each round according to a uniform distribution over the examples. In contrast, boosting resamples on each round according to a different distribution that is modified based on the performance of the classifier generated on the previous round. Second, bagging uses a simple majority vote over the T classifiers whereas boosting uses a weighted majority vote (the weight of a classifier depends on its error relative to the distribution from which it was generated).

Boosting Decision Stumps

As a base learner, Freund and Schapire [29] used a simple greedy algorithm for finding the decision stump with the lowest error (relative to a given distribution over the training examples). They ran their experiments on 27 benchmark datasets from the repository at the University of California at Irvine [52]. They set the number of boosting and bagging rounds to be $T = 100$.

Boosting did significantly and uniformly better than bagging. The boosting (test) error rate was worse than the bagging error rate on only one dataset, and the improvement of bagging over boosting was only 10%. In the most dramatic improvement (on the soybean-small dataset), the best stump had an error rate of 57.6%, bagging reduced the error to 20.5% and boosting achieved an error of 0.25%. On average, boosting improved the error rate over using a single (best) decision stump by 55.2%, compared to bagging which gave an improvement of 11.0%.

A comparison to C4.5 revealed that the method of boosting decision stumps does quite well as a learning algorithm in its own right. The algorithm beat C4.5 on 10 of the benchmarks (by at least 2%), tied on 14, and lost on 3. C4.5's improvement in performance over a single decision stump was 49.3% (compared to boosting's 55.2%).

Boosting C4.5

An algorithm that produces a decision stump classifier can be thought of as a weak learner. The last experiment showed that boosting was able to dramatically improve its performance, more often than bagging and to a greater degree. Freund and Schapire [29] and Quinlan [53] investigated the abilities of boosting and bagging to improve C4.5, a considerably stronger learning algorithm.

When using C4.5 as the base learner, boosting and bagging seem more evenly matched, although boosting still seems to have a slight advantage. Freund and Schapire's experiments revealed that on average, boosting improved the error rate of C4.5 by 24.8%, bagging by 20.0%. Bagging was superior to C4.5 on 23 datasets and tied otherwise, whereas boosting was superior on 25 datasets and actually degraded performance on 1 dataset (by 54%). Boosting beat bagging by more than 2% on 6 of the benchmarks, while bagging did not beat boosting by this amount (or more) on any benchmark. For the remaining 21 benchmarks, the difference in performance was less than 2%.

Quinlan's results [53] with bagging and boosting C4.5 were more compelling. He ran boosting and bagging for $T = 10$ rounds and used 27 datasets from the UCI repository, about half of which were also used by Freund and Schapire. He found that bagging reduced C4.5's classification error by 10% on average and was superior to C4.5 on 24 of the 27 datasets and degraded performance on 3 (the worst increase was 11%). Boosting reduced error by 15% but improved performance on 21 datasets and degraded performance on 6 (the worst increase was 36%). Compared to one another, boosting was superior to bagging (by more than 2%) on 20 of the 27 datasets. Quinlan concluded that boosting outperforms bagging, often by a significant amount, but bagging is less prone to degrade the base learner.

Drucker and Cortes [20] also found that AdaBoost was able to improve the performance of C4.5. They used AdaBoost to build ensembles of decision trees for optical character recognition (OCR) tasks. In each of their experiments, the boosted decision trees performed better than a single tree, sometimes reducing the error by a factor of four.

Boosting Past Zero

Quinlan experimented further to try to determine the cause for boosting's occasional degradation in performance. In the original AdaBoost paper [31], Freund and Schapire attributed this kind of degradation to overfitting. As discussed earlier, the goal of boosting is to construct a combined classifier consisting of weak classifiers. In order to produce the best classifier, one would naturally expect to run AdaBoost until the training error of the combined classifier reaches zero. Further rounds in this situation would seem only to overfit–they will increase the complexity of the combined classifier but cannot improve its performance on the training data.

To test the hypothesis that degradation in performance was due to overfitting, Quinlan repeated his experiments with $T = 10$ as before but stopped boosting if the training error reached zero. He found that in many cases, C4.5 required only three rounds of boosting to produce a combined classifier that performs perfectly on the training data; the average number of rounds was 4.9. Despite using fewer rounds, and thus being less prone to overfitting, the test error of boosted C4.5 was *worse:* the average error over the 27 datasets was 13% higher than when boosting was run for $T = 10$ rounds. This meant that boosting continued to improve the accuracy of the combined classifier (on the test set) even after the training error reached zero!

Drucker and Cortes [20] made a related observation of AdaBoost's resistance to overfitting in their experiments using boosting to build ensembles of decision trees, "Overtraining never seems to be a problem for these weak learners, that is, as one increases the number of trees, the ensemble test error rate asymptotes and never increases." We will return to this point in Section 3.4.

3.3.3 Other Experiments

We now describe other experiments with the AdaBoost algorithm.

Breiman [12] compared boosting and bagging using decision trees on real and synthetic data in order to determine the differences between the two methods. In the process he formulated an explanation of boosting's excellent generalization behavior, and he derived a new boosting algorithm. We discuss his work in more detail in Section 3.4.1.

Dietterich [18] built ensembles of decision trees using boosting, bagging, and randomization (the next attribute to add to the tree is chosen uniformly at random among a restricted set of attributes). His results were consistent with the trend we have seen: boosting produces better combined classifiers than bagging or randomization. However, when he introduced noise into the training data, meaning choosing a random subset of the examples and assigning each a label chosen randomly among the incorrect ones, he found that bagging performs much better than boosting and sometimes better than randomization.

Bauer and Kohavi [7] conducted an extensive experimental study of the effects of boosting, bagging, and related ensemble methods on various base learners, including various decision trees and the Naive-Bayes predictor [23]. Like Dietterich, they also found that boosting performs worse than bagging on noisy data.

Jackson and Craven [41] employed AdaBoost using sparse perceptrons as the weak learning algorithm. Testing on three datasets, they found that boosted sparse perceptrons outperformed more general multi-layered perceptrons, as well as C4.5. A main feature of their results was that the boosted classifiers were very simple and were easy for humans to interpret, whereas the classifiers produced by multi-layered perceptrons or C4.5 were much more complex and incomprehensible.

Maclin and Opitz [49] compared boosting and bagging using neural networks and decision trees. They performed their experiments on datasets from the UCI repository and found that boosting methods were better able to improve the performance of both of the base learners. They also observed that boosting is sensitive to noise.

Other experiments not surveyed here include those by Dietterich and Bakiri [19], Margineantu and Dietterich [50], Schapire [62], and Schwenk and Bengio [66].

3.3.4 Summary

We have seen that experiments with the AdaBoost algorithm revealed that it able to use a base learning algorithm to produce a highly accurate prediction rule. AdaBoost usually improves the base learner quite dramatically, with minimal extra computation costs. Along these lines, Leslie Valiant praised AdaBoost in his 1997 Knuth Prize Lecture [70], "The way to get practioners to use your work is to give them an extremely simple algorithm that, in minutes, does magic like *this*!" (referring to Quinlan's results).

These experiments raised many questions about the behavior of AdaBoost:

- Why is AdaBoost able to produce good classifiers?
- Why is AdaBoost able to continue to reduce the test error of the combined classifier after the training error reaches zero?
- Why does AdaBoost resist overfitting?
- How can AdaBoost be modified to work in the presence of noise?

In the next section we will present the attempts of various reseachers to answer these questions.

3.4 Why Does Boosting Work?

As theory suggested, the many experiments with the AdaBoost algorithm verified that it rapidly produced combined classifiers that were much more accurate on test data than any constituent classifier. In turn, the experiments raised questions that the theory was not prepared to answer, the most prominent of which is, why AdaBoost does not overfit when run for many rounds, and how it is able to continue to reduce the test error of the combined classifier after its training error reaches zero. This question has not been fully answered as of now. In this section we survey the various interpretations of boosting made by various researchers, usually in an attempt to explain the surprising phenomenon of its resistance to overfitting.

We begin with Breiman's justification of AdaBoost's performance based on an analysis of its affect on the bias and variance of the weak learner. We then survey the margins theory of Schapire, Freund, Bartlett, and Lee, which shed some light on how generalization performance could continue to improve after training error reaches zero. Next, we explore a connection between boosting and game theory elucidated by Freund and Schapire, and then briefly describe a recent statistical view of boosting by Friedman, Hastie, and Tibshirani. Finally, we conclude with a short discussion of AdaBoost's sensitivity to noise, meaning mislabelled training examples.

3.4.1 Arcing, bias, and variance

Intrigued by the performance of boosting and bagging on decision trees, Breiman [12] formulated an explanation of the differences between the two ensemble methods, as well as a justification for boosting's resistance to overfitting. Based on his observations he derived a new boosting algorithm, which he called an *arcing* algorithm (adaptively resample and combine), and compared it to AdaBoost on real and artificial datasets.

The goal of any learning algorithm for a classification task is to minimize its generalization error, the probability that its output classifier will misclassify an instance not in the training set. Breiman, and other researchers [45, 46, 68], have examined different ways of decomposing the generalization error of an algorithm into two terms called the (statistical) *bias* and *variance*. Intuitively, the bias term measures the *systematic* error of the learning algorithm, in other words, any inherent inability of the algorithm to perfectly classify the data. The variance term measures the error that is due to *fluctuations* such as noise in the data or random choices made by the algorithm.

Breiman defined one particular bias–variance decomposition and argued that in order for a learning algorithm to reduce its generalization error, it must reduce its bias and variance. With this observation in mind, he ran bagging and boosting, with CART decision trees [13] as the weak learner, on various artificial datasets, chosen so that he could accurately estimate the bias and variance of the ensemble methods. His experiments showed that both bagging and boosting (or arcing, as he called it) do not significantly affect the bias of the weak learner, but they do drastically reduce its variance (meaning the variance of the ensemble is much less than the variance of a single classifier). After observing that boosting achieved lower test error than bagging on some real-world datasets, Breiman concluded that boosting beats bagging because it is better at reducing variance (as he demonstrated on his artificial datasets). It is a natural conclusion that boosting and bagging should reduce variance since they average over many classifiers.

Breiman then developed an ad-hoc new boosting algorithm named arc-x4:

> After testing [AdaBoost] I suspected that its success lay not in its specific form but in its adaptive resampling property, where increasing weight was placed on those cases more frequently misclassified. To check on this, I tried three simple update schemes for the [distribution over examples]. In each, the update was of

the form $1 + m(n)^h$ [$m(n)$ is the number of times example n was misclassified by the previously constructed classifiers], and $h = 1, 2, 4$ was tested on the waveform data. The last one did the best and became arc-x4. Higher values of h were not tested so further improvement is possible.

Breiman found that arc-x4 performed comparably to AdaBoost in producing classifiers with small test error, and this was verified by other researchers [7, 49].

Breiman's observation that boosting's strength is its ability to reduce the variance of the weak learner was not replicated by other researchers; in fact, they found the opposite, that boosting tends to reduce *both* bias and variance [7, 33, 64]. In particular, Freund and Schapire [28] discussed Breiman's paper [12] in the same journal issue. Schapire et al. [64] also gave a detailed discussion of the merits and weaknesses of the application of the bias–variance analysis to boosting. They offered an alternative explanation for the effectiveness of boosting, which we present in the next section.

3.4.2 Margins analysis

In order to produce the best classifier, one would naturally expect to run AdaBoost until the training error of the combined classifier reaches zero. Further rounds in this situation would seem only to overfit–they will increase the complexity of the combined classifier but cannot improve its performance on the training data. As we saw in Section 3.3.2, experiments with AdaBoost displayed the opposite phenomenon: the test error of the combined classifier continues to decrease after its training error reaches zero [20, 53]. For example, the left side of Figure 2 shows the training and test curves of running AdaBoost on top of C4.5 on the "letter" dataset [52].

These observations seem to contradict Occam's Razor, one of the fundamental principles in the theory and practice of machine learning (see, for instance, Chapter 2 of Kearns and Vazirani [44]). The principle states that in order to achieve good test error, the classifier should be as simple as possible. By "simple" we mean that the classifier is chosen from a restricted space of classifiers. When the space is finite, we use its cardinality as the measure of complexity and when it is infinite we use the VC-dimension [72] which is often closely related to the number of parameters that define the classifier. Typically, in both theory and practice, the difference between the training error and test error increases when the complexity of the classifier increases [64].

Indeed, Freund and Schapire predicted this typical behavior for AdaBoost when they first developed the algorithm [31]. They showed that, for a combined hypothesis H generated from a training sample S of size m using a weak learner whose hypothesis space has VC-dimension d, with high probability the generalization error of H is bounded by

$$\Pr_{\mathcal{D}}[H(x) \neq y] \leq \Pr_S[H(x) \neq y] + \tilde{O}\left(\sqrt{\frac{Td}{m}}\right),$$

where $\Pr_S[\cdot]$ denotes the empirical probability on the sample S (Eq. (3.5), Section 3.2.4). This bound suggests that boosting will overfit if run for too many rounds (as T becomes large).

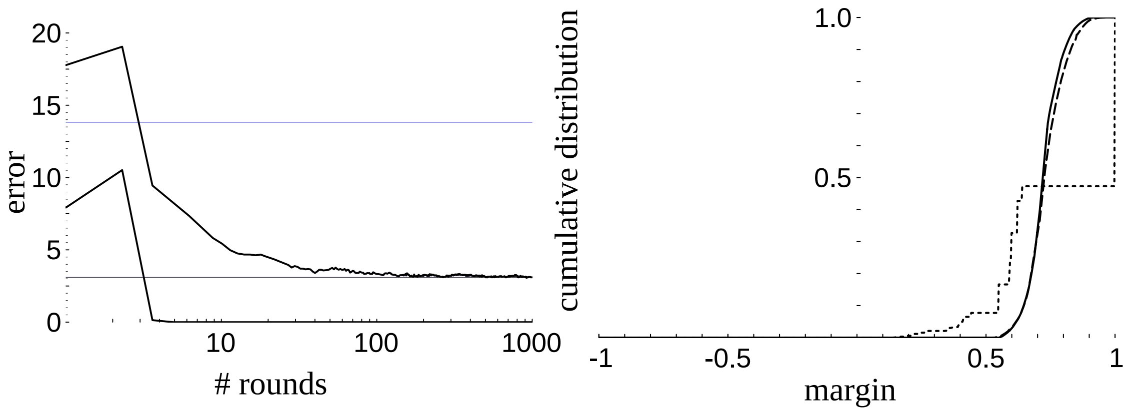

Figure 3-2: Error curves and margin distribution graphs for boosting C4.5 on the letter dataset as reported by Schapire et al. [64]. *Left:* The training and test error curves (lower and upper curves, respectively) of the combined classifier as a function of the number of rounds of boosting. The horizontal lines indicate the test error rate of the (first) base classifier as well as the test error of the final combined classifier. *Right:* The cumulative distribution of margins of the training examples after 5, 100, and 1000 iterations, included by short-dashed, long-dashed (mostly hidden), and solid curves, respectively. (Reprinted from Schapire [63].)

Faced with the contradicting experimental evidence that, when run for a large number of rounds, boosting often does not overfit and continues to improve the test of its combined classifier [12, 20, 53], Freund and Schapire sought a revised analysis of the generalization error of AdaBoost. Together with Bartlett and Lee, they formulated the margins analysis of boosting [64]. We follow Schapire's summary [63], using the notation of the AdaBoost pseudocode of Figure 3-1.

