Journal of Machine Learning Research 4 (2003) 933-969 Submitted 12/01; Revised 11/02; Published 11/03

An Efficient Boosting Algorithm for Combining Preferences

Yoav Freund
Center for Computational Learning Systems
Columbia University
500 West 120th St.
New York, NY 10027 FREUND@CS.COLUMBIA.EDU

Raj Iyer
Living Wisdom School
456 College Avenue
Palo Alto, CA 94306 DHARMARAJ@LIVINGWISDOMSCHOOL.ORG

Robert E. Schapire
Department of Computer Science
Princeton University
35 Olden Street
Princeton, NJ 08544 SCHAPIRE@CS.PRINCETON.EDU

Yoram Singer
School of Computer Science & Engineering
Hebrew University
Jerusalem 91904, Israel SINGER@CS.HUJI.AC.IL

Editor: Thomas G. Dietterich

Abstract

We study the problem of learning to accurately rank a set of objects by combining a given collection of ranking or preference functions. This problem of combining preferences arises in several applications, such as that of combining the results of different search engines, or the "collaborative-filtering" problem of ranking movies for a user based on the movie rankings provided by other users. In this work, we begin by presenting a formal framework for this general problem. We then describe and analyze an efficient algorithm called RankBoost for combining preferences based on the boosting approach to machine learning. We give theoretical results describing the algorithm's behavior both on the training data, and on new test data not seen during training. We also describe an efficient implementation of the algorithm for a particular restricted but common case. We next discuss two experiments we carried out to assess the performance of RankBoost. In the first experiment, we used the algorithm to combine different web search strategies, each of which is a query expansion for a given domain. The second experiment is a collaborative-filtering task for making movie recommendations.

1. Introduction

Consider the following movie-recommendation task, sometimes called a "collaborative-filtering" problem (Hill et al., 1995, Shardanand and Maes, 1995). In this task, a new user, Alice, seeks recommendations of movies that she is likely to enjoy. A collaborative-filtering system first asks Alice to rank movies that she has already seen. The system then examines the rankings of movies provided by other viewers and uses this information to return to Alice a list of recommended movies. To do that, the recommendation system looks for viewers whose preferences are similar to Alice's and combines their preferences to make its recommendations.

In this paper, we introduce and study an efficient learning algorithm called RankBoost for combining multiple rankings or preferences (we use these terms interchangeably). This algorithm is based on Freund and Schapire's (1997) AdaBoost algorithm and its recent successor developed by Schapire and Singer (1999). Similar to other boosting algorithms, RankBoost works by combining many "weak" 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.

We study the ranking problem in a general learning framework described in detail in Section 2. Roughly speaking, in this framework, the goal of the learning algorithm is simply to produce a single linear ordering of the given set of objects by combining a set of given linear orderings called the *ranking features*. As a form of feedback, the learning algorithm is also provided with information about which pairs of objects should be ranked above or below one another. The learning algorithm then attempts to find a combined ranking that misorders as few pairs as possible, relative to the given feedback.

In Section 3, we describe RankBoost in detail and we prove a theorem about its effectiveness on the training set. We also describe an efficient implementation for "bipartite feedback," a special case that occurs naturally in many domains. We analyze the complexity of all of the algorithms studied.

In Section 4, we describe an efficient procedure for finding the weak rankings that will be combined by RankBoost using the ranking features. For instance, for the movie task, this procedure translates into using very simple weak rankings that partition all movies into only two equivalence sets, those that are more preferred and those that are less preferred. Specifically, we use another viewer's ranked list of movies partitioned according to whether or not he prefers them to a 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. In other words, even if the ranking of movies is expressed by assigning each movie a numeric score, we ignore the numeric values of these scores and concentrate only on their relative order. This distinction becomes very important when we combine the rankings of many viewers who often use completely different ranges of scores to express identical preferences. Situations where we need to combine the rankings of different models also arise in meta-searching problems (Etzioni et al., 1996) and in information-retrieval problems (Salton, 1989, Salton and McGill, 1983).

In Section 5, for a particular probabilistic setting, we study the generalization performance of RankBoost, that is, how we expect it to perform on test data not seen during training. This analysis is based on a uniform-convergence theorem that we prove relating the performance on the training set to the expected performance on a separate test set.

In Section 6, we report the results of experimental tests of our approach on two different problems. The first is the meta-searching problem. In a meta-search application, the goal is to combine the rankings of several web search strategies. Each search strategy is an operation that takes a query as input, performs some simple transformation of the query (such as adding search directives like "AND", or search tokens like "home page") and sends it to a particular search engine. The outcome of using each strategy is an ordered list of URLs 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 second problem is the movie-recommendation problem described above. For this problem, there exists a large publicly available dataset that 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 using several evaluation measures. RankBoost was the clear winner in these experiments.

In addition to the experiments that we report, Collins (2000) and Walker, Rambow, and Rogati (2001) describe recent experiments using the RankBoost algorithm for natural-language processing tasks. Also, in a recent paper (Iyer et al., 2000), two versions of RankBoost were compared to traditional information retrieval approaches.

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 tend to be based either on nearest-neighbor methods (Resnick et al., 1995, Shardanand and Maes, 1995) or gradient-descent techniques (Bartell et al., 1994, Caruana et al., 1996). In the latter case, the rankings are viewed as real-valued scores and the problem of combining different rankings reduces to numerical search for a set of parameters that will minimize the disparity between the combined scores and the feedback of a user.

While the above (and other) approaches might work well in practice, they still do not guarantee that the combined system will match the user's preference when we view the scores as a means to express preferences. Cohen, Schapire and Singer (1999) proposed a framework for manipulating and combining multiple rankings in order to directly minimize the number of disagreements. In their framework, the rankings are used to construct preference graphs and the problem is reduced to a *combinatorial* optimization problem which turns out to be NP-complete; hence, an approximation is used to combine the different rankings. They also describe an efficient *on-line* algorithm for a related problem.

The algorithm we present in this paper uses a similar framework to that of Cohen, Schapire and Singer, but avoids the intractability problems. Furthermore, as opposed to their on-line algorithm, RankBoost is more appropriate for batch settings where there is enough time to find a good combination. Thus, the two approaches complement each other. Together, these algorithms constitute a viable approach to the problem of combining multiple rankings that, as our experiments indicate, works very well in practice.

2. A Formal Framework for the Ranking Problem

In this section, we describe our formal model for studying ranking.

Let $\mathcal{X}$ be a set called the *domain* or *instance space*. Elements of $\mathcal{X}$ are called *instances*. These are the objects that we are interested in ranking. For example, in the movie-ranking task, each movie is an instance.

Our goal is to combine a given set of preferences or rankings of the instance space. We use the term *ranking feature* to denote these given rankings of the instances. A ranking feature is nothing more than an ordering of the instances from most preferred to least preferred. To make the model flexible, we allow ties in this ordering, and we do not require that *all* of the instances be ordered by every ranking feature.

We assume that a learning algorithm in our model is given n ranking features denoted $f_1, \ldots, f_n$. Since each ranking feature f_i defines a linear ordering of the instances, we can equivalently think of f_i as a scoring function where higher scores are assigned to more preferred instances. That is, we can represent any ranking feature as a real-valued function where $f_i(x_1) > f_i(x_0)$ means that instance x_1 is preferred to x_0 by f_i. The actual numerical values of f_i are immaterial; only the ordering that they define is of interest. Note that this representation also permits ties (since f_i can assign equal values to two instances).

As noted above, it is often convenient to permit a ranking feature f_i to "abstain" on a particular instance. To represent such an abstention on a particular instance x, we simply assign $f_i(x)$ the special symbol $\perp$ which is incomparable to all real numbers. Thus, $f_i(x) = \perp$ indicates that no ranking is given to x by f_i. Formally, then, each ranking feature f_i is a function of the form $f_i : \mathcal{X} \to \overline{\mathbb{R}}$, where the set $\overline{\mathbb{R}}$ consists of all real numbers, plus the additional element $\perp$.

Ranking features 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. (However, we choose to call them "ranking features" rather than simply "features" to stress that they have a particular form and function.)

For example, in one formulation of the movie task, each ranking feature corresponds to a single viewer's past ratings of movies, so there are as many ranking features as there are past users of the recommendation service. Movies which were rated by that viewer are assigned the viewer's numerical rating of the movie; movies which were not rated at all by that viewer are assigned the special symbol $\perp$ to indicate that the movie was not ranked. Thus, $f_i(x)$ is movie-viewer i's numerical rating of movie x, or $\perp$ if no rating was provided.

The goal of learning is to combine all of the ranking functions into a single ranking of the instances called the *final* or *combined ranking*. The final ranking should have the same form as that of the ranking features; that is, it should give a linear ordering of the instances (with ties allowed). However, unlike ranking features, we do not permit the final ranking to abstain on any instances, since we want to be able to rank all instances, even those not seen during training. Thus, formally the final ranking can be represented by a function $H : \mathcal{X} \to \mathbb{R}$ with a similar interpretation to that of the ranking features, i.e., x_1 is ranked higher than x_0 by H if $H(x_1) > H(x_0)$. Note the explicit omission of $\perp$ from the range of H, thus prohibiting abstentions. For example, for the movie task, this corresponds to a complete ordering of all movies (with ties allowed), where the most highly recommended movies at the top of the ordering have the highest scores.

Finally, we need to assume that the learner has some feedback information describing the desired form of the final ranking. Note that this information is not encoded by the ranking features, which are merely the primitive elements with which the learner constructs its final ranking. In traditional classification learning, this feedback would take the form of labels on the examples which indicate the correct classification. Here our goal is instead to come up with a good ranking of the instances, so we need some feedback describing, by example, what it means for a ranking to be "good."

One natural way of representing such feedback would be in the same form as that of a ranking feature, i.e., as a linear ordering of all instances (with ties and abstentions allowed). The learner's goal then might be to construct a final ranking which is constructed from the ranking features and which is "similar" (for some appropriate definition of similarity) to the given feedback ranking. This model would be fine, for instance, for the movie ranking task since the target movie-viewer Alice provides ratings of all of the movies she has seen, information that can readily be converted into a feedback ranking in the same way that other users' have their rating information converted into ranking features.

However, in other domains, this form and representation of feedback information may be overly restrictive. For instance, in some cases, two instances may be entirely unrelated and we may not care about how they compare. For example, suppose we are trying to rate individual pieces of fruit. We might only have information about how individual apples compare with other apples, and how oranges compare with oranges; we might not have information comparing apples and oranges. A more realistic example is given by the meta-search task described in Section 2.1.

Another difficulty with restricting the feedback to be a linear ordering is that we may consider it very important (because of the strength of available evidence) to rank instance x_1 above x_0, but only slightly important that instance x_2 be ranked above x_3. Such variations in the importance of how instances are ranking against one another cannot be easily represented using a simple linear ordering of the instances.

To allow for the encoding of such general feedback information, we instead assume that the learner is provided with information about the relative ranking of individual pairs of instances. That is, for every pair of instances x_0, x_1, the learner is informed as to whether x_1 should be ranked above or below x_0, and also how important or how strong is the evidence that this ranking should exist. All of this information can be conveniently represented by a single function Φ. The domain of Φ is all pairs of instances. For any pair of instances x_0, x_1, $\Phi(x_0, x_1)$ is a real number whose sign indicates whether or not x_1 should be ranked above x_0, and whose magnitude represents the importance of this ranking.

Formally, then, we assume the feedback function has the form $\Phi : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. Here, $\Phi(x_0, x_1) > 0$ means that x_1 should be ranked above x_0 while $\Phi(x_0, x_1) < 0$ means the opposite; a value of zero indicates no preference between x_0 and x_1. As noted above, the larger the magnitude $|\Phi(x_0, x_1)|$, the more important it is to rank x_1 above or below x_0. Consistent with this interpretation, we assume that $\Phi(x, x) = 0$ for all $x \in \mathcal{X}$, and that Φ is anti-symmetric in the sense that $\Phi(x_0, x_1) = -\Phi(x_1, x_0)$ for all $x_0, x_1 \in \mathcal{X}$. Note, however, that we do not assume transitivity of the feedback function.[1]

For example, 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 Alice, -1 if the opposite was the case, and 0 if either of the movies was not seen or if they were equally rated.

As suggested above, a learning algorithm typically attempts to find a final ranking that is similar to the given feedback function. There are perhaps many possible ways of measuring such similarity. In this paper, we focus on minimizing the (weighted) number of pairs of instances which are misordered by the final ranking relative to the feedback function. 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 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 $\mathcal{X}$.) Let us define a pair x_0, x_1 to be *crucial* if $\Phi(x_0, x_1) > 0$ so that the pair receives non-zero weight under D.

The learning algorithms that we study attempt to find a final ranking H with a small weighted number of crucial-pair misorderings, a quantity called the *ranking loss* and denoted $\mathrm{rloss}_D(H)$.

1. In fact, we do not even use the property that Φ is anti-symmetric, so this condition also could be dropped. For instance, we might instead formalize Φ to be a nonnegative function in which a positive value for $\Phi(x_0, x_1)$ indicates that x_1 should be ranked higher than x_0, but there is no prohibition against both $\Phi(x_0, x_1)$ and $\Phi(x_1, x_0)$ being positive. This might be helpful when we have contradictory evidence regarding the "true" ranking of x_0 and x_1, and is analogous in classification learning to the same example appearing twice in a single training set with different labels.

Formally, the ranking loss is defined to be

$$\sum_{x_0,x_1} D(x_0,x_1)\,[\![H(x_1) \le H(x_0)]\!] = \Pr_{(x_0,x_1)\sim D}\left[H(x_1) \le H(x_0)\right]. \tag{1}$$

Here and throughout this paper, we use the notation $[\![\pi]\!]$ which is defined to be 1 if predicate π holds and 0 otherwise.

There are many other ways of measuring the quality of a final ranking. Some of these alternative measures are described and used in Section 6.

Of course, the real purpose of learning is to produce a ranking that performs well even on instances 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 is only of value if it includes 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. Some of these issues are addressed in Section 5.

In studying the complexity of our algorithms, it will be helpful to define various sets and quantities which measure the size of the input feedback function. First of all, we generally assume that the support of Φ is finite. Let $\mathcal{X}_\Phi$ denote the set of *feedback instances*, i.e., those instances that occur in the support of Φ:

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

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

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

In some settings, such as the meta-search task described next, 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.)

2.1 Example: Meta-search

To illustrate this framework, we now describe the meta-search problem and how it fits into the general framework. Experiments with this problem are described in Section 6.1.

For this task, we used the data of Cohen, Schapire and Singer (1999). 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,[2] 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 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 URLs that includes the correct homepage at or close to the top.

2. From `http://www.aic.nrl.navy.mil/~aha/research/machine-learning.html`.

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 Shannon Laboratory in Florham Park" 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.[3] 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 URLs 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 is 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 earlier: $\Phi = \sum_q \Phi_q$.