Recall that the combined hypothesis H output by AdaBoost cclassifies an instance x as $H(x) = \mathrm{sign}(f(x)) = \mathrm{sign}(\sum_{t=1}^{T} \alpha_t h_t(x))$. Define the *margin* of an example (x, y), where $y \in \{-1, +1\}$, as

$$\mathrm{margin}_f(x, y) = \frac{yf(x)}{\sum_{t=1}^{T} |\alpha_t|} = \frac{y \sum_{t=1}^{T} \alpha_t h_t(x)}{\sum_{t=1}^{T} |\alpha_t|}.$$

It is a number in $[-1, +1]$ that is positive if and only if H correctly classifies the example. Moreover, the magnitude of the margin can be interpreted as a measure of confidence in the prediction of H. Schapire et al. proved that larger margins on the training set translate into a superior upper bound on the generalization error. Specifically, for any $\theta > 0$, with high probability (over the choice of training set) the generalization error is at most

$$\mathrm{Pr}_S \left[\mathrm{margin}_f(x, y) \leq \theta\right] + \tilde{O}\left(\sqrt{\frac{d}{m\theta^2}}\right).$$

Unlike the previous generalization error bound, this bound is entirely independent of T, the number of rounds of boosting. In addition, Schapire et al. proved that boosting is particularly aggressive at reducing the margin (in a quantifiable sense) since it concentrates on the examples with the smallest margins.

Boosting's effect on the margins can be seen empirically, as shown on the right side of Figure 3-2, which shows the cumulative distribution of the margins of the training example on the "letter" dataset. In this case, even after the training error reaches zero, boosting continues to increase the margins of the training examples, effecting a corresponding drop in the test error.

Several authors [11, 35, 51] have made attempts to use the insights gleaned from the theory of margins. For example, Grove and Schuurmans [35] boosted decision trees using a linear programming algorithm for directly maximizing the minimum margin. They found that although this algorithm achieved a better minimum margin than AdaBoost, its test error was *worse*. They also reported experiments where AdaBoost did eventually overfit the training data (as observed by other researchers [29, 49, 53]) after a large number of rounds (e.g., 1000). They concluded that the margins theory alone does not accurately describe empirical observation.

In contrast, Mason, Bartlett, and Baxter [51] successfully applied the margins theory to derive a new boosting algorithm that achieves better empirical test error than AdaBoost. Rather than only minimizing training error as does AdaBoost, their algorithm instead optimizes a cost function which involves both the training error and the complexity of the combined classifier. Experiments with their algorithm, which is a gradient descent method, revealed that it sometimes avoids reducing training error in favor of reducing the complexity of the combined classifier, which is impossible behavior for AdaBoost.

In summary, the margins analysis of AdaBoost produced further insight into the strange resistance of boosting to overfitting. In addition, Schapire notes that the margin theory points to a strong connection between boosting and the support-vector machines of Vapnik and others [9, 16, 72] which explicitly attempt to maximize the minimum margin.

3.4.3 A game-theoretic interpretation

The behavior of AdaBoost can also be understood in a game-theoretic setting as explored by Freund and Schapire [30, 32] (see also Grove and Schuurmans [35] and Breiman [12]). We follow Schapire's summary [63].

In classical game theory, it is possible to put any two-person, zero-sum game in the form of a matrix $\mathbf{M}$ (see, for instance, Fudenberg and Tirole [34]). To play the game, one player chooses a row i and the other player chooses a column j. The loss of the row player (which is the same as the payoff to the column player) is $\mathbf{M}_{ij}$. More generally, the two sides may play randomly, choosing distributions $\mathbf{P}$ and $\mathbf{Q}$ over rows and columns, respectively. The expected loss then is $\mathbf{P}^T\mathbf{MQ}$.

Boosting can be viewed as repeated play of a particular game matrix. Assume that the weak hypotheses are binary, and let $\mathcal{H} = \{h_1, \ldots, h_n\}$ be the entire hypothesis space. To simplify the discussion, we assume $\mathcal{H}$ is finite. For a fixed training set $(x_1, y_1), \ldots, (x_m, y_m)$, the game matrix $\mathbf{M}$ has m rows and n columns, where

$$\mathbf{M}_{ij} = \begin{cases} 1 & \text{if } h_j(x_i) = y_i \\ 0 & \text{otherwise.} \end{cases}$$

The row player now is the boosting algorithm, and the column player is the weak learner. The boosting algorithm's choice of a distribution D_t over training examples becomes a distribution $\mathbf{P}$ over rows of $\mathbf{M}$, while the weak learner's choice of a weak hypothesis h_t becomes the choice of a column j of $\mathbf{M}$ (note the similarity to Freund's majority-vote game (Section 3.2.3) and the original derivation of AdaBoost (Section 3.2.4)).

As an example of the connection between boosting and game theory, consider von Neumann's famous minmax theorem which states that

$$\max_{\mathbf{Q}} \min_{\mathbf{P}} \mathbf{P}^T\mathbf{M}\mathbf{Q} = \min_{\mathbf{P}} \max_{\mathbf{Q}} \mathbf{P}^T\mathbf{M}\mathbf{Q}$$

for any matrix $\mathbf{M}$. When applied to the matrix just defined and reinterpreted in the boosting setting, this can be shown to have the following meaning: if, for any distribution over the training examples, there exists a weak hypothesis with error at most $\frac{1}{2} - \gamma$, then there exists a convex combination of weak hypotheses with a margin of at least 2γ on all training examples. As explained in Section 3.4.2, AdaBoost seeks to find such a final hypothesis with high margin on all examples by combining many weak hypotheses; so in a sense, the minmax theorem tells us that AdaBoost has at least the potential for success since, given a "good" weak learner, there must exist a good combination of weak hypotheses.

Going much further, AdaBoost can be shown to be a special case of a more general algorithm for playing repeated games, or for approximately solving matrix games. This shows that, asymptotically, the distribution over training examples as well as the weights over weak hypotheses in the final hypothesis have game-theoretic interpretations as approximate minmax or maxmin strategies.

3.4.4 Estimating probabilities

Let us return to definition of the AdaBoost algorithm. As we saw in Section 3.2.4, AdaBoost is a greedy method for choosing α_t in order to minimize Eq. (3.4):

$$\sum_{i=1}^{m} \exp(-y_i f(x_i)) = \sum_{i=1}^{m} \exp\left(-y_i \sum_{t=1}^{T} \alpha_t h_t(x_i)\right).$$

Starting from this interpretation, Friedman, Hastie, and Tibshirani [33] recently offered a statistical view of boosting. The following summary of their work in based on Schapire's presentation [63].

So far we have considered the classification problem of predicting the label y of an instance x with the intention of minimizing the probability of an incorrect prediction; we have assumed that x is assigned y by a *deterministic* labelling function. A related useful problem arises when we modify this assumption: perhaps x can receive different labels, according to a non-deterministic labelling function. In this case we are called upon to estimate the *probability* that a given instance x has label y. For example, given the observation (x) that Tracy has a headache, what is the probability that the headache stems from a disease (y_1), as opposed to the fact that she skipped breakfast this morning (y_2)? Friedman, Hastie, and Tibshirani suggested a method for using the output of AdaBoost to make reasonable estimates of such probabilities. Specifically, they suggest using a logistic function, and estimating

$$\Pr_f[y=+1|x] = \frac{\exp(f(x))}{\exp(f(x))+\exp(-f(x))} \tag{3.6}$$

where, as usual, $f(x)$ is the weighted average of weak hypotheses generated by AdaBoost. The rationale for this choice is the close connection between the log loss (negative log likelihood) of such a model, namely,

$$\sum_{i=1}^{m} \ln\left(1+\exp\left(-2y_i f(x_i)\right)\right) \tag{3.7}$$

and the function that, as we have already noted, AdaBoost attempts to minimize:

$$\sum_{i=1}^{m} \exp(-y_i f(x_i)). \tag{3.8}$$

Specifically, it can be verified that Eq. (3.8) is an upper bound on Eq. (3.7). In addition, if we add the constant $1 - \ln 2$ to Eq. (3.7) (which does not affect its minimization), then it can be verified that the resulting function and the one in Eq. (3.8) have identical Taylor expansions around zero up to second order; thus, their behavior near zero is very similar. Finally, it can be shown that, for any distribution over pairs (x, y), the expected log loss,

$$\mathrm{E}\left[\ln(1+\exp(-2yf(x)))\right],$$

and the expected error of AdaBoost,

$$\mathrm{E}\left[\exp(-yf(x))\right],$$

are minimized by the same function f, namely,

$$f(x) = \tfrac{1}{2}\ln\left(\frac{\Pr[y=+1|x]}{\Pr[y=-1|x]}\right).$$

Thus, for all these reasons, minimizing Eq. (3.8), as is done by AdaBoost, can be viewed as a method of approximately minimizing the negative log likelihood given in Eq. (3.7). Therefore, we may expect Eq. (3.6) to give a reasonable probability estimate.

Friedman, Hastie, and Tibshirani also make other connections between AdaBoost, logistic regression, and additive models. In particular, they derive new boosting algorithms based on these connections. The new algorithms exhibit comparable or superior performance to AdaBoost, as demonstrated in their experiments with decision trees.

3.4.5 Boosting in the presence of noise

Quinlan [53] was the first to notice that AdaBoost sometimes increased the error of a base learner (decision trees, in his case). When he presented this work at his AAAI'96 talk, he mentioned that he believed the poor performance of the algorithm was due to noise, meaning mislabelled training examples. Dietterich [18] confirmed this observation, finding that bagging outperformed AdaBoost when given noisy training data. Maclin and Opitz [49] made similar findings for neural networks: when they introduced noise into the data, AdaBoost produced classifiers with test error worse than arc-x4 [12] and much worse than bagging [10].

Bauer and Kohavi [7] reported detailed behavior of boosting in the presence of noise. When using decision trees, they found that AdaBoost performed worse than the single (best) tree on 4 out of 14 datasets, where 2 of the 4 definitely contained mislabelled training data. They listed this problem first on their list of open problems:

> The main problem with boosting seems to be [lack of] robustness to noise. We attempted to drop instances with very high weight but the experiments did not show this to be a successful approach. Should smaller sample sizes be used to force theories to be simple if tree sizes grow as trials progress? Are there methods to make boosting algorithms more robust when the dataset is noisy?

When he observed this phenomenon in his own experiments, Quinlan tried using confidence estimates to improve the performance of AdaBoost. Specifically, suppose base classifier h_t classifies instance x with an associated confidence $C_t(x)$ (between 0 and 1). Quinlan suggested that, in the combined hypothesis, rather than vote with weight α_t on instance x, h_t should modify the strength of its vote according to $C_t(x)$. He implemented a scheme for doing so, based on the Laplace ratio, and found that this improved the performance of AdaBoost on all but one of the noisy datasets. He concluded, "This modification is necessarily ad-hoc, since the confidence estimate...has only an intuitive meaning."

An approach to this problem has arisen from AdaBoost's predecessor, the Boost-By-Majority algorithm. As noted by Freund and Schapire [28], "...boost-by-majority is quite different than AdaBoost in the way it treats outliers, which suggests that it might be worth exploring experimentally." See a very recent paper by Freund [27] for the next chapter in the story which includes a new boosting algorithm, BrownBoost.

Chapter 4

The Algorithm RankBoost

In this chapter we introduce our formal model of the ranking problem and present a boosting algorithm, which we call RankBoost, for solving it.

4.1 A formal model of the ranking problem

In this section, we describe our formal model for studying ranking. Let X be a set called the *domain* or *instance space*. Elements of X are called *instances*. For example, in the movie-ranking task, each movie is an instance.

A learning algorithm in our model accepts as input a set of *ranking features* $f_1, \ldots, f_n$, which are functions. These are intended to provide a base level of information about the ranking task. Said differently, the learner's job will be to learn a ranking expressible in terms of the primitive ranking features, similar to ordinary features in more conventional learning settings. In one formulation of the movie task, each ranking feature corresponds to a single viewer's past ratings of movies.

Formally, each ranking feature f_i is a function of the form $f_i : X \to \overline{\mathbb{R}}$. The set $\overline{\mathbb{R}}$ consists of all real numbers, plus one additional element $\perp$ that indicates that no ranking is given and which is defined to be incomparable to all real numbers. For two instances x_0 and x_1, we interpret $f_i(x_1) > f_i(x_0)$ to mean that x_1 is ranked higher than x_0 by f_i. If $f_i(x) = \perp$ then x is unranked by f_i. For the movie ranking task, $f_i(x)$ is simply the numerical rating provided by movie-viewer i on movie x, or $\perp$ if the movie was not rated.

The final input to the learning algorithm is a *feedback function* Φ. This function encodes known relative ranking information about a subset of the instances. Typically, the learner will try to approximate Φ to produce a ranking of unseen instances. For the movie task, the feedback consists of Alice's preferences among the movies she has already seen.

Formally, we assume the feedback function has the form $\Phi : X \times X \to \mathbb{R}$ with the interpretation that $\Phi(x_0, x_1)$ represents how important it is that x_1 be ranked above x_0. Positive values mean that x_1 should be ranked above x_0 while negative values mean the opposite; a value of zero indicates no preference between x_0 and x_1. Consistent with this interpretation, we assume that $\Phi(x, x) = 0$ for all $x \in X$, and that Φ is anti-symmetric in the sense that $\Phi(x_0, x_1) = -\Phi(x_1, x_0)$ for all $x_0, x_1 \in X$. Note, however, that we do not assume transitivity of the feedback function.

For the movie task, we can define $\Phi(x_0, x_1)$ to be $+1$ if movie x_1 was preferred to movie x_0 by the current viewer, -1 if the opposite was the case, and 0 if either of the movies was not seen or if they were equally rated.

We generally assume that the support of Φ is finite. Let X_Φ denote the set of *feedback instances*, i.e., those instances that occur in the support of Φ:

$$X_\Phi = \{x \in X \mid \exists x' \in X : \Phi(x, x') \neq 0\}.$$

Also, let $|\Phi|$ be the size of the support of Φ:

$$|\Phi| = |\{(x_0, x_1) \in X \times X \mid \Phi(x_0, x_1) \neq 0\}|.$$

In some settings, it may be appropriate for the learner to accept a set of feedback functions $\Phi_1, \ldots, \Phi_m$. However, all of these can be combined into a single function Φ simply by adding them: $\Phi = \sum_j \Phi_j$. (If some have greater importance than others, then a weighted sum can be used.)

Formally, we require the learner to output a ranking of all instances represented in the form of a function $H : X \to \mathbb{R}$ with a similar interpretation to that of the ranking features, i.e., x_1 is ranked higher than x_0 by H if $H(x_1) > H(x_0)$. For the movie task, this corresponds to a complete ordering of all movies (with possible ties allowed).

The goal of the learner is to produce a "good" ranking of all instances, including those not observed in training. For instance, for the movie task, we would like to find a ranking of all movies that accurately predicts which ones a movie-viewer will like more or less than others; obviously, this ranking should include movies that the viewer has not already seen. As in other learning settings, how well the learning system performs on unseen data depends on many factors, such as the number of instances covered in training and the representational complexity of the ranking produced by the learner.

There are various methods that can be used to evaluate such a ranking. Some of these are discussed in Chapter 5. The boosting algorithm described in the next section attempts to minimize one possible measure called the ranking loss.

4.2 A boosting algorithm for the ranking task

In this section, we describe an approach to the ranking problem based on a machine learning method called boosting, in particular, Freund and Schapire's [31] AdaBoost algorithm and its successor developed by Schapire and Singer [65]. Boosting is a method of producing highly accurate prediction rules by combining many "weak" rules which may be only moderately accurate.

In the current setting, we seek a learning algorithm that will produce a function $H : X \to \mathbb{R}$ whose induced ordering of X will approximate the relative orderings encoded by the feedback function Φ. To formalize this goal, let $D(x_0, x_1) = c \cdot \max\{0, \Phi(x_0, x_1)\}$ so that all negative entries of Φ (which carry no additional information) are set to zero. Here, c is a

Algorithm **RankBoost**
Given: initial distribution D over $X \times X$.
Initialize: $D_1 = D$.
For $t = 1, \ldots, T$:

- Train weak learner using distribution D_t.
- Get weak hypothesis $h_t : X \to \mathbb{R}$.
- Choose $\alpha_t \in \mathbb{R}$.
- Update: $D_{t+1}(x_0, x_1) = \dfrac{D_t(x_0, x_1) \exp\left(\alpha_t(h_t(x_0) - h_t(x_1))\right)}{Z_t}$
  where Z_t is a normalization factor (chosen so that D_{t+1} will be a distribution).

Output the final hypothesis: $H(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$.

Figure 4-1: The RankBoost algorithm.

positive constant chosen so that

$$\sum_{x_0, x_1} D(x_0, x_1) = 1.$$

(When a specific range is not specified on a sum, we always assume summation over all of X.) A pair x_0, x_1 is said to be *crucial* if $\Phi(x_0, x_1) > 0$ so that the pair receives non-zero weight under D.

Our boosting algorithm is designed to find an H with a small weighted number of crucial-pair misorderings, namely,

$$\sum_{x_0, x_1} D(x_0, x_1) \llbracket H(x_1) \leq H(x_0) \rrbracket = \Pr_{(x_0, x_1) \sim D}\left[H(x_1) \leq H(x_0)\right]. \tag{4.1}$$

Here and throughout this paper, we define $\llbracket \pi \rrbracket$ to be 1 if predicate π holds and 0 otherwise. We call the quantity in Eq. (4.1) the *ranking loss* and we denote it by $\mathrm{rloss}_D(H)$.

4.2.1 The RankBoost algorithm

We call our boosting algorithm RankBoost, and its pseudocode is shown in Figure 4-1. Like all boosting algorithms, RankBoost operates in rounds. We assume access to a separate procedure called the *weak learner* that, on each round, is called to produce a *weak hypothesis*. RankBoost maintains a distribution D_t over $X \times X$ that is passed on round t to the weak learner. Intuitively, RankBoost chooses D_t to emphasize different parts of the training data. A high weight assigned to a pair of instances indicates a great importance that the weak learner order that pair correctly.

Weak hypotheses have the form $h_t : X \to \mathbb{R}$. We think of these as providing ranking information in the manner described above. The weak learner we used in our experiments is based on the given ranking features; details are given in Section 4.3.

The boosting algorithm uses the weak hypotheses to update the distribution as shown in

Figure 4-1. Suppose that x_0, x_1 is a crucial pair so that we want x_1 to be ranked higher than x_0 (in all other cases, D_t will be zero). Assuming for the moment that the parameter $\alpha_t > 0$ (as it usually will be), this rule decreases the weight $D_t(x_0, x_1)$ if h_t gives a correct ranking ($h_t(x_1) > h_t(x_0)$) and increases the weight otherwise. Thus, D_t will tend to concentrate on the pairs whose relative ranking is hardest to determine. The actual setting of α_t will be discussed shortly.

The *final* or *combined hypothesis* H is a weighted sum of the weak hypotheses. In the following theorem we prove a bound on the ranking loss of H on the training set. This theorem also provides guidance in choosing α_t and in designing the weak learner as we discuss below. As in standard classification problems, the loss on a separate test set can also be theoretically bounded given appropriate assumptions using uniform-convergence theory [17, 38, 71] . In Section 4.4 we will derive one such bound on the ranking generalization error of H and explain why the classification generalization error bounds do not trivially carry over to the ranking setting.

Theorem 1 *Assuming the notation of Figure 4-1, the ranking loss of H is*

$$\mathrm{rloss}_D(H) \leq \prod_{t=1}^{T} Z_t \ .$$

Proof: Unraveling the update rule, we have that

$$D_{T+1}(x_0, x_1) = \frac{D(x_0, x_1) \exp\left(H(x_0) - H(x_1)\right)}{\prod_t Z_t}.$$

Note that $[\![x \geq 0]\!] \leq e^x$ for all real x. Therefore, the ranking loss with respect to initial distribution D is

$$\begin{aligned} \sum_{x_0, x_1} D(x_0, x_1) [\![H(x_0) \geq H(x_1)]\!] &\leq \sum_{x_0, x_1} D(x_0, x_1) \exp\left(H(x_0) - H(x_1)\right) \\ &= \sum_{x_0, x_1} D_{T+1}(x_0, x_1) \prod_t Z_t = \prod_t Z_t. \end{aligned}$$

This proves the theorem. ■

Note that our methods for choosing α_t, which are presented in the next section, guarantee that $Z_t \leq 1$. Note also that RankBoost generally requires $O(|\Phi|)$ space and time per round.

4.2.2 Choosing α_t and criteria for weak learners

In view of the bound established in Theorem 1, we are guaranteed to produce a combined hypothesis with low ranking loss if on each round t we choose α_t and the weak learner constructs h_t so as to minimize

$$Z_t = \sum_{x_0, x_1} D_t(x_0, x_1) \exp\left(\alpha_t (h_t(x_0) - h_t(x_1))\right).$$

Formally, RankBoost uses the weak learner as a black box and has no control over how it chooses its weak hypotheses. In practice, however, we are often faced with the task of implementing the weak learner, in which case we can design it to minimize Z_t.

There are various methods for achieving this end. Here we sketch three. Let us fix t and drop all t subscripts when clear from context. (In particular, for the time being, D will denote D_t rather than an initial distribution.)

First method. First and most generally, for any given weak hypothesis h, it can be shown that Z, viewed as a function of α, has a unique minimum which can be found numerically via a simple binary search (except in trivial degenerate cases). For details, see Section 6.2 of Schapire and Singer [65].

Second method. The second method of minimizing Z is applicable in the special case that h has range $\{0, 1\}$. In this case, we can minimize Z analytically as follows: For $b \in \{-1, 0, +1\}$, let

$$W_b = \sum_{x_0,x_1} D(x_0, x_1) \, [\![h(x_0) - h(x_1) = b]\!] \, .$$

Also, abbreviate W_{+1} by W_+ and W_{-1} by W_-. Then $Z = W_- e^{-\alpha} + W_0 + W_+ e^{\alpha}$. Using simple calculus, it can be verified that Z is minimized by setting

$$\alpha = \tfrac{1}{2} \ln \left(\frac{W_-}{W_+} \right) \tag{4.2}$$

which yields

$$Z = W_0 + 2\sqrt{W_- W_+}. \tag{4.3}$$

Thus, if we are using weak hypotheses with range restricted to $\{0, 1\}$, we should attempt to find h that tends to minimize Eq. (4.3) and we should then set α as in Eq. (4.2).

Third method. Here we consider weak hypotheses of intermediate generality, namely those with range $[0, 1]$. For these hypotheses, we can use a third method to setting α based on an approximation of Z. Specifically, by the convexity of $e^{\alpha x}$ as a function of x, it can be verified that

$$e^{\alpha x} \le \left(\frac{1+x}{2} \right) e^{\alpha} + \left(\frac{1-x}{2} \right) e^{-\alpha}$$

for all real α and $x \in [-1, +1]$. Thus, we can approximate Z by

$$\begin{aligned} Z &\le \sum_{x_0,x_1} D(x_0, x_1) \left[\left(\frac{1 + h(x_0) - h(x_1)}{2} \right) e^{\alpha} + \left(\frac{1 - h(x_0) + h(x_1)}{2} \right) e^{-\alpha} \right] \\ &= \left(\frac{1-r}{2} \right) e^{\alpha} + \left(\frac{1+r}{2} \right) e^{-\alpha} \end{aligned} \tag{4.4}$$

where

$$r = \sum_{x_0,x_1} D(x_0, x_1)(h(x_1) - h(x_0)). \tag{4.5}$$

The right hand side of Eq. (4.4) is minimized when

$$\alpha = \tfrac{1}{2} \ln \left(\frac{1+r}{1-r} \right) \tag{4.6}$$

which, plugging into Eq. (4.4), yields $Z \leq \sqrt{1-r^2}$. Thus, to approximately minimize Z using weak hypotheses with range $[0,1]$, we can attempt to maximize $|r|$ as defined in Eq. (4.5) and then set α as in Eq. (4.6).

We now consider the case when any of these three methods for setting α assign a weak hypothesis h a weight $\alpha < 0$. For example, according to Eq. (4.2), α is negative if W_+, the weight of misordered pairs, is greater than W_-, the weight of correctly ordered pairs. Similarly for Eq. (4.6), $\alpha < 0$ if $r < 0$ (note that $r = W_- - W_+$). Intuitively, this means that h is negatively correlated with the feedback; the reverse of its predicted order will better approximate the feedback. RankBoost allows such weak hypotheses and its update rule reflects this intuition: the weights of the pairs that h correctly orders are *increased*, and the weights of the incorrect pairs are *decreased*.

4.2.3 An efficient implementation for bipartite feedback

In this section, we describe a more efficient implementation of RankBoost for feedback of a special form. We say that the feedback function is *bipartite* if there exist disjoint subsets X_0 and X_1 of X such that Φ ranks all instances in X_1 above all instances in X_0 and says nothing about any other pairs. That is, formally, for all $x_0 \in X_0$ and all $x_1 \in X_1$ we have that $\Phi(x_0, x_1) = +1$, $\Phi(x_1, x_0) = -1$ and Φ is zero on all other pairs.

Such feedback arises naturally, for instance, in document rank-retrieval tasks common in the field of information retrieval. Here, a set of documents may have been judged to be relevant or irrelevant. A feedback function that encodes these preferences will be bipartite. The goal of an algorithm for this task is to discover the relevant documents and present them to a user. Rather than output a classification of documents as relevant or irrelevant, the goal here is to output a ranked list of all documents that tends to place all relevant documents near the top of the list. One reason a ranking is preferred over a hard classification is that a ranking expresses the algorithm's confidence in its predictions. Another reason is that typically users of ranked-retrieval systems do not have the patience to examine every document that was predicted as relevant, especially if there is large number of such documents. A ranking allows the system to guide the user's decisions about which documents to read.

If RankBoost is implemented naively as in Section 4.2.2, then the space and time-per-round requirements will be $O(|X_0|\ |X_1|)$. In this section, we show how this can be improved to $O(|X_0| + |X_1|)$. Note that, in this section, $X_\Phi = X_0 \cup X_1$.

The main idea is to maintain a set of weights v_t over X_Φ (rather than the two-argument distribution D_t), and to maintain the condition that, on each round,

$$D_t(x_0, x_1) = v_t(x_0) v_t(x_1) \tag{4.7}$$

for all crucial pairs x_0, x_1 (recall that D_t is zero for all other pairs).

The pseudocode for this implementation is shown in Figure 4-2. Eq. (4.7) can be proved

Algorithm **RankBoost.B**
Given: disjoint subsets X_0 and X_1 of X.
Initialize:

$$v_1(x) = \begin{cases} 1/|X_1| & \text{if } x \in X_1 \\ 1/|X_0| & \text{if } x \in X_0 \end{cases}$$

For $t = 1, \ldots, T$:

- Train weak learner using distribution D_t (as defined by Eq. (4.7)).
- Get weak hypothesis $h_t : X \to \mathbb{R}$.
- Choose $\alpha_t \in \mathbb{R}$.
- Update:

$$v_{t+1}(x) = \begin{cases} \dfrac{v_t(x) \exp\left(-\alpha_t \, h_t(x)\right)}{Z_t^1} & \text{if } x \in X_1 \\ \dfrac{v_t(x) \exp\left(\alpha_t \, h_t(x)\right)}{Z_t^0} & \text{if } x \in X_0 \end{cases}$$

  where Z_t^1 and Z_t^0 normalize v_t over X_1 and X_0:

$$\begin{aligned} Z_t^1 &= \sum_{x \in X_1} v_t(x) \exp(-\alpha_t h_t(x)) \\ Z_t^0 &= \sum_{x \in X_0} v_t(x) \exp(\alpha_t h_t(x)) \end{aligned}$$

Output the final hypothesis: $H(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$.

Figure 4-2: A more efficient version of RankBoost for bipartite feedback.

by induction on t. It clearly holds initially. Using our inductive hypothesis, it is straightforward to expand the computation of $Z_t = Z_t^0 \cdot Z_t^1$ in Figure 4-2 to see that it is equivalent to the computation of Z_t in Figure 4-1. To show that Eq. (4.7) holds on round $t+1$, we have, for crucial pair x_0, x_1:

$$\begin{aligned} D_{t+1}(x_0, x_1) &= \frac{D_t(x_0, x_1) \exp\left(\alpha_t (h_t(x_0) - h_t(x_1))\right)}{Z_t} \\ &= \frac{v_t(x_0) \exp\left(\alpha_t h_t(x_0)\right)}{Z_t^0} \cdot \frac{v_t(x_1) \exp\left(-\alpha_t h_t(x_1)\right)}{Z_t^1} \\ &= v_{t+1}(x_0) \cdot v_{t+1}(x_1). \end{aligned}$$

Finally, note that all space requirements and all per-round computations are $O(|X_0| + |X_1|)$, with the possible exception of the call to the weak learner. However, if we want the weak learner to maximize $|r|$ as in Eq. (4.5), then we also only need to pass $|X_\Phi|$ weights to the weak learner, all of which can be computed in time linear in $|X_\Phi|$. Omitting t subscripts, and defining

$$s(x) = \begin{cases} +1 & \text{if } x \in X_1 \\ -1 & \text{if } x \in X_0 \end{cases},$$

we can rewrite r as

$$
\begin{aligned}
r &= \sum_{x_0,x_1} D(x_0,x_1)(h(x_1)-h(x_0)) \\
&= \sum_{x_0\in X_0}\sum_{x_1\in X_1} v(x_0)v(x_1)\left(h(x_1)s(x_1)+h(x_0)s(x_0)\right) \\
&= \sum_{x_0\in X_0}\left(v(x_0)\sum_{x_1\in X_1} v(x_1)\right) s(x_0)\,h(x_0) + \sum_{x_1\in X_1}\left(v(x_1)\sum_{x_0\in X_0} v(x_0)\right) s(x_1)\,h(x_1) \\
&= \sum_x d(x)s(x)h(x) \qquad (4.8)
\end{aligned}
$$

where

$$d(x) = v(x) \sum_{x' : s(x)\neq s(x')} v(x') \ .$$

All of the weights $d(x)$ can be computed in linear time by first computing the sums that appear in this equation for the two possible cases that x is in X_0 or X_1. Thus, we only need to pass $|X_0| + |X_1|$ weights to the weak learner in this case rather than the full distribution D_t of size $|X_0|\ |X_1|$.

4.3 Weak hypotheses for ranking

As described in Section 4.2, our algorithm RankBoost requires access to a weak learner to produce weak hypotheses. In this section, we describe an efficient implementation of a weak learner for ranking.

Perhaps the simplest and most obvious weak learner would find a weak hypothesis h that is equal to one of the ranking features f_i, except on unranked instances. For example, recall that a ranking features for the movie task is a movie viewer. That is,

$$h(x) = \begin{cases} f_i(x) & \text{if } f_i(x) \in \mathbb{R} \\ q_{\text{def}} & \text{if } f_i(x) = \perp \end{cases}$$

for some $q_{\text{def}} \in \mathbb{R}$.

Although perhaps appropriate in some settings, the main problem with such a weak learner is that it depends critically on the *actual values* defined by the ranking features, rather than relying exclusively on the relative-ordering information which they provide. We believe that learning algorithms of the latter form will be much more general and applicable. Such methods can be used even when features provide only an ordering of instances and no scores or other information are available. Such methods also side-step the issue of combining ranking features whose associated scores have different semantics (such as the different scores assigned to URL's by different search engines).

For these reasons, we focus in this section and in our experiments on $\{0,1\}$-valued weak hypotheses that use the ordering information provided by the ranking features, but ignore specific scoring information. In particular, we will use weak hypotheses h of the form

$$h(x) = \begin{cases} 1 & \text{if } f_i(x) > \theta \\ 0 & \text{if } f_i(x) \le \theta \\ q_{\text{def}} & \text{if } f_i(x) = \perp \end{cases} \tag{4.9}$$

where $\theta \in \mathbb{R}$ and $q_{\text{def}} \in \{0, 1\}$. That is, a weak hypothesis is derived from a ranking feature f_i by comparing the score of f_i on a given instance to a threshold θ. To instances left unranked by f_i, the weak hypothesis assigns the default score q_{def}. For the remainder of this section, we show how to choose the "best" feature, threshold and default score.