The output of a learning algorithm is some final ranking H. In the meta-search example H is a weighted combination of the different search templates f_i. To apply H, given a test base query q, we first form all of the expanded queries from search templates and send these to the search engine to obtain lists of URLs. We then evaluate H on each pair (q,u), where u is a returned URL, to obtain a predicted ranking of all of the URLs.

3. 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 (1997) AdaBoost algorithm and its successor developed by Schapire and Singer (1999). 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 use boosting to produce a function $H : \mathcal{X} \to \mathbb{R}$ whose induced ordering of $\mathcal{X}$ will approximate the relative orderings encoded by the feedback function Φ.

3.1 The RankBoost Algorithm

We call our boosting algorithm RankBoost, and its pseudocode is shown in Figure 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 ranking*. RankBoost maintains a distribution D_t over $\mathcal{X} \times \mathcal{X}$ 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.

3. See (Cohen, Schapire, and Singer, 1999) for the list of search templates.

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

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

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

Figure 1: The RankBoost algorithm.

Weak rankings have the form $h_t : \mathcal{X} \to \mathbb{R}$. We think of these as providing ranking information in the same manner as ranking features and the final ranking. The weak learner we used in our experiments is based on the given ranking features; details are given in Section 4.

The boosting algorithm uses the weak rankings to update the distribution as shown in Figure 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 ranking H is a weighted sum of the weak rankings. In the following theorem we prove a bound on the ranking loss of H. This theorem also provides guidance in choosing α_t and in designing the weak learner as we discuss below. on the training data. As in standard classification problems, the loss on a separate test set can also be theoretically bounded given appropriate assumptions using uniform-convergence theory (Bartlett, 1998, Haussler, 1992, Schapire et al., 1998, Vapnik, 1982). In Section 5 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 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

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

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

$$\sum_{x_0, x_1} D(x_0, x_1) [\![H(x_0) \geq H(x_1)]\!] \quad \leq \quad \sum_{x_0, x_1} D(x_0, x_1) \exp(H(x_0) - H(x_1))$$

$$= \sum_{x_0,x_1} D_{T+1}(x_0,x_1) \prod_t Z_t = \prod_t Z_t.$$

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.

3.2 Choosing α_t and Criteria for Weak Learners

In view of the bound established in Theorem 1, we are guaranteed to produce a combined ranking 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 rankings. 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 ranking 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 (1999).

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

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

which yields

$$Z = W_0 + 2\sqrt{W_- W_+}. \tag{3}$$

Thus, if we are using weak rankings with range restricted to $\{0,1\}$, we should attempt to find h that tends to minimize Equation (3) and we should then set α as in Equation (2).

Third method. For weak rankings with range $[0,1]$, we can use a third method of 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

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

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

$$\begin{aligned} Z &\leq \sum_{x_0,x_1} D(x_0,x_1)\left[\left(\frac{1+h(x_0)-h(x_1)}{2}\right)e^{\alpha}+\left(\frac{1-h(x_0)+h(x_1)}{2}\right)e^{-\alpha}\right] \\ &= \left(\frac{1-r}{2}\right)e^{\alpha}+\left(\frac{1+r}{2}\right)e^{-\alpha} \end{aligned} \tag{4}$$

where

$$r = \sum_{x_0,x_1} D(x_0,x_1)(h(x_1)-h(x_0)). \tag{5}$$

The right hand side of Equation (4) is minimized when

$$\alpha = \tfrac{1}{2}\ln\left(\frac{1+r}{1-r}\right) \tag{6}$$

which, plugging into Equation (4), yields $Z \leq \sqrt{1-r^2}$. Thus, to approximately minimize Z using weak rankings with range $[0,1]$, we can attempt to maximize $|r|$ as defined in Equation (5) and then set α as in Equation (6). This is the method used in our experiments.

We now consider the case when any of these three methods for setting α assign a weak ranking h a weight $\alpha < 0$. For example, according to Equation (2), α is negative if W_+, the weight of misordered pairs, is greater than W_-, the weight of correctly ordered pairs. Similarly for Equation (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 rankings 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*.

3.3 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 $\mathcal{X}$ 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.

The results in this section can also be extended to the case in which the feedback function is not itself bipartite, but can nevertheless be decomposed into a sum of bipartite feedback functions. For

Algorithm **RankBoost.B**
Given: disjoint subsets X_0 and X_1 of $\mathcal{X}$.
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 Equation (7)).
- Get weak ranking $h_t : \mathcal{X} \to \mathbb{R}$.
- Choose $\alpha_t \in \mathbb{R}$.
- Update:

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

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

$$Z_t^1 = \sum_{x \in X_1} v_t(x) \exp(-\alpha_t h_t(x))$$

$$Z_t^0 = \sum_{x \in X_0} v_t(x) \exp(\alpha_t h_t(x))$$

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

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

instance, this is the case for the meta-search problem described in Sections 2.1 and 6.1. However, for the sake of simplicity, we omit a full description of this straightforward extension and instead restrict our attention to the simpler case.

If RankBoost is implemented naively as in Section 3.2, 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, $\mathcal{X}_\Phi = X_0 \cup X_1$.

The main idea is to maintain a set of weights v_t over $\mathcal{X}_\Phi$ (rather than the two-argument distribution D_t), and to maintain the condition that, on each round,

$$D_t(x_0, x_1) = v_t(x_0) v_t(x_1) \tag{7}$$

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 2. Equation (7) can be proved 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 2 to see that it is equivalent to the computation of Z_t in Figure 1. To show that Equation (7) holds on round $t+1$, we have, for crucial pair x_0, x_1:

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

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. Fortunately, if we want the weak learner to maximize $|r|$ as in Equation (5), then we also only need to pass $|\mathcal{X}_\Phi|$ weights to the weak learner, all of which can be computed in time linear in $|\mathcal{X}_\Phi|$. 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

$$\begin{aligned} r &= \sum_{x_0,x_1} D(x_0,x_1)(h(x_1) - h(x_0)) \\ &= \sum_{x_0 \in X_0} \sum_{x_1 \in X_1} v(x_0)v(x_1)\,(h(x_1)s(x_1) + h(x_0)s(x_0)) \\ &= \sum_{x_0 \in X_0} \left(v(x_0) \sum_{x_1 \in X_1} v(x_1) \right) s(x_0)\,h(x_0) + \sum_{x_1 \in X_1} \left(v(x_1) \sum_{x_0 \in X_0} v(x_0) \right) s(x_1)\,h(x_1) \\ &= \sum_x d(x)s(x)h(x) \end{aligned} \tag{8}$$

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. Finding Weak Rankings

As described in Section 3, our algorithm RankBoost requires access to a weak learner to produce weak rankings. 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 ranking h that is equal to one of the ranking features f_i, except on unranked instances. 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 URLs by different search engines).

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

$$h(x) = \begin{cases} 1 & \text{if } f_i(x) > \theta \\ 0 & \text{if } f_i(x) \le \theta \\ q_{\text{def}} & \text{if } f_i(x) = \perp \end{cases} \tag{9}$$

where $\theta \in \mathbb{R}$ and $q_{\text{def}} \in \{0,1\}$. That is, a weak ranking 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 ranking 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 rankings are $\{0,1\}$-valued, we can use either the second or third methods described in Section 3.2 to guide us in our search for a weak ranking. 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 ranking that minimizes Equation (3). For a given candidate weak ranking, we can directly compute the quantities W_0, W_-, and W_+, as defined in Section 3.2, in $O(|\Phi|)$ time. Moreover, for each of the n ranking features, there are at most $|\mathcal{X}_\Phi| + 1$ thresholds to consider (as defined by the range of f_i on $\mathcal{X}_\Phi$) and two possible default scores (0 and 1). Thus a straightforward implementation of the second method requires $O(n|\Phi||\mathcal{X}_\Phi|)$ time to generate a weak ranking.