Since our weak hypotheses are $\{0, 1\}$-valued, we can use either the second or third methods described in Section 4.2.2 to guide us in our search for a weak hypothesis. We chose the third method because we can implement it more efficiently than the second. According to the second method, the weak learner should seek a weak hypothesis that minimizes Eq. (4.3). For a given candidate weak hypothesis, we can directly compute the quantities W_0, W_-, and W_+, as defined in Section 4.2.2, in $O(|\Phi|)$ time. Moreover, for each of the n ranking features, there are at most $|X_\Phi| + 1$ thresholds to consider (as defined by the range of f_i on X_Φ) and two possible default scores (0 and 1). Thus a straightforward implementation of the second method requires $O(n|\Phi||X_\Phi|)$ time to generate a weak hypothesis.

The third method of Section 4.2.2 requires maximizing $|r|$ as given by Eq. (4.5) and has the disadvantage that it is based on an approximation of Z. However, although a straightforward implementation also requires $O(n|\Phi||X_\Phi|)$ time, we will show how to implement it in $O(n|X_\Phi| + |\Phi|)$ time. (In the case of bipartite feedback, if the boosting algorithm of Section 4.2.3 is used, only $O(n|X_\Phi|)$ time is needed.) This is a significant improvement from the point of view of our experiments in which $|\Phi|$ was large.

We now describe a time and space efficient algorithm for maximizing $|r|$. Let us fix t and drop it from all subscripts to simplify the notation. We begin by rewriting r for a given D and h as follows:

$$\begin{aligned} r &= \sum_{x_0, x_1} D(x_0, x_1)\,(h(x_1) - h(x_0)) \\ &= \sum_{x_0, x_1} D(x_0, x_1) h(x_1) - \sum_{x_0, x_1} D(x_0, x_1) h(x_0) \\ &= \sum_x h(x) \sum_{x'} D(x', x) - \sum_x h(x) \sum_{x'} D(x, x') \\ &= \sum_x h(x) \sum_{x'} (D(x', x) - D(x, x')) \\ &= \sum_x h(x)\,\pi(x)\ , \end{aligned} \tag{4.10}$$

where we define $\pi(x) = \sum_{x'} (D(x', x) - D(x, x'))$ as the *potential* of x. Note that $\pi(x)$ depends only on the current distribution D. Hence, the weak learner can precompute all the potentials at the beginning of each boosting round in $O(|\Phi|)$ time and $O(|X_\Phi|)$ space. When the feedback is bipartite, comparing Eqs. (4.8) and (4.10), we see that $\pi(x) = d(x)s(x)$ where d and s are defined in Section 4.2.3; thus, in this case, π can be computed even faster in only

$O(|X_\Phi|)$ time.

Now let us address the problem of finding a good threshold value θ and default value $q_{\rm def}$. We need to scan the candidate ranking features f_i and evaluate $|r|$ (defined by Eq. (4.10)) for each possible choice of f_i, θ and $q_{\rm def}$. Substituting into Eq.(4.10) the h defined by Eq. (4.9), we have that

$$
\begin{aligned}
r &= \sum_{x:f_i(x)>\theta} h(x)\,\pi(x) + \sum_{x:f_i(x)\le\theta} h(x)\,\pi(x) + \sum_{x:f_i(x)=\perp} h(x)\,\pi(x) && (4.11)\\
&= \sum_{x:f_i(x)>\theta} \pi(x) + q_{\rm def} \sum_{x:f_i(x)=\perp} \pi(x). && (4.12)
\end{aligned}
$$

For a fixed ranking feature f_i, let $X_{f_i} = \{x \in X_\Phi \mid f_i(x) \neq \perp\}$ be the set of feedback instances ranked by f_i. We only need to consider $|X_{f_i}| + 1$ threshold values, namely, $\{f_i(x) \mid x \in X_{f_i}\} \cup \{-\infty\}$ since these define all possible behaviors on the feedback instances. Moreover, we can straightforwardly compute the first term of Eq. (4.12) for *all* thresholds in this set in time $O(|X_{f_i}|)$ simply by scanning down a presorted list of threshold values and maintaining the partial sum in the obvious way.

For each threshold, we also need to evaluate $|r|$ for the two possible assignments of $q_{\rm def}$ (0 or 1). To do this, we simply need to evaluate $\sum_{x:f_i(x)=\perp} \pi(x)$ once. Naively, this takes $O(|X_\Phi - X_{f_i}|)$ time, i.e., linear in the number of *unranked* instances. We would prefer all operations to depend instead on the number of ranked instances since, in applications such as meta-searching and information retrieval, each ranking feature may rank only a small fraction of the instances. To do this, note that $\sum_x \pi(x) = 0$ by definition of $\pi(x)$. This implies that

$$\sum_{x:f_i(x)=\perp} \pi(x) = -\sum_{x:f_i(x)\neq\perp} \pi(x). \tag{4.13}$$

The right hand side of this equation can clearly be computed in $O(|X_{f_i}|)$ time. Combining Eqs. (4.12) and (4.13), we have

$$r = \sum_{x:f_i(x)>\theta} \pi(x) - q_{\rm def} \sum_{x\in X_{f_i}} \pi(x). \tag{4.14}$$

The pseudocode for the weak learner is given in Figure 4-3. Note that the input to the algorithm includes for each feature a sorted list of candidate thresholds $\{\theta_j\}_{j=1}^J$ for that feature. For convenience we assume that $\theta_1 = \infty$ and $\theta_J = -\infty$. Also, the value $|r|$ is calculated according to Eq. (4.14): the variable L stores the left summand and the variable R stores the right summand. Finally, if the default rank $q_{\rm def}$ is specified by the user, then step 6 is skipped.

Thus, for a given ranking feature, the total time required to evaluate $|r|$ for all candidate weak hypotheses is only linear in the number of instances that are ranked by that feature. In summary, we have shown:

Theorem 2 *The algorithm of Figure 4-3 finds the weak hypothesis of the form given in Eq. (4.9) that maximizes $|r|$ as in Eq. (4.10). The running time is $O(n|\Phi||X_\Phi|)$ per round*

Algorithm **WeakLearn**

Given: distribution D over $X \times X$.
set of features $\{f_i\}_{i=1}^N$.
for each f_i, the set $X_{f_i} = \{x_k\}_{k=1}^K$ such that $f_i(x_1) \geq \ldots \geq f_i(x_K)$.
for each f_i, the set of candidate thresholds $\{\theta_j\}_{j=1}^J$ such that $\theta_1 \geq \ldots \geq \theta_J$.

Initialize: for all $x \in X_\Phi$, $\pi(x) = \sum_{x' \in X_\Phi} D(x', x) - D(x, x')$.
$r = 0$.

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

$L = 0$.
$R = \sum_{x \in X_{f_i}} \pi(x)$. /* $L - q_{\mathrm{def}} R$ is rhs of Eq. (4.14) */
$\theta_0 = \infty$.
For $j = 1, \ldots, J$:
  $L = L + \sum_{x: \theta_{j-1} \geq f_i(x) > \theta_j} \pi(x)$. /* compute $L = \sum_{x: f_i(x) > \theta} \pi(x)$ */
  if $|L| > |L - R|$ /* find best value for q_{def} */
    then $q = 0$.
    else $q = 1$.
  if $|L - q\,R| > |r|$ /* find best weak hypothesis */
    $r^* = L - q\,R$.
    $i^* = i$.
    $\theta^* = \theta_j$.
    $q^*_{\mathrm{def}} = q$.

Output weak hypothesis $(f_{i^*}, \theta, q_{\mathrm{def}})$.

Figure 4-3: The weak learner.

of boosting. An efficient implementation runs in time

$$O\left(|\Phi| + \sum_{i=1}^{n} |X_{f_i}|\right) = O(|\Phi| + n|X_\Phi|) .$$

If the feedback is bipartite, the running time can be improved to $O(n|X_\Phi|)$.

Positive cumulative weights. Since the final hypothesis has the form $H(x) = \sum_{t=1}^T \alpha_t h_t(x)$ and the hypotheses output by WeakLearn are binary, if $h_t(x) = 1$ then h_t contributes its weight α_t to the final score of x. During the boosting process, WeakLearn may output distinct hypotheses that correspond to different thresholds of the same feature f. If we view these hypotheses in increasing order by threshold, we see that f's contribution to the final score of x is the sum of the weights of the hypotheses whose thresholds are less than $f(x)$. To simplify matters, if we assume that h_t occurs exactly once among $h_1, \ldots, h_T$, then if the weights α_t are always positive, then f's contribution increases monotonically with the score it assigns instances.

This behavior of a feature's contribution being positively correlated with the score it assigns is desirable in some applications. In the meta-search task, it is natural that the search strategy f should contribute more weight to the final score of those instances that appear higher on its ranked list. Put another way, it would seem strange if, for example, f contributed more weight to instances in the middle of its list and less to those at either end, as would be the case if some of the α_t's were negative. Also, from the perspective of generalization error, if we allow some α_t's to be negative then we can construct arbitrary functions of the instance space by thresholding a single feature, and this is probably more complexity than we would like to allow in the combined hypothesis (in order to avoid overfitting).

To address this situation, we implemented an additional version of WeakLearn that chooses its hypotheses to exhibit this monotonic behavior. In practice, our earlier assumption that all h_t's are unique may not hold. If it doesn't, then the contribution of a particular hypothesis h will be its *cumulative* weight, the sum of those α_t's for which $h_t = h$. Thus we need to ensure that this cumulative weight is positive. Our implementation outputs the hypothesis that maximizes $|r|$ subject to the constraint that the cumulative weight of that hypothesis remains positive. We refer to this modified weak learner as WeakLearn.cum.

4.4 Generalization Error

In this section we derive a bound on the generalization error of the combined hypothesis when the weak hypotheses are binary functions and the feedback is bipartite. That is, we assume that the feedback partitions the instance space into two sets, X and Y, such that $\Phi(x, y) > 0$ for all $x \in X$ and $y \in Y$, meaning the instances in Y are ranked above those in X. Many problems can be viewed as providing bipartite feedback, including the meta-search and movie recommendation tasks described in Chapter 5, as well as many of the problems in information retrieval [57, 58].

4.4.1 Probabilistic Model

Up to this point we have not discussed where our train and test data comes from. The assumption of machine learning , as described in Section 3.1, is that there exists a fixed and unknown distribution over the instance space. The training set (and test set) is a set of independent samples according to this distribution. This model clearly translates to the classification setting, where the goal is predict the class of an instance. The training set consists of an independent sample of instances, where each instance is labeled with its correct class. A learning algorithm formulates a classification rule after running on the training set, and the rule is evaluated on the test set, which is a disjoint independent sample of unlabeled instances.

This probabilistic model does not translate as readily to the ranking setting, however, where the goal is to predict the order of a pair of instances. A natural approach for the bipartite case would be to assume a fixed and unknown distribution D over $(X \cup Y) \times (X \cup Y)$, pairs from the instance space.[1] The obvious next step would be to declare the training set

[1]Note that assuming a distribution over $X \times Y$ trivializes the ranking problem: the rule which always ranks the second instance over the first is perfect.

to be a collection of instances sampled independently at random according to D. The generalization results for classification would then trivially extend to ranking. The problem is that the pairs in the training set are *not* independent: if (x_1, y_1) and (x_2, y_2) are in the training set, then so are (x_1, y_2) and (x_2, y_1).

Here we present a revised approach that permits sampling independence assumptions. Rather than a single distribution D, we assume the existence of two distributions, D_0 over X and D_1 over Y. The training instances are the union of an independent sample according to D_0 and an independent sample according to D_1. (This is similar to the "two button" learning model in classification.) The training set, then, consists of all pairs of training instances.

Consider the movie recommendation task as an example of this model. The model suggests that movies viewed by a person can be partitioned into an independent sample of good movies and an independent sample of bad movies. This assumption is not entirely true since people usually choose which movies to view based on movies they've seen. However, such independence assumptions are common in machine learning (take for instance, the Naive-Bayes classifier [23]).

4.4.2 Sampling Error Definitions

Given this probabilistic model of the ranking problem, we can now define generalization error. The final hypothesis output by RankBoost has the form

$$H(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$$

and orders instances according to the scores it assigns them. We are concerned here with the predictions of such hypotheses on pairs of instances, so we consider hypotheses of the form $H : (X \cup Y) \times (X \cup Y) \to \{-1, 0, +1\}$, where

$$H(x, y) = \operatorname{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(y) - \sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

where the h_t come from some class of binary functions $\mathcal{H}$. Let $\mathcal{C}$ be the set of all such functions H.

H misorders $(x, y) \in X \times Y$ if $H(x, y) \neq 1$, which leads us to define the generalization error of H as

$$\begin{aligned} \varepsilon_g(H) &= \Pr_{x \sim D_0, y \sim D_1}\left[[\![H(x, y) \neq 1]\!]\right] \\ &= \mathrm{E}_{D_0, D_1}\left[[\![H(x, y) \neq 1]\!]\right] . \end{aligned}$$

We first verify that this definition is consistent with our notion of test error. For a given test sample $T_0 \times T_1$ where $T_0 = \langle x_1, \ldots, x_p \rangle$ and $T_1 = \langle y_1, \ldots, y_q \rangle$, the test error of H is

$$\begin{aligned}
\mathrm{E}_{T_0,T_1}\left[\frac{1}{pq}\sum_{i,j} [\![H(x_i,y_j) \neq 1]\!]\right] &= \frac{1}{pq}\sum_{i,j} \mathrm{E}_{T_0,T_1}\left[[\![H(x_i,y_j) \neq 1]\!]\right] \\
&= \frac{1}{pq}\sum_{i,j} \mathrm{Pr}_{x_i,y_j}\left[[\![H(x_i,y_j) \neq 1]\!]\right] \\
&= \frac{1}{pq}\sum_{i,j} \varepsilon_g(H) = \varepsilon_g(H) \ .
\end{aligned}$$

Similarly, if we have a training sample $S_0 \times S_1$ where $S_0 = \langle x_1, \ldots, x_m \rangle$ and $S_1 = \langle y_1, \ldots, y_n \rangle$, the training (or empirical) error of H is

$$\hat{\varepsilon}(H) = \frac{1}{mn}\sum_{i,j} [\![H(x_i,y_j) \neq 1]\!] \ .$$

Our goal is to show that, with high probability, the difference between $\hat{\varepsilon}(H)$ and $\varepsilon_g(H)$ is small, meaning that the performance of the combined hypothesis H on the training sample is representative of its performance any random sample.

4.4.3 VC analysis

We now bound the difference between the training error and test error of the combined hypothesis output by RankBoost using standard VC-dimension analysis techniques [17, 71]. We will show that, with high probability taken over the choice of training set, this difference is small for every $H \in \mathcal{C}$. If this happens then, no matter which combined hypothesis is chosen by our algorithm, the training error of the combined hypothesis will accurately estimate its generalization error. Another way of saying this is that the probability (over the choice of training set) is very small that there exists an $H \in \mathcal{C}$ such that $\hat{\varepsilon}(H)$ and $\varepsilon_g(H)$ differ by more than a small amount. In other words, we will show that for every $\delta > 0$, there exists a small ϵ such that

$$\mathrm{Pr}_{S_0 \sim D_0^m, S_1 \sim D_1^n}\left[\exists H \in \mathcal{C} \ : \ \left|\frac{1}{mn}\sum_{i,j} [\![H(x_i,y_j) \neq 1]\!] - \mathrm{E}_{x,y}\left[[\![H(x,y) \neq 1]\!]\right]\right| > \epsilon\right] \leq \delta \quad (4.15)$$

where the choice of ϵ will be determined during the course of the proof.

Our approach will be to separate (4.15) into two probabilities, one over the choice of S_0 and the other over the choice of S_1, and then to bound each of these using classification generalization error theorems.

In order to use these theorems we will need to convert H into a binary function. Define $F : X \times Y \to \{0, 1\}$ as a function which indicates whether or not H misorders the pair (x, y), meaning $F(x, y) = [\![H(x, y) \neq 1]\!]$. Although H is a function on $(X \cup Y) \times (X \cup Y)$, we only care about its performance on pairs $(x, y) \in X \times Y$, which is to say that it incurs no penalty for its ordering of two instances from either X or Y. The quantity inside the absolute value of (4.15) can then be rewritten as

$$\begin{aligned}
&\frac{1}{mn}\sum_{i,j} F(x_i, y_j) - \mathrm{E}_{x,y}\left[F(x,y)\right] \\
&= \frac{1}{mn}\sum_{i,j} F(x_i, y_j) - \frac{1}{m}\sum_i \mathrm{E}_y\left[F(x_i,y)\right] + \frac{1}{m}\sum_i \mathrm{E}_y\left[F(x_i,y)\right] - \mathrm{E}_{x,y}\left[F(x,y)\right] \\
&= \frac{1}{m}\sum_i \left(\frac{1}{n}\sum_j F(x_i, y_j) - \mathrm{E}_y\left[F(x_i,y)\right]\right) + \qquad (4.16) \\
&\quad \mathrm{E}_y\left[\frac{1}{m}\sum_i F(x_i, y) - \mathrm{E}_x\left[F(x,y)\right]\right] . \qquad (4.17)
\end{aligned}$$

So if we prove that there exist ϵ_0 and ϵ_1 such that $\epsilon_0 + \epsilon_1 = \epsilon$ and

$$\mathrm{Pr}_{S_1 \sim D_1^n}\left[\exists F \in \mathcal{F}, \exists x_i \in X : \left|\frac{1}{n}\sum_j F(x_i, y_j) - \mathrm{E}_y\left[F(x_i,y)\right]\right| > \epsilon_1\right] \leq \delta/2 \qquad (4.18)$$

$$\mathrm{Pr}_{S_0 \sim D_0^m}\left[\exists F \in \mathcal{F}, \exists y \in Y : \left|\frac{1}{m}\sum_i F(x_i, y) - \mathrm{E}_x\left[F(x,y)\right]\right| > \epsilon_0\right] \leq \delta/2 , \qquad (4.19)$$

we will have shown (4.15), because with high probability, the summand of (4.17) will be less than ϵ_1 for every x_i, which implies that the sum will be less than ϵ_1. Likewise, the quantity inside the expectation of (4.17) will be less than ϵ_0 for every y and so the expectation will be less than ϵ_0.

We now prove (4.19) using standard classification results, and (4.18) follows by a symmetric argument. Consider (4.19) for a fixed y, which means that $F(x, y)$ is a single argument binary-valued function. Let $\mathcal{F}_y$ be the set of all such functions F for a fixed y. Then the choice of F in (4.19) comes from $\bigcup_y \mathcal{F}_y$. A theorem of Vapnik [71] applies to (4.19) and gives a choice of ϵ_0 that depends on the size m of the training set S_0, the error probability δ, and the complexity d' of $\bigcup_y \mathcal{F}_y$, measured as its VC-dimension (for details, see Vapnik [71] or Devroye, Györfi, and Lugosi [17]). Specifically, for any $\delta > 0$,

$$\mathrm{Pr}_{S_0 \sim D_0^m}\left[\exists F \in \bigcup \mathcal{F}_y \; : \; \left|\frac{1}{m}\sum_i F(x_i, y) - \mathrm{E}_x\left[F(x,y)\right]\right| > \epsilon_0(m, \delta, d')\right] < \delta ,$$

where

$$\epsilon_0(m, \delta, d') = 2\sqrt{\frac{d'(\ln(2m/d') + 1) + \ln(9/\delta)}{m}} .$$

The parameters m and δ are given; it remains to calculate d', the VC-dimension of $\bigcup_y \mathcal{F}_y$. (We note that although we are using a classification result to bound (4.19), the probability correspond to a peculiar classification problem (trying to differentiate X from Y by picking an F and one $y \in Y$) that does seem to have a natural interpretation.)

Let's determine the form of the functions in $\bigcup_y \mathcal{F}_y$. For a fixed $y \in Y$,

$$F(x, y) \;\; = \;\; [\![H(x, y) \neq 1]\!]$$

$$
\begin{aligned}
&= \left[\!\left[\operatorname{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(y) - \sum_{t=1}^{T} \alpha_t h_t(x) \right) \neq 1 \right]\!\right] \\
&= \left[\!\left[\sum_{t=1}^{T} \alpha_t h_t(x) - \sum_{t=1}^{T} \alpha_t h_t(y) \geq 0 \right]\!\right] \\
&= \left[\!\left[\sum_{t=1}^{T} \alpha_t h_t(x) - b \geq 0 \right]\!\right]
\end{aligned}
$$

where $b = \sum_{t=1}^{T} \alpha_t h_t(y)$ is constant because y is fixed. So the functions in $\bigcup_y \mathcal{F}_y$ consist of all possible thresholds of all linear combinations of T weak hypotheses. Freund and Schapire's Theorem 8 [31] bounds the VC-dimension of this class in terms of T and the VC-dimension of the weak hypothesis class $\mathcal{H}$. Applying their result, we have that if $\mathcal{H}$ has VC-dimension $d \geq 2$, then d' is at most $2(d+1)(T+1)\log_2(e(T+1))$, where e is the base of the natural logarithm.

As the final step, repeating the same reasoning for (4.18) keeping x fixed, and putting it all together, we have that with probability $1-\delta$ over the choice of training sample, all $H \in \mathcal{C}$ satisfy

$$
|\hat{\varepsilon}(H) - \varepsilon_g(H)| < 2\sqrt{\frac{d'(\ln(2m/d') + 1) + \ln(18/\delta)}{m}} + 2\sqrt{\frac{d'(\ln(2n/d') + 1) + \ln(18/\delta)}{n}} ,
$$

where $d' = 2(d+1)(T+1)\log_2(e(T+1))$ and d is the VC-dimension of the class of weak hypotheses. ■

Chapter 5

Experimental evaluation of RankBoost

In this chapter, we report experiments with RankBoost on two ranking problems. The first is a simplified web meta-search task, the goal of which is to build a search strategy for finding homepages of machine-learning researchers and universities. The second task is a collaborative-filtering problem of making movie recommendations for a new user based on the preferences of other users.

In each experiment, we divided the available data into training data and test data, ran each algorithm on the training data, and evaluated the output hypothesis on the test data. Details are given below.

5.1 Meta-search

We first present experiments on learning to combine the results of several web searches. This problem exhibits many facets that require a general approach such as ours. For instance, approaches that combine similarity scores are not applicable since the similarity scores of web search engines often have different semantics or are unavailable.

5.1.1 Description of task and data set

In order to test RankBoost on this task, we used the data of Cohen, Schapire and Singer [15]. Their goal was to simulate the problem of building a domain-specific search engine. As test cases, they picked two fairly narrow classes of queries—retrieving the homepages of machine-learning researchers (ML), and retrieving the homepages of universities (UNIV). They chose these test cases partly because the feedback was readily available from the web. They obtained a list of machine-learning researchers, identified by name and affiliated institution, together with their homepages,[1] and a similar list for universities, identified by name and (sometimes) geographical location from Yahoo! We refer to each entry on these lists (i.e., a

[1]From 'http://www.aic.nrl.navy.mil/~aha/research/machine-learning.html'.

name-affiliation pair or a name-location pair) as a *base query*. The goal is to learn a meta-search strategy that, given a base query, will generate a ranking of URL's that includes the correct homepage at or close to the top.

Cohen, Schapire and Singer also constructed a series of special-purpose *search templates* for each domain. Each template specifies a query expansion method for converting a base query into a likely seeming AltaVista query which we call the *expanded query*. For example, one of the templates has the form `+"NAME" +machine +learning` which means that AltaVista should search for all the words in the person's name plus the words 'machine' and 'learning'. When applied to the base query 'Joe Researcher from Learning University' this template expands to the expanded query `+"Joe Researcher" +machine +learning`.

A total of 16 search templates were used for the ML domain and 22 for the UNIV domain.[2] Each search template was used to retrieve the top thirty ranked documents. If none of these lists contained the correct homepage, then the base query was discarded from the experiment. In the ML domain, there were 210 base queries for which at least one search template returned the correct homepage; for the UNIV domain, there were 290 such base queries.

We mapped the meta-search problem into our framework as follows. Formally, the instances now are pairs of the form (q, u) where q is a base query and u is one of the URL's returned by one of the search templates for this query. Each ranking feature f_i is constructed from a corresponding search template i by assigning the jth URL u on its list (for base query q) a rank of $-j$; that is, $f_i((q, u)) = -j$. If u was not ranked for this base query, then we set $f_i((q, u)) = \perp$. We also construct a separate feedback function Φ_q for each base query q that ranks the correct homepage URL u_* above all others. That is, $\Phi_q((q, u), (q, u_*)) = +1$ and $\Phi_q((q, u_*), (q, u)) = -1$ for all $u \neq u_*$. All other entries of Φ_q are set to zero. All the feedback functions Φ_q were then combined into one feedback function Φ by summing as described in Section 4.1: $\Phi = \sum_q \Phi_q$. Thus our feedback function is bipartite. As discussed in Section 4.2.3, even though our input feedback is binary, our goal is to output not a classification but a ranked list that places the homepages at the top of the list.

Given this mapping of the ranking problem into our framework, we can immediately apply RankBoost. This mapping implies that each weak hypothesis is defined by a search template i (corresponding to ranking feature f_i), and a threshold value θ. Given a base query q and a URL u, this weak hypothesis outputs 1 or 0 if u is ranked above or below the threshold θ on the list of URL's returned by the expanded query associated with search template i applied to base query q. As usual, the final hypothesis H is a weighted sum of the weak hypotheses. Thus, given a test base query q, we first form all of the expanded queries and send these to the search engine to obtain lists of URL's. We then evaluate H as above on each pair (q, u), where u is a returned URL, to obtain a predicted ranking of all of the URL's.

For evaluation, we divided the data into training and test sets using four-fold cross-validation. We created four partitions of the data, each one using 75% of the base queries for training and 25% for testing. Of course, the learning algorithms had no access to the test data during training.

[2]See Cohen, Schapire and Singer [15] for the list of search templates.

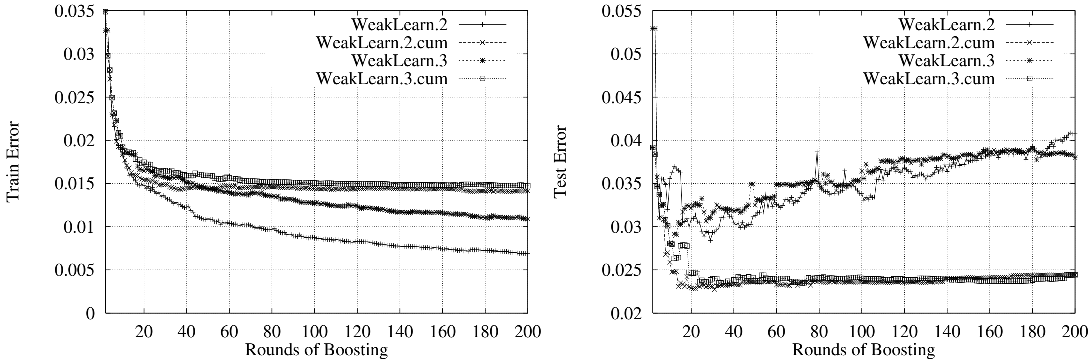

Figure 5-1: Performance of the four weak learners WeakLearn.{2,3,2.cum,3.cum} on the ML dataset. Left: Train error Right: Test error

5.1.2 Experimental parameters and evaluation

Since all search templates had access to the same set of documents, if a URL was not returned in the top 30 documents by a search template, we interpreted this as ranking the URL below all of the returned documents. Thus we set the parameter q_{def}, the default value for weak hypotheses, to be 0 (see Section 4.3).

Our implementation of RankBoost used a definition of ranking loss modified from the original given in Section 4.1, Eq. (4.1):

$$\text{rloss}_D(H) = \sum_{x_0, x_1} D(x_0, x_1) \, [\![H(x_1) \leq H(x_0)]\!] \ .$$

If the output hypothesis ranked as equal a pair (x_0, x_1) of instances that the feedback ranked as unequal, we assigned the hypothesis an error of 1/2 instead of 1. This represents the fact that if we used the hypothesis to produce an ordered list of documents, breaking ties randomly, then its expected error on (x_0, x_1) is 1/2, since the probability that x_0 is listed above x_1 is equal to the probability that x_1 is listed above x_0. The modified definition is

$$\text{rloss}_D(H) = \sum_{x_0, x_1} D(x_0, x_1) \, [\![H(x_1) < H(x_0)]\!] + \tfrac{1}{2} \sum_{x_0, x_1} D(x_0, x_1) \, [\![H(x_1) = H(x_0)]\!] \ . \qquad (5.1)$$

RankBoost parameters. Since WeakLearn outputs binary weak hypotheses, we can set the parameter α using either the second or third methods presented in Section 4.2.2. The second method sets α as the minimum of Z, and the third method sets α to approximately minimize Z. The third method can be implemented more easily and runs faster. We implemented both methods, called WeakLearn.2 and WeakLearn.3, to determine if the extra time required by the second method (almost ten times that of the third method on the ML dataset) was made up for by a reduction in test error rate. We also implemented weak learners that restricted their hypotheses to have positive cumulative weights in order to test whether such hypotheses were helpful or harmful in reducing test error (as discussed at the end of Section 4.3). We called these WeakLearn.2.cum and WeakLearn.3.cum.

To measure the accuracy of a weak learner on a given dataset, after each round of boosting

ML Domain	Top 1	Top 2	Top 5	Top 10	Top 20	Top 30	Avg Rank
RankBoost	102	144	173	184	194	202	4.38
Best (Top 1)	**117**	137	154	167	177	181	6.80
Best (Top 10)	112	**147**	**172**	**179**	185	187	**5.33**
Best (Top 30)	95	129	159	178	**187**	**191**	5.68
University Domain							
RankBoost	95	141	197	215	247	263	7.74
Best single query	112	144	198	221	238	247	8.17

Table 5.1: Comparison of the combined hypothesis and individual search templates.

we plotted the train and test error of the combined hypothesis generated thus far. We ran each weak learner for 1000 rounds of boosting on each of the four partitions of the data and averaged the results. Figure 5-1 displays the plots of train error (left) and test error (right) for the first 200 rounds of boosting on the ML dataset. (The slopes of the curves did not change during the remaining 800 rounds.) The plots for the UNIV dataset were similar.

WeakLearn.2 achieved the lowest train error, followed by WeakLearn.3, and finally WeakLearn.2.cum and WeakLearn.3.cum, whose performance was nearly identical. However, WeakLearn.2.cum and WeakLearn.3.cum produced the lowest test error (again behaving nearly identically) and resisted overfitting, unlike their counterparts. So we see that restricting the weak hypotheses to have positive cumulative weights hampers training performance but improves test performance. Also, when we subject the hypotheses to this restriction, we see no difference between the second and third methods of setting α. Therefore, in our experiments we used WeakLearn.cum.3.cum, the third method of setting α that allows only positive cumulative hypothesis weights.

Evaluation. In order to determine a good number of boosting rounds, we first ran RankBoost on each partition of the data and produced a graph of the average training error. For the ML data set, the training error did not decrease significantly after 50 rounds of boosting (see Fig. 5-1 (left)), so we used the final hypothesis built after 50 rounds. For the UNIV data set, the training error did not decrease significantly after 40 rounds of boosting (graph omitted), so we used the final hypothesis built after 40 rounds.

To evaluate the performance of the individual search templates in comparison to the combined hypothesis output by RankBoost, we measured the number of queries for which the correct document was in the top k ranked documents, for various values of k. We then compared the performance of the combined hypothesis to that of the best search template for each value of k. The results for the ML and UNIV domains are shown in Table 5.1. All columns except the last give the number of base queries for which the correct homepage received a rank greater than or equal to k. Bold figures give the maximum value over all of the search templates on the test data. Note that the best search template is determined based on its performance on the *test* data, while RankBoost only has access to *training* data.