The third method of Section 3.2 requires maximizing $|r|$ as given by Equation (5) and has the disadvantage that it is based on an approximation of Z. However, although a straightforward implementation also requires $O(n|\Phi||\mathcal{X}_\Phi|)$ time, we will show how to implement it in $O(n|\mathcal{X}_\Phi| + |\Phi|)$ time. (In the case of bipartite feedback, if the boosting algorithm of Section 3.3 is used, only $O(n|\mathcal{X}_\Phi|)$ 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:

$$\begin{aligned} r &= \sum_{x_0,x_1} D(x_0,x_1)\,(h(x_1) - h(x_0)) \\ &= \sum_{x_0,x_1} D(x_0,x_1)h(x_1) - \sum_{x_0,x_1} D(x_0,x_1)h(x_0) \\ &= \sum_x h(x)\sum_{x'} D(x',x) - \sum_x h(x)\sum_{x'} D(x,x') \\ &= \sum_x h(x)\sum_{x'}(D(x',x) - D(x,x')) \\ &= \sum_x h(x)\,\pi(x)\ , \end{aligned} \tag{10}$$

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 $O(|\mathcal{X}_\Phi|)$ space. When the feedback is bipartite, comparing Equations (8) and (10), we see that $\pi(x) = d(x)s(x)$ where d and s are defined in Section 3.3; thus, in this case, π can be computed even faster in only $O(|\mathcal{X}_\Phi|)$ time.

Now let us address the problem of finding a good threshold value θ and default value q_{def}. We need to scan the candidate ranking features f_i and evaluate $|r|$ (defined by Equation (10)) for each possible choice of f_i, θ and q_{def}. Substituting into Equation(10) the h defined by Equation (9), we have that

$$\begin{aligned} r &= \sum_{x: f_i(x) > \theta} h(x)\pi(x) + \sum_{x: f_i(x) \le \theta} h(x)\pi(x) + \sum_{x: f_i(x) = \perp} h(x)\pi(x) && (11) \\ &= \sum_{x: f_i(x) > \theta} \pi(x) + q_{\text{def}} \sum_{x: f_i(x) = \perp} \pi(x). && (12) \end{aligned}$$

For a fixed ranking feature f_i, let $\mathcal{X}_{f_i} = \{x \in \mathcal{X}_\Phi \mid f_i(x) \neq \perp\}$ be the set of feedback instances ranked by f_i. We only need to consider $|\mathcal{X}_{f_i}| + 1$ threshold values, namely, $\{f_i(x) \mid x \in \mathcal{X}_{f_i}\} \cup \{-\infty\}$ since these define all possible behaviors on the feedback instances. Moreover, we can straightforwardly compute the first term of Equation (12) for *all* thresholds in this set in time $O(|\mathcal{X}_{f_i}|)$ simply by scanning down a pre-sorted 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_{def} (0 or 1). To do this, we simply need to evaluate $\sum_{x: f_i(x) = \perp} \pi(x)$ once. Naively, this takes $O(|\mathcal{X}_\Phi - \mathcal{X}_{f_i}|)$ 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

$$\sum_{x: f_i(x) = \perp} \pi(x) = - \sum_{x: f_i(x) \neq \perp} \pi(x). \tag{13}$$

The right hand side of this equation can clearly be computed in $O(|\mathcal{X}_{f_i}|)$ time. Combining Equations (12) and (13), we have

$$r = \sum_{x: f_i(x) > \theta} \pi(x) - q_{\text{def}} \sum_{x \in \mathcal{X}_{f_i}} \pi(x). \tag{14}$$

The pseudocode for the weak learner is given in Figure 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 Equation (14): the variable L stores the left summand and the variable R stores the right summand. Finally, if the default rank q_{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 rankings is only linear in the number of instances that are ranked by that feature. In summary, we have shown the following theorem:

Theorem 2 *The algorithm of Figure 3 finds the weak ranking of the form given in Equation (9) that maximizes $|r|$ as in Equation (10). The running time is $O(n|\Phi||\mathcal{X}_\Phi|)$ per round of boosting. An efficient implementation runs in time*

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

If the feedback is bipartite, the running time can be improved to $O(n|\mathcal{X}_\Phi|)$.

Algorithm **WeakLearn**

Given: distribution D over $\mathcal{X} \times \mathcal{X}$
set of features $\{f_i\}_{i=1}^N$
for each f_i, the set $\mathcal{X}_{f_i} = \{x_k\}_{k=1}^K$ 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 $x \in \mathcal{X}_\Phi$, $\pi(x) = \sum_{x' \in \mathcal{X}_\Phi} D(x',x) - D(x,x')$; $r^* = 0$

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

1. $L = 0$
2. $R = \sum_{x \in \mathcal{X}_{f_i}} \pi(x)$ /* $L - q_{\text{def}}R$ is rhs of Equation (14) */
3. $\theta_0 = \infty$
4. For $j = 1, \ldots, J$:
5. $\quad L = L + \sum_{x:\theta_{j-1} \geq f_i(x) > \theta_j} \pi(x)$ /* compute $L = \sum_{x: f_i(x) > \theta} \pi(x)$ */
6. $\quad$ if $|L| > |L - R|$ /* find best value for q_{def} */
7. $\quad\quad$ then $q = 0$
8. $\quad\quad$ else $q = 1$
9. $\quad$ if $|L - qR| > |r^*|$ /* find best weak ranking */
10. $\quad\quad r^* = L - qR$
11. $\quad\quad i^* = i$
12. $\quad\quad \theta^* = \theta_j$
13. $\quad\quad q^*_{\text{def}} = q$

Output weak ranking $(f_{i^*}, \theta^*, q^*_{\text{def}})$

Figure 3: The weak learner.

Positive cumulative weights. Since the final ranking has the form $H(x) = \sum_{t=1}^T \alpha_t h_t(x)$ and the rankings 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 rankings that correspond to different thresholds of the same feature f. If we view these rankings 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 rankings 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 ranking (in order to avoid overfitting). In summary, while RankBoost may set some α_t's to be negative, we developed an alternative version that enforces the constraint that all of the values of α_t's are positive. Thus, each ranking feature is forced to be positively correlated with the final ranking that RankBoost outputs.

To address this situation, we implemented an additional version of WeakLearn that chooses its rankings to exhibit this monotonic behavior. In practice, our earlier assumption that all h_t's are unique may not hold. If it does not, then the contribution of a particular ranking 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 ranking that maximizes $|r|$ subject to the constraint that the cumulative weight of that ranking remains positive. We refer to this modified weak learner as WeakLearn.cum.

5. Generalization Error

In this section, we derive a bound on the generalization error of the combined ranking when the weak rankings are binary functions and the feedback is bipartite. That is, we assume that the feedback partitions the instance space $\mathcal{X}$ into two disjoint 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 Section 6, as well as many of the problems in information retrieval (Salton, 1989, Salton and McGill, 1983).

5.1 Probabilistic Model

Up to this point we have not discussed where our training and test data come from. The usual assumption of machine learning 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 to 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 separate 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 $\mathcal{X} \times \mathcal{X}$, pairs from the instance space.[4] The obvious next step would be to declare the training set 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, as describe by Haussler et al. 1991.) 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

4. 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.

which movies to view based on movies they've seen. However, such independence assumptions are common in machine learning.