For the ML data set, the combined hypothesis closely tracked the performance of the best expert at every value of k, which is especially interesting since no single template was the best for all values of k. For the UNIV data set, a single template was the best[3] for all values of k, and the combined hypothesis performed almost as well as the best template for $k = 1, 2, \ldots, 10$ and then outperformed the best template for $k = 20, 30$. Of course, having found a single best template, there is no need to use RankBoost.

We also computed (an approximation to) average rank, i.e., the rank of the correct homepage URL, averaged over all base queries in the test set. For this calculation, we viewed each search template as assigning a rank of 1 through 30 to its returned URL's, rank 1 being the best. Since the correct URL was sometimes not ranked by a search template, we artificially assigned a rank of 31 to every unranked document. For each base query, RankBoost ranked every URL returned by every search template. Thus if the total number of URL's was larger than 30, RankBoost assigned to some instances ranks greater than 30. To avoid an unfair comparison to the search templates, we limited the maximum rank of RankBoost to 31. The last column of Table 5.1 gives average rank.

5.2 Movie recommendations

Our second set of experiments dealt with the movie-recommendation task described in the introduction, the goal of which is to produce for a given user a list of unseen movies ordered by predicted preference. Unlike the meta-search task where the output ordering was evaluated according to relative rank of a single document (the correct homepage), in the movie task the output ordering is compared to the correct ordering given by the user. Thus, the movie task tests RankBoost on a more general ranking problem. However, performance measures for comparing two ranked lists are not as clear cut; we defined four such measures for this purpose. To evaluate the performance of RankBoost, we compared it to a nearest-neighbor algorithm and a regression algorithm.

5.2.1 Description of task and data set

For these experiments we used publicly available data[4] provided by the Digital Equipment Corporation which ran its own EachMovie recommendation service for the eighteen months between March 1996 and September 1997 and collected user preference data. Movie viewers were able to assign a movie a score from the set $R = \{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$, 1.0 being the best. We used the data of 61,625 viewers entering a total of 2,811,983 numeric ratings for 1,628 different movies (films and videos).

Most of the mapping of this problem into our framework was described in Section 4.1. For our experiments, we selected a subset C of the viewers to serve as ranking features: each viewer in C defined an ordering of the set of movies that he or she viewed. The feedback function Φ was then defined as in Section 4.1 using the movie ratings of a single target user. We used half of the movies viewed by the target user for the feedback function in

[3]The best search template for the UNIV domain was `"NAME" PLACE`.

[4]From 'http://www.research.digital.com/SRC/eachmovie/'.

training and used the other half of the viewed movies for testing as described below. We then averaged all results over multiple runs with many different target users (details are given in Section 5.2.5).

5.2.2 Experimental parameters

In the meta-search task we assumed that all search engines had access to all documents and thus the absence of a document on a search engine's list indicated low preference. This assumption does not hold in the movie task as it is not clear what a viewer's preference will be on an unseen movie. Thus we did not set the parameter q_{def}, allowing the weak learner to choose it adaptively. As in the meta-search task, we used the modified definition of ranking loss given in Eq. (5.1). We also used WeakLearn.3.cum because preliminary experiments revealed that this weak learner achieved a lower test error rate than WeakLearn.3 and also resisted overfitting. In these experiments, we ran RankBoost for 100 rounds.

5.2.3 Algorithms for comparison

We compared the performance of RankBoost on this data set to two other algorithms, a regression algorithm and a nearest-neighbor algorithm.

Regression. We used a regression algorithm similar to the ones used by Hill et al. [39]. The algorithm employs the assumption that the scores assigned a movie by a target user Alice can be described as a linear combination of the scores assigned to that movie by other movie viewers. Formally, let $\mathbf{a}$ be a row vector whose components are the scores Alice assigned to movies (discarding unranked movies). Let $\mathbf{C}$ be a matrix containing the scores of the other viewers for the subset of movies that Alice has ranked. Since some of the viewers have not ranked movies that were ranked by Alice, we need to decide on a default rank for these movies. For each viewer represented by a row in $\mathbf{C}$, we set the score of the viewer's unranked movies to be the viewer's average score over all movies. We next use linear regression to find a vector $\mathbf{w}$ of minimum length that minimizes $||\mathbf{w}\,\mathbf{C} - \mathbf{a}||$. This can be done using standard numerical techniques (we used the package available in Matlab). Given $\mathbf{w}$ we can now predict Alice's ratings of all the movies.

Nearest neighbor. Given a target user Alice with certain movie preferences, the nearest-neighbor algorithm (NN) finds a movie viewer Bob whose preferences are most similar to Alice's and then uses Bob's preferences to make recommendations for Alice. More specifically, we find the ranking feature f_i (corresponding to one of the other movie viewers) that gives an ordering most similar to that of the target user as encoded by the feedback function Φ. The measure of similarity we use is the ranking loss of f_i with respect to the same initial distribution D that was constructed by RankBoost. Thus, in some sense, NN can be viewed as a single weak hypothesis output after one round of RankBoost (although no threshold of f_i is performed).

As with regression, a problem with NN is that the neighbor it selects may not rank all the movies ranked by the target user. To fix this, we modified the algorithm to associate with each feature f_i a default score $q_{\text{def}} \in R$ which f_i assigns to unranked movies. When searching for the best feature, NN chooses q_{def} by calculating and then minimizing the ranking loss (on the training set) for each possible value of q_{def}. If it is the case that this viewer ranks all of the (training) movies seen by the target user, then NN sets q_{def} to the average score over all movies that it ranked (including those not ranked by the target user).

5.2.4 Performance measures

In order to evaluate and compare performance, we used four error measures, disagreement, predicted-rank-of-top, coverage, and average precision. Disagreement compares the entire predicted order to the entire correct order, whereas the other three measures are concerned only with the predicted rank of those instances that should have received the top rank.

We assume that each of the algorithms described in the previous section produces a real-valued function H that orders movies in the usual way: x_1 ranked higher than x_0 if $H(x_1) > H(x_0)$. The correct ordering of test movies, c, is also represented as a real-valued function.

For each of the following measures, we first give the definition when H is a total order, meaning it assigns a unique score to each movie. When H is a partial order, as is the case for some of the algorithms, we assume that ties are broken randomly when producing a list of movies ordered by H. In this situation we calculate the expectation of the error measure over the random choices to break the ties.

Disagreement. Disagreement is the fraction of distinct pairs of movies (in the test set) that H misorders with respect to c. If N is the number of distinct pairs of movies ordered by c, then the disagreement d is

$$\text{disagreement} = \frac{1}{N} \sum_{x_0,x_1:\ c(x_0)<c(x_1)} [\![H(x_0) > H(x_1)]\!] \ .$$

This is equivalent to the ranking loss of H (Eq. (4.1)) where c is used to construct the feedback function. If H is a partial order, then its expected disagreement with respect to c is

$$\mathrm{E}\,[\text{disagreement}] = \frac{1}{N} \sum_{x_0,x_1:\ c(x_0)<c(x_1)} \Big([\![H(x_0) > H(x_1)]\!] + \tfrac{1}{2}\,[\![H(x_0) = H(x_1)]\!]\Big) \ .$$

This is equivalent to Eq. (5.1) where c is used to construct the feedback function.

Precision/recall measures Disagreement is one way of comparing two orderings, and it is the function that both RankBoost and NN attempt to minimize. We should consider evaluating the rankings of these algorithms using other measures as well, for a number of reasons. One reason is to test whether RankBoost's minimization of ranking loss produces rankings that have high quality with respect to other measures. This can be evaluated also by looking at the comparative performance on another measure of RankBoost and regression, since the latter doesn't directly minimize disagreement. Another reason is motivated by the application: people looking for movie recommendations will likely be more interested in the top of the predicted ranking than the bottom. That is, they will want to know what movies to go and see, not what movies to avoid at all costs.

For these reasons we considered three other error measures, which view the movie recommendation task as having bipartite feedback. According to these measures, the goal of the movie task is find movies that Alice will love. Thus any set of movies that she has seen is partitioned in two: those which she assigned her highest score and those which she assigned a lesser score. This is an example of a ranked-retrieval task in the field of information retrieval, where only the movies that Alice assigns her highest score are considered relevant. As discussed in Section 4.2.3, the goal here is not to classify but to rank.

We refer to the movies to which Alice assigns her highest score as *good movies*. We based our error measures on the precision measures used for that task. The *precision* of the kth good movie appearing in a ranked list is defined as k divided by the number of movies on the list up to and including this movie. For example, if all the good movies appear one after another at the top of a list, then the precision of every good movie is 1.

More formally, define $\text{rank}(m)$, the rank of movie m appearing in the list ordered by H, as the position of m in the list, e.g. first=1, second=2, etc. Suppose there are K good movies (according to Alice), and denote their sequence on H's list as $\{t_k\}_{k=1}^{K}$. In other words, $H(t_1) \geq \cdots \geq H(t_K)$. Then the precision of the first good movie is $1/\text{rank}(t_1)$, and, more generally, the precision of the kth good movie is $k/\text{rank}(t_k)$. Again, if all K good movies appear one after another at the top of H's list, meaning $\text{rank}(t_k) = k$ for every k, then the precision of every good movie is 1.

Average Precision (AP). Average precision, commonly used in the information retrieval community, measures how good H is at putting good movies high on its list. It is defined as

$$\text{AP} = \frac{1}{K} \sum_{k=1}^{K} \frac{k}{\text{rank}(t_k)} .$$

If H is a partial order, then t_k is a random variable, and therefore so is $\text{rank}(t_k)$, and we calculate expected average precision. Let N be the total number of movies ranked by H. Then,

$$\text{E}\left[\text{AP}\right] = \frac{1}{K} \sum_{k=1}^{K} k \sum_{i=k}^{N-K+k} \Pr\left[\text{rank}(t_k) = i\right] \frac{1}{i} .$$

The formula for $\Pr\left[\text{rank}(t_k) = i\right]$ is a ratio with binomial coefficients in the numerator and denominator, and we defer its statement and derivation to Appendix 5.2.7.

Predicted-rank-of-top (PROT). PROT is the precision of the first good movie on H's list and measures how good H is at ranking one good movie high on its list. It is

$$\text{PROT} = \frac{1}{\text{rank}(t_1)} .$$

If H is a partial order, its expected PROT is

$$\text{E}\left[\text{PROT}\right] = \sum_{i=1}^{N-K+1} \Pr\left[\text{rank}(t_1) = i\right] \frac{1}{i} .$$

Coverage. Coverage is the precision of the last good movie on H's list (also known as precision at recall 1), and it measures how good H is at ranking its lowest good movie. It is

$$\text{coverage} = \frac{1}{\text{rank}(t_K)} \ .$$

If H is a partial order, its expected coverage is

$$\mathrm{E}\left[\text{coverage}\right] = \sum_{i=K}^{N} \Pr\left[\text{rank}(t_K) = i\right] \frac{K}{i} \ .$$

5.2.5 Experimental results

We now describe our experimental results. We ran a series of three tests, examining the performance of the algorithms as we varied the number of features, the *density* of the features, meaning the number of movies ranked by each movie viewer, and the density of the feedback, meaning the number of movies ranked by each target user.

We first experimented with the number of features used for ranking. We selected two disjoint random sets T and T' of 2000 viewers each. Subsets of the viewers in T were used as feature sets, and each of the users in T' was used as feedback. Specifically, we further broke T down into six subsets of sizes 100, 200, 500, 750, 1000, and 2000. Each subset served as a feature set for training on half of a target user's movies and testing on the other half, for each user in T'. For each algorithm, we calculated the four measures described above, averaged over the 2000 target users. We ran the algorithms on five disjoint random splits of the data into feature and feedback sets, and we averaged the results, which are shown in Figure 5-2.

RankBoost was the clear winner for all four performance measures, achieving the lowest disagreement and the highest AP, PROT, and coverage. Also, the slopes of the curves indicated that RankBoost was best able to improve its performance as the number of features increased.

NN did well on disagreement, AP, and coverage, but on PROT it performed worse than random guessing. This suggests that, although NN places good movies relatively high in its list (because of its good AP), it does not place a single good movie near the top of its list (because of its poor PROT). An investigation of the data revealed that almost always the nearest neighbor did not view all of the movies in the test feedback and therefore NN assigned some movies a default score (as described in Section 5.2.3). Sometimes the default score was high and placed the unseen movies at the top of NN's list, which can drive down the PROT if most of the unseen movies are not good movies (according to the feedback).

RankBoost and NN directly tried to minimize disagreement whereas regression did not, and its disagreement was little better than that of random guessing. Regression did perform better than random guessing on PROT and coverage, but on average precision it was worse. This suggests that most of the good movies appear low on regression's list even though the first good movie appears near the top. Also, judging by the slopes of its performance curves, regression did not make much use of the additional information provided by a larger number of features. We discuss possible reasons for this poor performance at the end of this section.

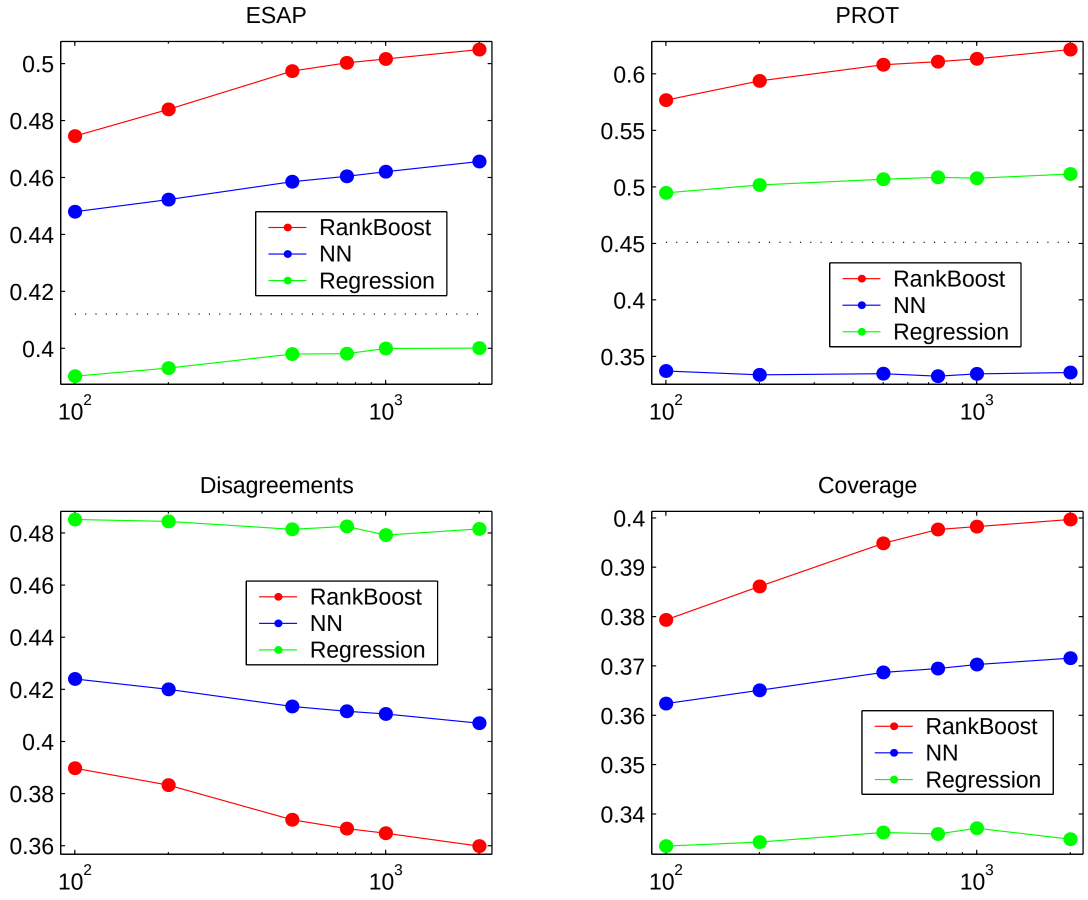

Figure 5-2: Performance of algorithms with respect to feature sets of sizes 100, 200, 500, 750, 1000, 2000. For PROT and ESAP, the dashed line shows the (expected) performance of a random permutation of the movies. For disagreement, the expected performance of a random ordering was 0.5. For coverage, the expected performance was 0.32.
For disagreement, the expected performance of a random ordering was 0.5.