5.2 Sampling Error Definitions

Given this probabilistic model of the ranking problem, we can now define generalization error. The final ranking 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 rankings on pairs of instances, so we consider rankings of the form $H : \mathcal{X} \times \mathcal{X} \to \{-1, 0, +1\}$, where

$$H(x,y) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(y) - \sum_{t=1}^{T} \alpha_t h_t(x)\right) \tag{15}$$

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

A function 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

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

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 expected test error of H is

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

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 $\varepsilon(H)$ is small, meaning that the performance of the combined ranking H on the training sample is representative of its performance on any random sample.

5.3 VC Analysis

We now bound the difference between the training error and test error of the combined ranking output by RankBoost using standard VC-dimension analysis techniques (Devroye et al., 1996, Vapnik, 1982). 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 ranking is chosen by our algorithm, the training error of the combined ranking will accurately estimate its generalization error. Another way of saying this is as follows. Let Z denote the event that there exists an $H \in \mathcal{C}$ such that $\hat{\varepsilon}(H)$ and $\varepsilon(H)$ differ by more than a small specified amount. Then, the probability (over the choice of training set) of the event Z is very small. Formally, we will show that for every $\delta > 0$, there exists a small ε such that

$$\Pr_{S_0 \sim D_0^m, S_1 \sim D_1^n}\left[\exists H \in \mathcal{C} \,:\, \left|\frac{1}{mn}\sum_{i,j} [\![H(x_i, y_j) \neq 1]\!] - \mathrm{E}_{x,y}\left[[\![H(x,y) \neq 1]\!]\right]\right| > \varepsilon\right] \leq \delta \tag{16}$$

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

Our approach will be to separate (16) 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 $\mathcal{X} \times \mathcal{X}$, 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 (16) can then be rewritten as

$$\begin{aligned}
&\frac{1}{mn}\sum_{i,j} F(x_i,y_j) - \mathrm{E}_{x,y}\left[F(x,y)\right] \\
&= \frac{1}{mn}\sum_{i,j} F(x_i,y_j) - \frac{1}{m}\sum_i \mathrm{E}_y\left[F(x_i,y)\right] + \\
&\quad \frac{1}{m}\sum_i \mathrm{E}_y\left[F(x_i,y)\right] - \mathrm{E}_{x,y}\left[F(x,y)\right] \\
&= \frac{1}{m}\sum_i \left(\frac{1}{n}\sum_j F(x_i,y_j) - \mathrm{E}_y\left[F(x_i,y)\right]\right) + \qquad (17) \\
&\quad \mathrm{E}_y\left[\frac{1}{m}\sum_i F(x_i,y) - \mathrm{E}_x\left[F(x,y)\right]\right] . \qquad (18)
\end{aligned}$$

So if we prove that there exist ε_0 and ε_1 such that $\varepsilon_0 + \varepsilon_1 = \varepsilon$ and

$$\Pr_{S_1 \sim D_1^n}\left[\exists F \in \mathcal{F}, \exists x \in X : \left|\frac{1}{n}\sum_j F(x,y_j) - \mathrm{E}_y\left[F(x,y)\right]\right| > \varepsilon_1\right] \leq \delta/2 , \tag{19}$$

$$\Pr_{S_0 \sim D_0^m}\left[\exists F \in \mathcal{F}, \exists y \in Y : \left|\frac{1}{m}\sum_i F(x_i,y) - \mathrm{E}_x\left[F(x,y)\right]\right| > \varepsilon_0\right] \leq \delta/2 , \tag{20}$$

we will have shown (16), because with high probability, the summand of (18) will be less than ε_1 for every x_i, which implies that the average will be less than ε_1. Likewise, the quantity inside the expectation of (18) will be less than ε_0 for every y and so the expectation will be less than ε_0.

We now prove (20) using standard classification results, and (19) follows by a symmetric argument. Consider (20) 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 (20) comes from $\bigcup_y \mathcal{F}_y$. A theorem of Vapnik (1982) applies to (20) 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 1982 or Devroye et al. 1996). Specifically, for any $\delta > 0$,

$$\Pr_{S_0 \sim D_0^m}\left[\exists F \in \bigcup \mathcal{F}_y \ : \ \left|\frac{1}{m}\sum_i F(x_i,y) - \mathrm{E}_x\left[F(x,y)\right]\right| > \varepsilon_0(m,\delta,d')\right] < \delta \,,$$

where

$$\varepsilon_0(m,\delta,d') = 2\sqrt{\frac{d'(\ln(2m/d')+1)+\ln(9/\delta)}{m}} \,.$$

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 (20), the probability corresponds to a peculiar classification problem (trying to differentiate X from Y by picking an F and one $y \in Y$) that does not seem to have a natural interpretation.)

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

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

where $b = \sum_{t=1}^{T}\alpha_t h_t(y)$ is constant because y is fixed. So the functions in $\bigcup_y \mathcal{F}_y$ are a subset of all possible thresholds of all linear combinations of T weak rankings. Freund and Schapire's (1997) Theorem 8 bounds the VC-dimension of this class in terms of T and the VC-dimension of the weak ranking 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 (19) keeping x fixed, and putting it all together, we have thus proved the main result of this section:

Theorem 3 *Let $\mathcal{C}$ be the set of all functions of the form given in Eq (15) where all the h_t's belong to a class $\mathcal{H}$ of VC-dimension d. Let S_0 and S_1 be samples of size m and n, respectively. Then with probability at least $1-\delta$ over the choice of training sample, all $H \in \mathcal{C}$ satisfy*

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

where $d' = 2(d+1)(T+1)\log_2(e(T+1))$.

6. Experimental Evaluation of RankBoost

In this section, 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 ranking on the test data. Details are given below.

6.1 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.

6.1.1 Description of Task and Data Set

Most of the details of this dataset and how we mapped it into the general ranking framework were described in Section 2.1.

Given this mapping of the ranking problem into our framework, we can immediately apply RankBoost. Note that the feedback function for this problem is a sum of bipartite feedback functions so the more efficient implementation described in Section 3.3 can be used.

Under this mapping, each weak ranking 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 ranking outputs 1 or 0 if u is ranked above or below the threshold θ on the list of URLs returned by the expanded query associated with search template i applied to base query q. As usual, the final ranking H is a weighted sum of the weak rankings.

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.

6.1.2 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 rankings, to be 0 (see Section 4).