In our next experiment we explored the effect of the density of the features, the number number of movies ranked by each viewer. We partitioned the set of features into bins according to their density. The bins were 10-20, 21-40, 41-60, 61-100, 101-1455, where 1455 was the maximum number of movies ranked by a single viewer in the data set. We selected a random set of 1000 features (viewers) from each bin to be evaluated on a disjoint random set of 1000 feedback target users (of varying densities). We ran the algorithms on six such random splits, calculated the averages of the four error measures on each split, and then averaged them together. The results are shown in Figure 5-3. The x-coordinate of each point is the average density of the features in a single bin; for example, 80 is the average density of features whose density is in the range 61-100.

The relative performance of the algorithms was the same as in Figure 5-2. RankBoost was the winner again, and it was best able to improve its performance when presented with the additional information provided by the denser features. As feature density increased,

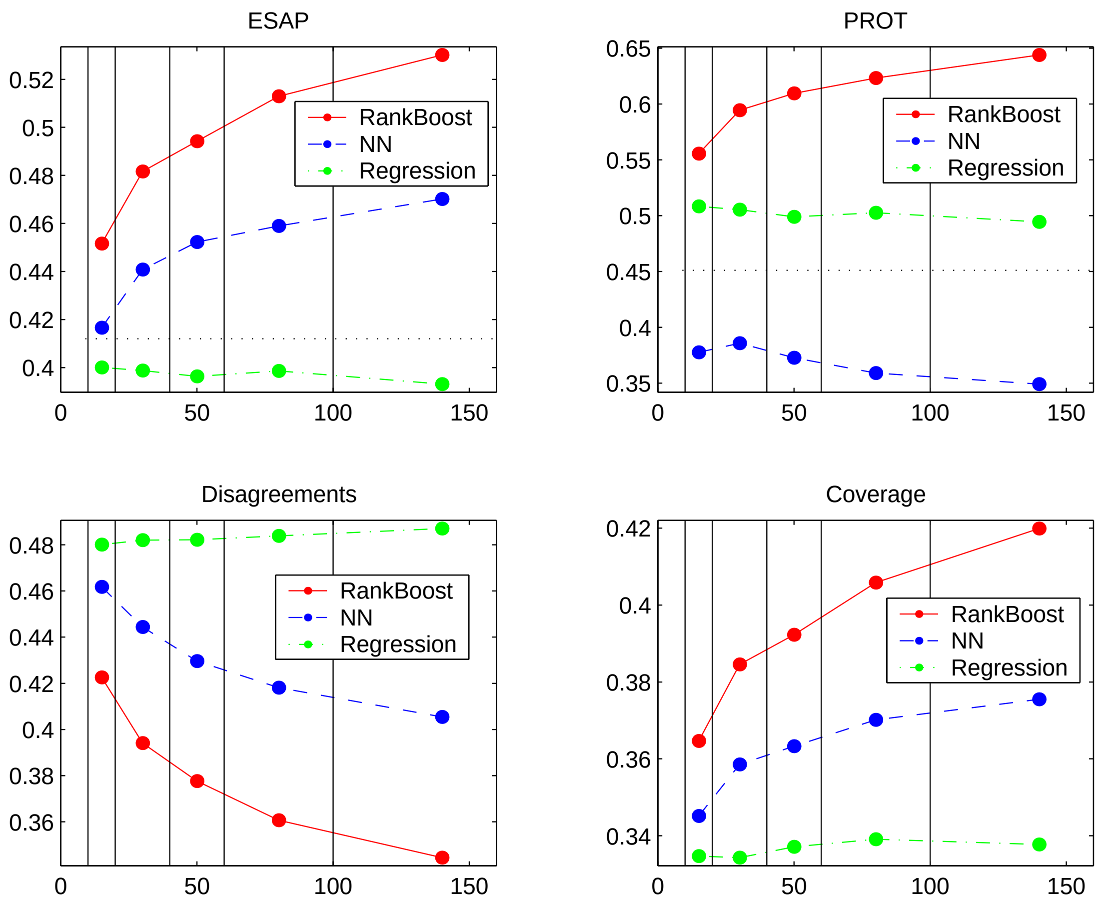

Figure 5-3: Performance of the algorithms on different feature densities. For PROT and ESAP, the dashed line shows the (expected) performance of a random permutation of the movies. For disagreement, the expected performance of a random ordering was 0.5. For coverage, the expected performance was 0.32.

NN's performance on AP, disagreement, and coverage improved more significantly than when simply the number of features increased (Figure 5-2). However, NN continued to perform worse than random guessing on PROT, and its performance degraded as feature density increased. As for regression, it continued to perform similarly to or worse than random guessing, and its performance was largely unaffected as feature density increased.

The previous two experiments varied the amount of information provided by the features; in the next experiment, we varied the amount of information provided by the feedback. We varied the feedback density, the number of movies ranked by the target user. We partitioned the users into bins according to density in the same way as in the previous experiment. We ran the algorithms on 1000 target users of each density, using half of the movies ranked by each user for training and the other half for testing. We used a fixed randomly chosen set of 1000 features. We repeated this experiment on six random splits of the data and averaged the results, which appear in Figure 5-4.

The most noticeable effect of increasing the feedback density is that it *degrades* the perfor-

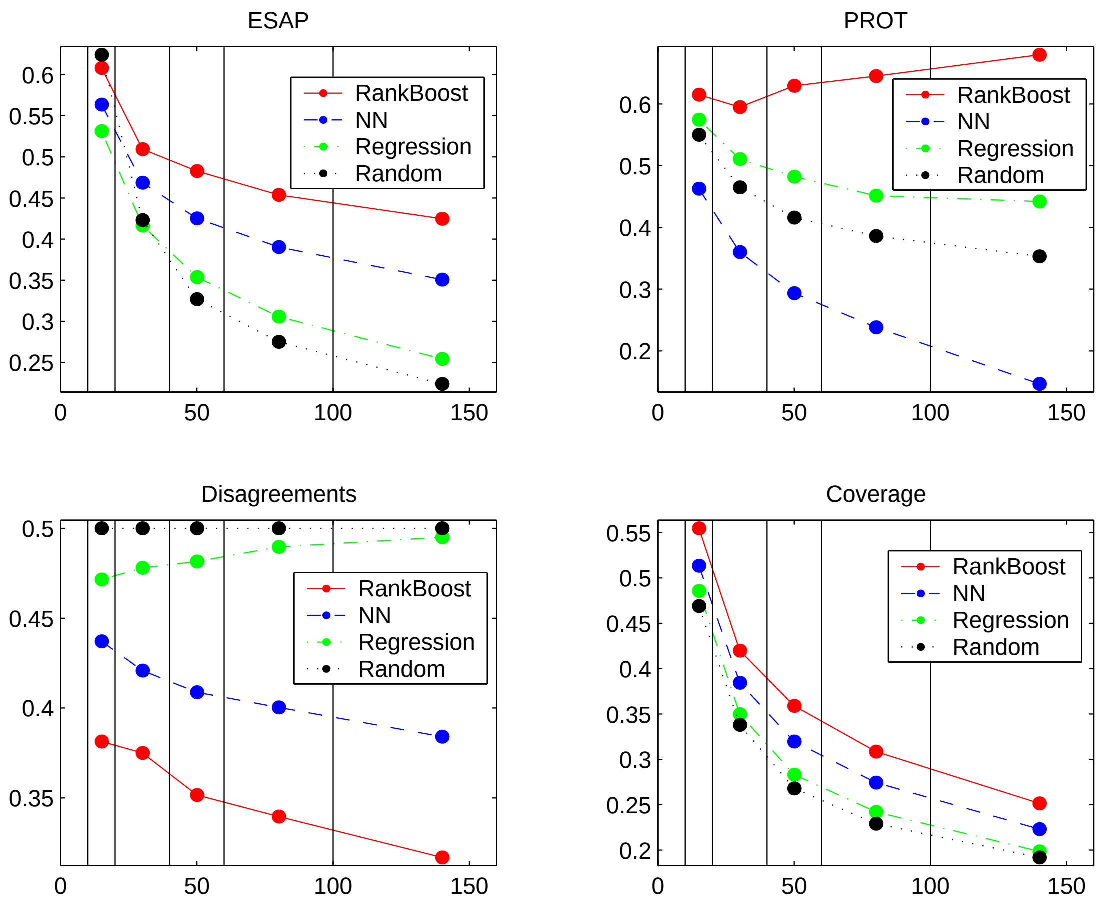

Figure 5-4: Performance of algorithms, including randomly ordering the movies, on different feedback densities.

mance of all three algorithms on AP and coverage and the performance of NN and regression on PROT (RankBoost is able to improve PROT). Other than that, the comparative performance of the algorithms to one another was the same as in the previous experiments, with two exceptions. First, the AP of the random ordering was higher than that of all of the algorithms when the target user ranked 10 to 20 movies. This advantage of the random ordering disappeared when the target user ranked more than 20 movies, which suggests that the algorithms need to train on a ranking of at least 10 movies in order to beat random guessing. The other difference in behavior was that regression finally performed better than random guessing on AP.

At first, it appears counterintuitive that the algorithms should perform worse as the number of movies ranked by the target user increases. One would expect that the algorithms would do better with more training feedback. Indeed this is the case for the disagreement measure (with the exception of regression, as in the previous experiments). This might suggest a weakness of the precision-based measures: that they are sensitive to the number of movies in the feedback. On the other hand, randomly ordering the movies also degrades as the feedback density increases, which suggests that the ranking problem is intrinsically

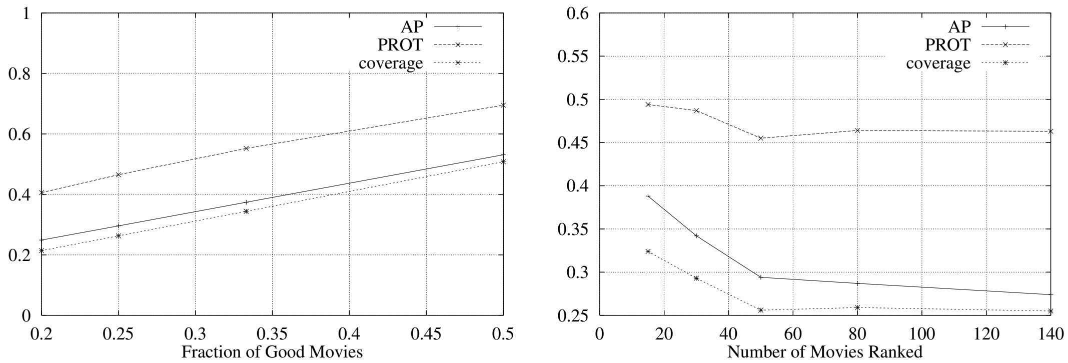

Figure 5-5: Left: The performance of random guessing when the fraction of good movies is varied (the number of movies is 60). Right: The performance of random guessing when the number of movies ranked is varied (the fraction of good movies is $\frac{1}{4}$).

more difficult. This would certainly be the case if the *fraction* of good movies in the feedback decreases as feedback density increases. We discovered that both effects occur.

We first calculated the fraction of good movies in the feedback for each feedback density. For densities of 10-20, 21-40, 41-60, 61-100, and over 100, the fractions were, respectively, 0.33, 0.26, 0.22, 0.20, 0.18. As this fraction decreases, the ranking problem becomes more difficult. For example, the left plot of Figure 5-5 shows the performance of random guessing, with respect to the three measures, as the fraction of good movies varies as $\frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}$ (the number of movies ranked was 60).

However, consider the right plot of Figure 5-5, which shows the performance of random guessing when the fraction of good movies is constant (0.25) and the number of movies ranked is varied. Here to the measured performance degrades, which is an effect of the measures, not the difficulty of the problem. That is, this is the effect of taking a training set of data and making a (fixed) number of copies of each (movie,score) pair, which provides no additional information or challenge to the algorithms described in Section 5.2.3. This sensitivity to the number of movies ranked is a weakness of these three precision-based measures, since ideally we would like them to remain constant for problems of the same difficulty.

5.2.6 Discussion

Our experiments show that RankBoost clearly performed better than regression and nearest neighbor on the movie recommendation task.

RankBoost's approach of ordering based on relative comparisons performed much better than regression which treats the movie scores as absolute numerical values. One reason for regression's poor performance may be overfitting: its solution is subject only to a mild restriction (shortest length, as described in Section 5.2.3). Even so, it is not clear whether this improvement of RankBoost over regression is due to using relative preferences or to boosting or both. To try to separate these effects, we could test regression on relative preferences by normalizing the scores of each movie viewer so that the distribution of scores used by that viewer has the same mean and variance as the distribution of scores of every other viewer.

RankBoost also performed better than the nearest-neighbor algorithm presented here. Based on these experiments we could design a better nearest-neighbor algorithm, choosing default ranks in a better way and, when choosing a nearest neighbor, perhaps taking into account the number of movies ranked by the neighbor. It would also be worthwhile to compare RankBoost to an algorithm which finds k nearest neighbors to a target user and averages their predictions. Such an experiment would differentiate between a straightforward search for and combination of similar users and boosting's method of search and combination. Averaging the prediction of the k nearest neighbors introduces a dependence on absolute scores, however, so this proposed experiment would further test our hypothesis that relative preferences are more informative.

5.2.7 Performance measures for the movie task

For the movie recommendation task, we provided various measures of the performance of a predicted ordering H of movies output by a ranking algorithm (Section 5.2.4). We assumed that if there were ties between movies, meaning that H is a partial order, the ties would be broken randomly when listing one item over an other. To analyze this performance, we calculated the expectation over all ways to break ties, that is, over all total orders that are consistent with H. This expectation involved the quantity $\Pr[\mathrm{rank}(t_k) = i]$, the probability that the kth good movie occurs at position i in H's list, taken over all total orders consistent with H. Here we calculate this probability.

Let R be the number of movies that definitely appear before t_k on H's list,

$$R = |\{m \ : \ H(m) > H(t_k)\}| \ .$$

Let r be the set of all good movies definitely appearing before t_k,

$$r = |\{t \in \{t_1, \ldots, t_{k-1}\} \ : \ H(t) > H(t_k)\}| \ .$$

Let Q be the number of movies tied with t_k,

$$Q = |\{m \ : \ H(m) = H(t_k)\}| \ .$$

Let q be the number of good movies tied with t_k,

$$q = |\{t \in \{t_1, \ldots, t_K\} \ : \ H(t) = H(t_k)\}| \ .$$

Finally, let $j = r - k$, meaning t_k is the jth good movie among the set of tied good movies. Then,

$$\Pr[\mathrm{rank}(t_k) = i] = \frac{\binom{i-R-1}{j-1}\binom{Q-i+R}{q-j}}{\binom{Q}{q}} \ . \tag{5.2}$$

We prove (5.2) as follows. Define the random variable Y_j to be the rank of t_k within the set of tied movies. For example, if t_k is the first movie listed then $Y_j = 1$. Then

$$\Pr[\mathrm{rank}(t_k) = i] = \Pr[R + Y_j = i] = \Pr[Y_j = \ell] \ , \tag{5.3}$$

where $\ell = i - R$. So now we need to calculate the probability that, in a group of equally scored movies, the jth good movie appears at position ℓ.