Our implementation of RankBoost used a definition of ranking loss modified from the original given in Section 2, Equation (1):

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

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

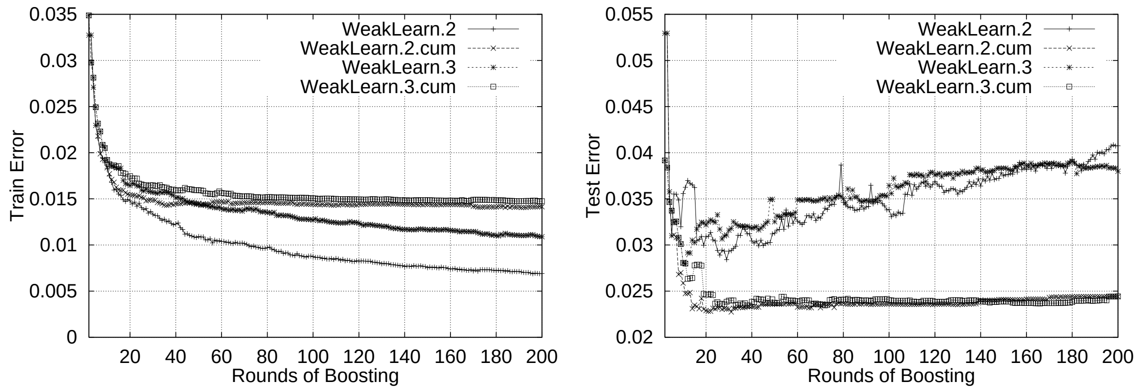

Figure 4: Performance of the four weak learners WeakLearn.{2,3,2.cum,3.cum} on the ML dataset. Left: Train error Right: Test error

(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

$$\text{rloss}_D(H) = \sum_{x_0,x_1} D(x_0,x_1) \, [\![H(x_1) < H(x_0)]\!] + \tfrac{1}{2} \sum_{x_0,x_1} D(x_0,x_1) \, [\![H(x_1) = H(x_0)]\!] \; . \tag{21}$$

RankBoost parameters. Since WeakLearn outputs binary weak rankings, we can set the parameter α using either the second or third methods presented in Section 3.2. 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 rankings to have positive cumulative weights in order to test whether such rankings were helpful or harmful in reducing test error (as discussed at the end of Section 4). 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 we plotted the train and test error of the combined ranking 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 4 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 rankings to have positive cumulative weights hampers training performance but improves test performance. Also, when we subject the rankings to this restriction, we see no difference between the second and third methods of setting α. Therefore, in our experiments we used WeakLearn.3.cum, the third method of setting α that allows only positive cumulative ranking weights.

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 1: Comparison of the combined ranking and individual search templates. See text for further explanation of the results.

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 Figure 4, left), so we used the final ranking 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 ranking built after 40 rounds.

To evaluate the performance of the individual search templates in comparison to the combined ranking output by RankBoost, we measured the number of queries for which the correct document was in the top k ranked documents, for $k = 1, 2, 5, 10, 20, 30$. We then compared the performance of the combined ranking to that of the best search template for each value of k. The results for the ML and UNIV domains are shown in Table 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 ranking 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[5] for all values of k, and the combined ranking 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 URLs, 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 URLs was larger than 30, RankBoost assigned to some

5. The best search template for the UNIV domain was `"NAME" PLACE`.

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 1 gives average rank.

6.2 Movie Recommendations

Our second set of experiments dealt with the movie-recommendation task described in the introduction. The goal here 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 the 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.

6.2.1 Description of Task and Data Set

For these experiments we used publicly available data[6] 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 2. 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 2 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 training and 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 6.2.5).

6.2.2 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 Equation (21). 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 set the number of rounds T to be $40 + N/10$ where N is the number of features. This choice was based on performance on held-out data which was not used in any of the other experiments.

6.2.3 Algorithms for Comparison

We compared the performance of RankBoost on this data set to three other algorithms, a regression algorithm, a nearest-neighbor algorithm, and a memory-based algorithm called vector similarity.

6. From `http://www.research.digital.com/SRC/eachmovie/`.

Regression. We used a regression algorithm similar to the ones used by Hill et al. (1995). 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 ranking 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).

Vector Similarity (VSIM). This algorithm was proposed by Breese, Heckerman and Kadie (1998) and is based on the notion of similarity between vectors that is commonly used in information retrieval. In the field of information retrieval, the similarity between two documents is often measured by treating each document as a vector of term frequencies. The similarity between two documents is defined to be the normalized inner-product formed by the two frequency vectors representing the different documents (Salton and McGill, 1983). Breese, Heckerman and Kadie adopted this formalism for the task of collaborative filtering by viewing the rating of each viewer as a sparse vector over the reals. In their setting, the users take the role of documents, movies take the role of the terms appearing in documents, and viewers' scores take the role of term frequencies. Let $\mathbf{C}_i$ denote the scores of the ith viewer. Then correlation between the jth viewer and the ith viewer is

$$w_{i,j} = \frac{\mathbf{C}_i \cdot \mathbf{C}_j}{\parallel \mathbf{C}_i \parallel_2 \parallel \mathbf{C}_j \parallel_2} \ ,$$

where both the inner product and the norms are computed over the subset of movies rated by each viewer. To accommodate different scales, Breese, Heckerman and Kadie also compute for each viewer i her average score, denoted $\bar{\mathbf{C}}_i$. To predict the rating of a new viewer, indexed k, we first compute the similarity coefficients $w_{k,i}$ with each previous viewer i and then assign a real-valued score $\hat{\mathbf{C}}_{k,j}$ for each movie j as follows,

$$\hat{\mathbf{C}}_{k,j} = \bar{\mathbf{C}}_k + \alpha \sum_i w_{k,i}(\mathbf{C}_{i,j} - \bar{\mathbf{C}}_i) \ ,$$

where α is a normalizing factor which ensures that $\sum_i |w_{k,i}| = 1$. We use the abbreviation VSIM when referring to this algorithm. VSIM and another correlation-based algorithm were found to be the top performers in the experiments performed by Breese, Heckerman and Kadie (1998) with the EachMovie dataset. Furthermore, in the experiments they described, VSIM outperformed four other algorithms when the number of movies that were rated was small (less than 5).

6.2.4 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 (Equation (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

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

This is equivalent to Equation (21) where c is used to construct the feedback function.[7]

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

7. This disagreement measure is proportional to another measure of two Linear orders, the Pearson r correlation coefficient, that was found by Shardanand and Maes to be the best in their collaborative filtering experiments (Shardanand and Maes, 1995)

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 to 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 to which Alice assigns her highest score are considered relevant. As discussed in Section 3.3, 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 $\mathrm{rank}(m)$, 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 $1/\mathrm{rank}(t_1)$, and, more generally, the precision of the kth good movie is $k/\mathrm{rank}(t_k)$. Again, if all K good movies appear one after another at the top of H's list, meaning $\mathrm{rank}(t_k) = k$ 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

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

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

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

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

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

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

If H is a partial order, its expected PROT is

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

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. Formally, it is

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

If H is a partial order, its expected coverage is

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

6.2.5 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 divided T into six subsets $T_1, T_2, \ldots, T_6$ of respective sizes 100, 200, 500, 750, 1000, 2000, such that $T_1 \subset T_2 \subset \cdots \subset T_6$. Each T_j 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.

RankBoost clearly outperformed regression and NN for all four performance measures. RankBoost also outperformed VSIM when the feature set size was greater than 200. For medium and large feature sizes, RankBoost achieved 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 (whose PROT was 0.45). 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 6.2.3). 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 (whose disagreement was 0.5). Regression did perform better than random guessing on PROT and coverage, but on AP it was worse than random guessing (whose AP was 0.41). 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.

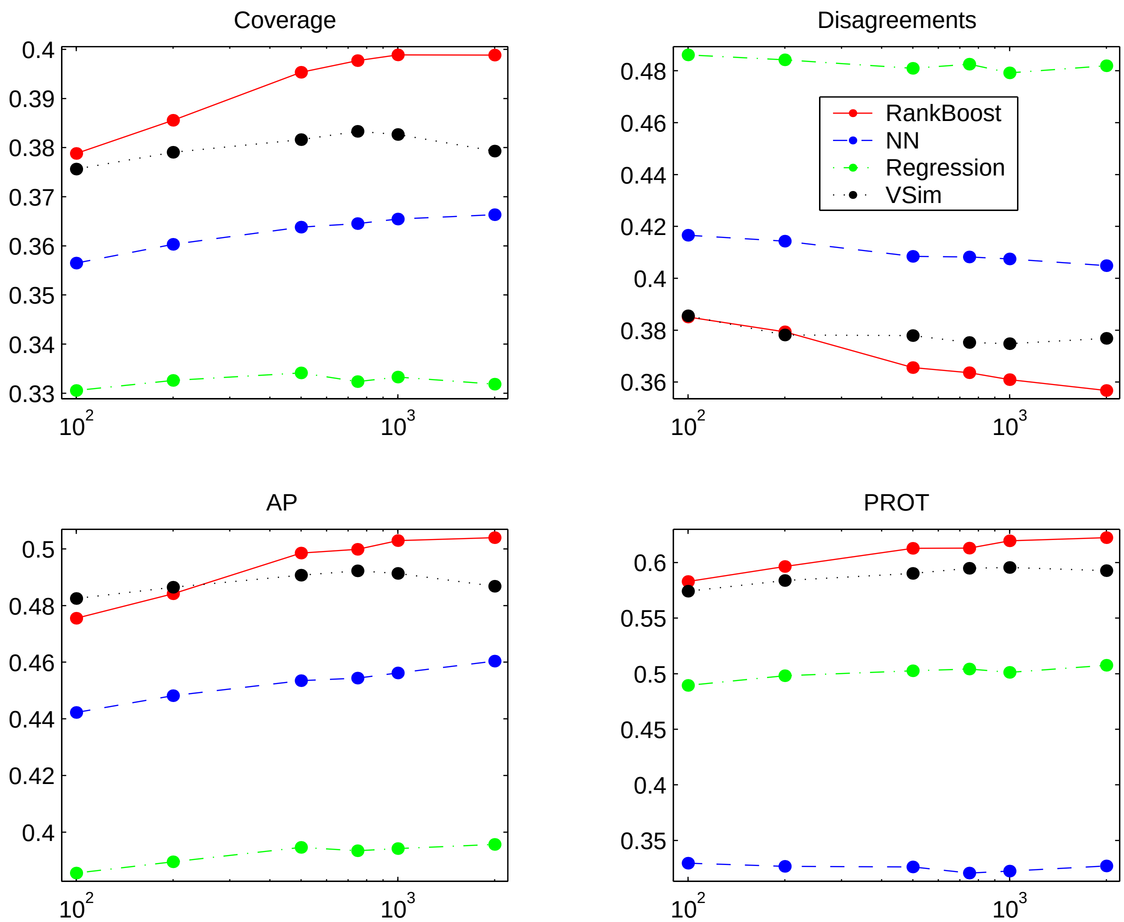

Figure 5: Performance of algorithms as a function of the size of the features' sets.

The performance of VSIM was very close to RankBoost for feature sets of size 100 and 200. This performance is especially impressive considering the fact that RankBoost attempts to minimize the number of disagreements while VSIM is a rather simple approach based on correlations. A similar behavior for VSIM was observed by Breese, Heckerman and Kadie (1998) in the experiments they performd with the EachMovie dataset. However, VSIM does not seem to scale as well as RankBoost when the size of the feature set increases. Like regression, it seems that VSIM did not make much use of the additional information provided by a large number of features and, in fact, its performance degraded when moving from feature sets of size 750 to size 1000. We defer further discussion of possible reasons for this behavior of VSIM compared to RankBoost to the end of this section.

In our next experiment, we explored the effect of the density of the features, the 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

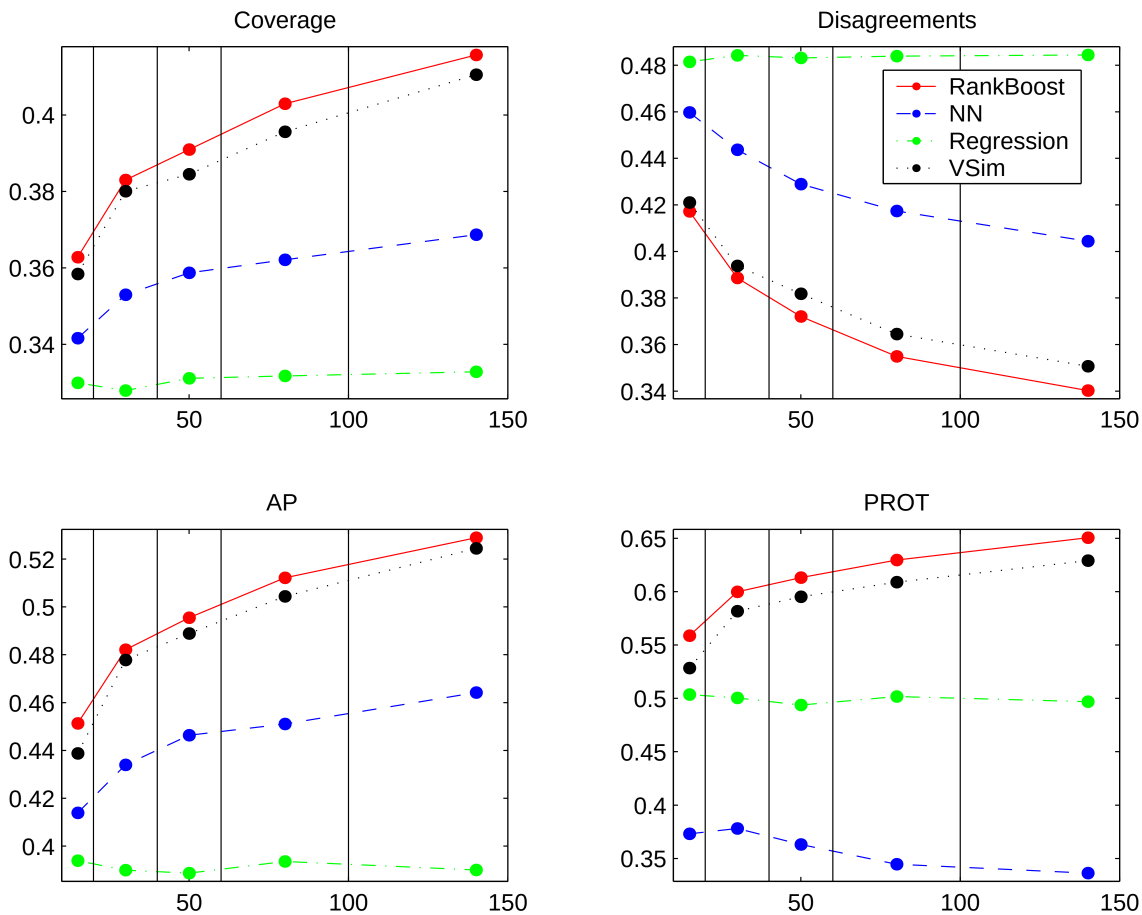

Figure 6: Performance of the algorithms on different feature densities.

measures on each split, and then averaged them together. The results are shown in Figure 6. 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 similar to that in Figure 5. 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, NN's performance on AP, disagreement, and coverage improved more significantly than when simply the number of features increased (Figure 5). However, on PROT NN continued to perform worse than random guessing (whose PROT was 0.45). Furthermore, the performance of NN degraded as feature density increased. Regression maintained the same relative performance to random guessing as in the previous experiment, and its performance was largely unaffected as feature density increased.