This process can be modeled as sampling without replacement Q times from an urn with Q balls, q colored green and $Q - q$ colored red. The event $Y_j = \ell$ means that the jth green ball was drawn on the ℓth draw. Looking at the entire sequence of draws, this means that $j - 1$ green balls came up during draws $1, \ldots, \ell - 1$, the jth green ball was drawn on draw ℓ, and $q - j$ green balls came up during draws $\ell + 1, \ldots, Q$. There are $\binom{\ell-1}{j-1}$ ways to arrange the drawings of the first $j - 1$ green balls and $\binom{Q-\ell}{q-j}$ ways to arrange the drawings of the remaining $q - j$ green balls. The total number of all possible sequences of draws is $\binom{Q}{q}$. Thus

$$\Pr\left[Y_j = \ell\right] = \frac{\binom{\ell-1}{j-1}\binom{Q-\ell}{q-j}}{\binom{Q}{q}}. \tag{5.4}$$

Substituting $\ell = i - R$ from (5.3) into this equation gives (5.2), the desired result. ■

Chapter 6

Conclusion

6.1 Summary

The problem of combining preferences arises in several applications, including combining the results of different search engines and collaborative-filtering tasks such as making movie recommendations. One important property of these tasks is that the most relevant information to be combined represents *relative preferences* rather than *absolute ratings*. We have given both a formal framework and an efficient algorithm for the problem of combining preferences, which our experiments indicate works well in practice.

Comparison to Cohen, Schapire, Singer. Our model of the ranking problem is similar to the one proposed by Cohen, Schapire, and Singer [15]. They consider ranking features of the form $f_i : X \to S \cup \{\perp\}$, where S is a totally ordered set and $\perp \notin S$ is incomparable to all elements in S and indicates that no ranking is given. They combine the ranking features using a two step approach. In the first step, they use each f_i to construct a function $R_{f_i} : X \times X \to [0,1]$ which indicates f_i's preference of one instance over another. Given the set $\{R_{f_i}\}$ as input, a learning algorithm in their framework must output a final hypothesis PREF: $X \times X \to [0,1]$ which is a weighted combination $\sum_i w_i R_{f_i},\ w_i \in [0,1]$. In the second step, they are faced with the problem of constructing a total order ρ that has minimum disagreement with PREF (disagreement is similar to ranking loss as defined in Eq. (4.1)). They prove that this problem, which we call MIN-DISAGREE, is NP-complete, and they give a 2-approximation algorithm for it.

In our framework, the ranking features are of the form $f_i : X \to \mathbb{R} \cup \{\perp\}$. This allows us to combine the output of the features but does not change the complexity of MIN-DISAGREE. The real difference between our approach and theirs is that, rather than attempting to minimize disagreement directly, we minimize a function that approximates disagreement, namely the training error bound proven in Theorem 1 of Section 4.2. This function can be minimized efficiently as we have shown by providing fast algorithms for doing so.

Efficiency of algorithms. Our learning system consists of two algorithms: the boosting algorithm RankBoost and the weak learner. The input to the system includes an instance space X, n ranking features, and a feedback function of size $|\Phi|$ that ranks a subset of the instances $X_\Phi \subseteq X$. Given this input, RankBoost generally runs in $O(|\Phi|)$ time, and a naive implementation of the weak learner we present runs in $O(n|\Phi|X_\Phi)$ time. We have shown two improvements in efficiency, as summarized in Theorem 2 of Section 4.3. If we use binary weak hypotheses and we search for the best weak hypothesis using the third method in Section 4.2.2, then we can implement the weak learner in time $O(n|X_\Phi| + |\Phi|)$. If we use a binary feedback function, then we can implement RankBoost in time linear in the number of instances in the feedback. If in addition we use binary weak hypotheses, we can implement the weak learner in $O(n|X_\Phi|)$ time.

These two restriction are both natural and useful. Binary hypotheses are quite simple and this makes them easy to design, analyze, and compute efficiently. Although a single such hypothesis may have only weak predictive power, many of them can be combined via boosting into a highly accurate prediction rule, as is indicated by our experiments. As for restricting the feedback function to be binary, this often does not reduce the applicability of the algorithm, since many applications come with binary feedback, such as those in information retrieval.

Experimental results. In our experiments we used the weak learner that outputs a thresholded ranking feature as weak hypothesis. Although these prediction rules have limited power, RankBoost was nevertheless able to combine them into a highly accurate prediction rule. In the meta-search task, RankBoost performed just as well as the best search strategy for each error measure. In the movie-recommendation task, RankBoost consistently outperformed a standard regression algorithm and a nearest-neighbor algorithm.

Our experiments also indicate that RankBoost is able to do well on data sets of varying sizes. The meta-search task had a small number of ranking features (16 to 22), a large instance space (10,000 URL's) and large feedback (10,000 URL's). The movie task had a large number of ranking features (100 to 2000), a smaller instance space (1,628 movies), and a range of feedback sizes (10-1455).

6.2 Future work

The are a variety of directions for future work. We contend that relative preferences can be more important than absolute scores. The results of our experiments on the movie recommendation task support this: RankBoost significantly outperformed nearest neighbor and regression. to further differentiate between scores and ranks, we proposed two experiments (Section 5.2.6): testing regression on relative preferences by normalizing the scores of each movie viewer, and testing the averaged combination of k nearest neighbors.

As we have pointed out before, many ranking problems have bipartite feedback and therefore can also be viewed as binary classification problems. For such problems it would be interesting to compare RankBoost to AdaBoost combined with a weak leaner for minimizing classification error. AdaBoost outputs a real-valued score for each instance which is then thresholded to produce a classification (see Section 3.2.4). We could compare RankBoost's ordering to AdaBoost's ordering the instances by classification weight to see if minimizing ranking loss is superior to minimizing classification error. There is some anecdotal evidence that this is the case [59], but a thorough empirical evaluation is needed.

As for the RankBoost algorithm itself, the first method for setting α_t is the most general and requires numerical search. Schapire and Singer [65] suggest using general iterative methods such as Newton-Raphson. Because such methods often have no proof of convergence or can be numerically unstable, we would like to find a special purpose iterative method with a proof of convergence. Of course, to be practical, the method would also need to converge quickly.

Perhaps the most important practical research direction is to apply RankBoost to information retrieval (IR) problems, including text, speech, and image retrieval. These IR problems are important today due to the vast amount of data available to people via the WWW and large scale databases, and they are receiving attention from a variety of scientific communities. We would like to test RankBoost's alternative approach of combining preferences by minimize misorderings to see how its performance compares to traditional methods.

In IR experiments where the goal is to order documents by relevance to a query, a ranking feature might be a word or phrase. The feature assigns a document a score which is a function of the document length and the number of times the word or phrase appeared in the document. In this setting the numerical ratings given by the all the features have the same range and meaning. Thus it makes sense to consider combining the features using their actual numeric scores. If we conduct experiments comparing a weak learner that uses the features directly as real-valued functions to the weak learner presented here which thresholds the features, our results can tell us how RankBoost performs with absolute versus relative ratings.

Bibliography

[1] AltaVista: The most powerful and useful guide to the Net. `http://www.altavista.com/`.

[2] Dogpile. `http://www.dogpile.com/`.

[3] MetaCrawler. `http://www.metacrawler.com/`.

[4] Movie Critic. `http://www.moviecritic.com/`.

[5] MovieFinder. `http://www.moviefinder.com/`.

[6] Brian T. Bartell, Garrison W. Cottrell, and Richard K. Belew. Automatic combination of multiple ranked retrieval systems. In *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1994.

[7] Eric Bauer and Ron Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine Learning*, to appear.

[8] Eric B. Baum and David Haussler. What size net gives valid generalization? *Neural Computation*, 1(1):151–160, 1989.

[9] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory*, pages 144–152, 1992.

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

[11] Leo Breiman. Arcing the edge. Technical Report 486, Statistics Department, University of California at Berkeley, 1997.

[12] Leo Breiman. Arcing classifiers. *Annals of Statistics*, 26(3):801–849, 1998.

[13] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees.* Wadsworth & Brooks, 1984.

[14] Rich Caruana, Shumeet Baluja, and Tom Mitchell. Using the future to “sort out” the present: Rankprop and multitask learning for medical risk evaluation. In *Advances in Neural Information Processing Systems 8*, pages 959–965, 1996.

[15] William W. Cohen, Robert E. Schapire, and Yoram Singer. Learning to order things. *Journal of Artificial Intelligence Research*, 10:243–270, 1999.

[16] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, September 1995.

[17] Luc Devroye, Lázló Györfi, and Gábor Lugosi. *A Probabilistic Theory of Pattern Recognition*. Springer, 1996.

[18] Thomas G. Dietterich. An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. Unpublished manuscript, 1998.

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

[20] Harris Drucker and Corinna Cortes. Boosting decision trees. In *Advances in Neural Information Processing Systems 8*, pages 479–485, 1996.

[21] Harris Drucker, Corinna Cortes, L. D. Jackel, Yann LeCun, and Vladimir Vapnik. Boosting and other ensemble methods. *Neural Computation*, 6(6):1289–1301, 1994.

[22] Harris Drucker, Robert Schapire, and Patrice Simard. Boosting performance in neural networks. *International Journal of Pattern Recognition and Artificial Intelligence*, 7(4):705–719, 1993.

[23] Richard O. Duda and Peter E. Hart. *Pattern Classification and Scene Analysis*. Wiley, 1973.

[24] O. Etzioni, S. Hanks, T. Jiang, R. M. Karp, O. Madani, and O. Waarts. Efficient information gathering on the internet. In *37th Annual Symposium on Foundations of Computer Science*, 1996.

[25] Yoav Freund. *Data Filtering and Distribution Modeling Algorithms for Machine Learning*. PhD thesis, University of California at Santa Cruz, 1993. Retrievable from: ftp.cse.ucsc.edu/pub/tr/ucsc-crl-93-37.ps.Z.

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

[27] Yoav Freund. An adaptive version of the boost by majority algorithm. In *Proceedings of the Twelfth Annual Conference on Computational Learning Theory*, pages 102–113, 1999.

[28] Yoav Freund and Robert Schapire. Discussion of the paper "Arcing Classifiers" by leo breiman. *Annals of Statistics*, 26(3):824–832, 1998.

[29] Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm. In *Machine Learning: Proceedings of the Thirteenth International Conference*, pages 148–156, 1996.

[30] Yoav Freund and Robert E. Schapire. Game theory, on-line prediction and boosting. In *Proceedings of the Ninth Annual Conference on Computational Learning Theory*, pages 325–332, 1996.

[31] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, August 1997.

[32] Yoav Freund and Robert E. Schapire. Adaptive game playing using multiplicative weights. *Games and Economic Behavior*, (to appear).

[33] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting. Technical Report, 1998.

[34] Drew Fudenberg and Jean Tirole. *Game Theory*. MIT Press, 1991.

[35] Adam J. Grove and Dale Schuurmans. Boosting in the limit: Maximizing the margin of learned ensembles. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 1998.

[36] Venkatesan Guruswami and Amit Sahai. Multiclass learning, boosting, and error-correcting codes. In *Proceedings of the Twelfth Annual Conference on Computational Learning Theory*, pages 145–155, 1999.

[37] L. Guttman. What is not what in statistics. *The Statistician*, 26:81–107, 1978.

[38] David Haussler. Decision theoretic generalizations of the PAC model for neural net and other learning applications. *Information and Computation*, 100(1):78–150, 1992.

[39] Will Hill, Larry Stead, Mark Rosenstein, and George Furnas. Recommending and evaluating choices in a virtual community of use. In *Human Factors in Computing Systems CHI'95 Conference Proceedings*, pages 194–201, 1995.

[40] W. Iba and P. Langley. Polynomial learnability of probabilistic concepts with respect to the Kullback-Liebler divergence. In *Machine Learning: Proceedings of the Ninth International Conference*, pages 233–240, 1992.

[41] Jeffrey C. Jackson and Mark W. Craven. Learning sparse perceptrons. In *Advances in Neural Information Processing Systems 8*, pages 654–660, 1996.

[42] P.B. Kantor. Decision level data fusion for routing of documents in the TREC3 context: a best case analysis of worst case results. In *Proceedings of the third text retrieval conference (TREC-3)*, 1994.

[43] Michael Kearns and Leslie G. Valiant. Cryptographic limitations on learning Boolean formulae and finite automata. *Journal of the Association for Computing Machinery*, 41(1):67–95, January 1994.

[44] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[45] Ron Kohavi and David H. Wolpert. Bias plus variance decomposition for zero-one loss functions. In *Machine Learning: Proceedings of the Thirteenth International Conference*, pages 275–283, 1996.

[46] Eun Bae Kong and Thomas G. Dietterich. Error-correcting output coding corrects bias and variance. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 313–321, 1995.

[47] Nick Littlestone. Learning when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2:285–318, 1988.

[48] Nick Littlestone and Manfred Warmuth. The weighted majority algorithm. In *30th Annual Symposium on Foundations of Computer Science*, pages 256–261, October 1989.

[49] Richard Maclin and David Opitz. An empirical evaluation of bagging and boosting. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 546–551, 1997.

[50] Dragos D. Margineantu and Thomas G. Dietterich. Pruning adaptive boosting. In *Machine Learning: Proceedings of the Fourteenth International Conference*, pages 211–218, 1997.

[51] Llew Mason, Peter Bartlett, and Jonathan Baxter. Direct optimization of margins improves generalization in combined classifiers. Technical report, Deparment of Systems Engineering, Australian National University, 1998.

[52] C. J. Merz and P. M. Murphy. UCI repository of machine learning databases, 1998. http://www.ics.uci.edu/∼mlearn/MLRepository.html.

[53] J. R. Quinlan. Bagging, boosting, and C4.5. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 725–730, 1996.

[54] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[55] Lawrence Rabiner and Bing-Hwang Juang. *Fundamentals of Speech Recognition*. Prentice Hall, 1993.

[56] Paul Resnick, Neophytos Iacovou, Mitesh Sushak, Peter Bergstrom, and John Riedl. Grouplens: An open architecture for collaborative filtering of netnews. In *Proceedings of Computer Supported Cooperative Work*, 1995.

[57] Gerard Salton. *Automatic text processing: the transformation, analysis and retrieval of information by computer*. Addison-Wesley, 1989.

[58] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[59] Robert E. Schapire. Personal communication.

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

[61] Robert E. Schapire. *The Design and Analysis of Efficient Learning Algorithms*. MIT Press, 1992.

[62] Robert E. Schapire. Using output codes to boost multiclass learning problems. In *Machine Learning: Proceedings of the Fourteenth International Conference*, pages 313–321, 1997.

[63] Robert E. Schapire. Theoretical views of boosting. In *Computational Learning Theory: Fourth European Conference, EuroCOLT '99*, pages 1–10, 1999.

[64] Robert E. Schapire, Yoav Freund, Peter Bartlett, and Wee Sun Lee. Boosting the margin: A new explanation for the effectiveness of voting methods. *Annals of Statistics*, 26(5):1651–1686, 1998.

[65] Robert E. Schapire and Yoram Singer. Improved boosting algorithms using confidence-rated predictions. In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, pages 80–91, 1998.

[66] Holger Schwenk and Yoshua Bengio. Training methods for adaptive boosting of neural networks. In *Advances in Neural Information Processing Systems 10*, pages 647–653, 1998.

[67] Upendra Shardanand and Pattie Maes. Social information filtering: Algorithms for automating "word of mouth". In *Human Factors in Computing Systems CHI'95 Conference Proceedings*, 1995.

[68] Robert Tibshirani. Bias, variance and prediction error for classification rules. Technical report, University of Toronto, November 1996.

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

[70] L. G. Valiant. Learning is computational (Knuth Prize Lecture). In *38th Annual Symposium on Foundations of Computer Science*, October 1997.

[71] V. N. Vapnik. *Estimation of Dependences Based on Empirical Data*. Springer-Verlag, 1982.

[72] V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, 1995.

[73] V. G. Vovk. A game of prediction with expert advice. *Journal of Computer and System Sciences*, 56(2):153–173, April 1998.