Here again VSIM comes up the second best and its performance is close to the performance of RankBoost with respect to all four performance measures. Furthermore, like RankBoost, VSIM seems to scale well as the feature density increases. The rate of increase seems comparable to that of RankBoost for Coverage, AP, and PROT, and with a slightly slower increase in the case of disagreements. Although the differences in the performance between RankBoost and VSIM are

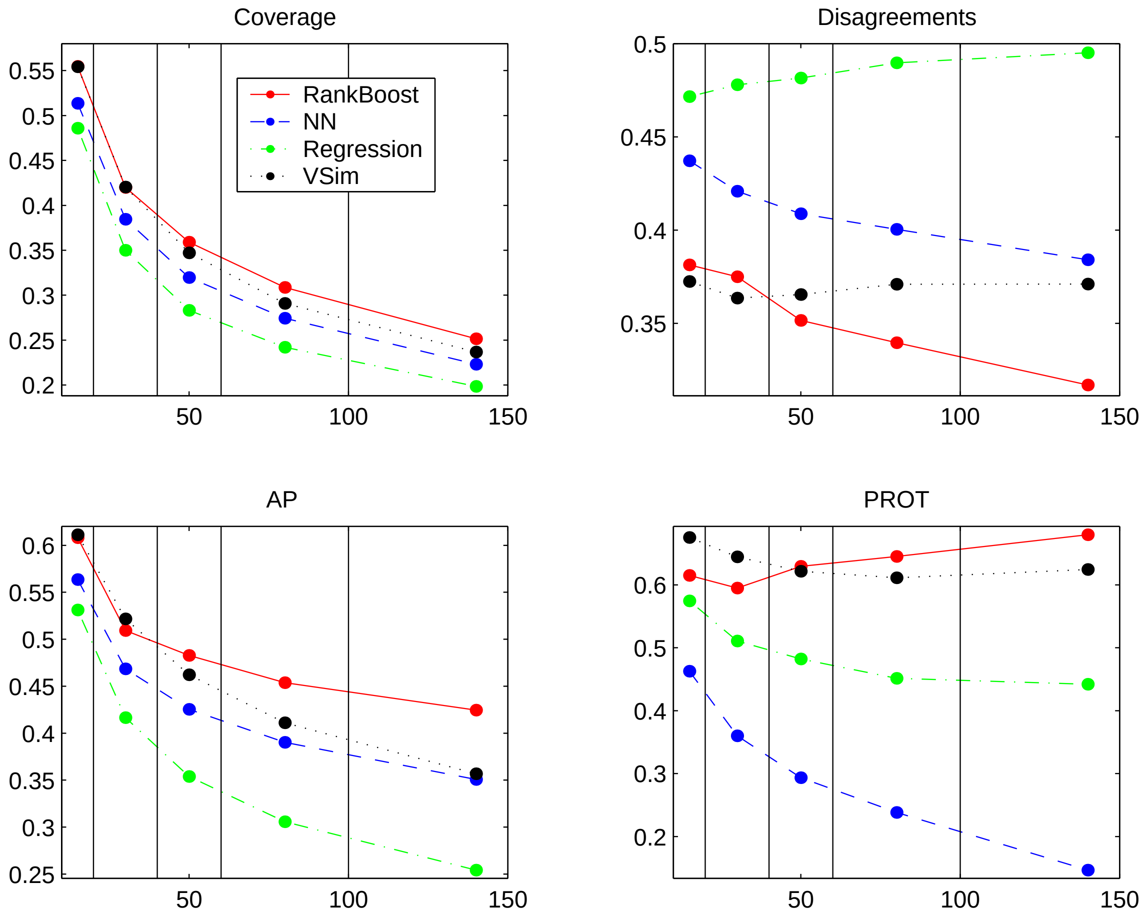

Figure 7: Performance of algorithms on different feedback densities.

statistically significant, the advantage of RankBoost over VSIM is far less pronounced than the overwhelmingly better performance of RankBoost compared to NN and regression.

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 7.

The most noticeable effect of increasing the feedback density is that it *degrades* the performance of all four algorithms on coverage and AP and the performance of VSIM, NN, and regression on PROT (RankBoost is able to improve PROT). Other than that, the comparative performance of the algorithms to one another was similar to the results in the previous experiments, with the exception that the differences between RankBoost and VSIM, as a function of the feedback densities, are more pronounced.

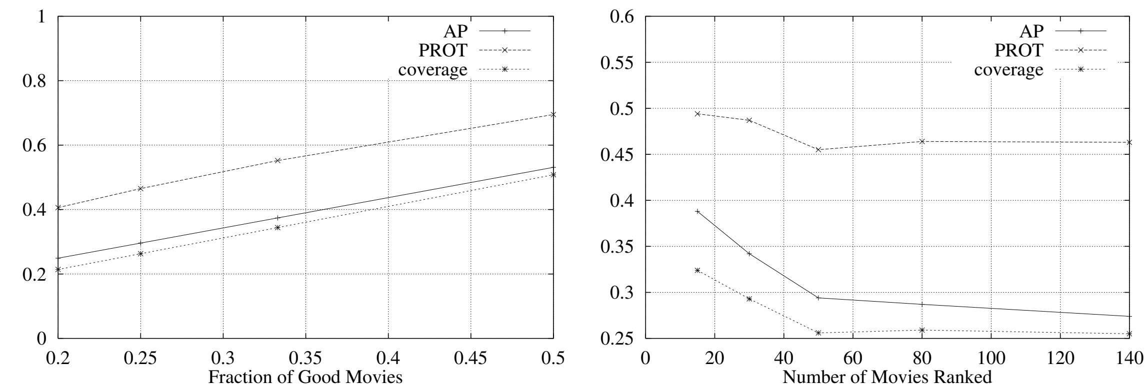

Figure 8: Left: The performance of random guessing when the fraction of good movies is varied (the number of movies is 60). Right: The performance of random guessing when the number of movies ranked is varied (the fraction of good movies is 0.25).

Especially interesting is the good performance of VSIM when the feedback density is at most 40. In this case, VSIM is the best performing algorithm with respect to disagreements, AP, and PROT, and achieves practically the same coverage as RankBoost. However, as the density of the feedback grows, the performance of VSIM deteriorates and for feedback densities of over 100 the performance of VSIM becomes mediocre and it is comparable to NN with respect to all of the performance measures, with the exception of PROT. In contrast, RankBoost consistently improves as the size of the feedback set grows. This behavior of RankBoost is common in supervised learning algorithms which typically improve proportionally to the amount of supervision they get.

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 and VSIM, as in the first set of 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, we observed that the performance of random guessing also degrades as the feedback density increases, which suggests that the ranking problem is intrinsically 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 8 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 8, 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 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 6.2.3. 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.

6.2.6 DISCUSSION

Our experiments show that RankBoost clearly performed better than regression and nearest neighbor on the movie recommendation task. It also performs better than VSIM when the feature size or the feedback density is relatively large.

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 6.2.3). 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, so this proposed experiment would further test our hypothesis that relative preferences are more informative.

In our experiments, VSIM was the algorithm that achieved the closest performance to RankBoost, and for small-scale problems, it often achieved better performance than RankBoost. However, as the complexity of the training data (i.e., feature size) or the amount of supervision (feedback) increases, RankBoost's performance gets a boost while the performance of VSIM seems to asymptote or even degrade. A possible explanation is that RankBoost, being a learning algorithm that attempts to minimize an objective function designed for ranking, is able to build more complex hypotheses as the amount and variety of training examples increase. VSIM, on the other hand, employs a fixed memory-based correlation paradigm and thus exhibits less flexibility and adaptability as the number of training examples grows. Put another way, since RankBoost builds a different ranking function separately for each user, it has a higher potential to exploit information conveyed by individual users who rated a large number of movies. We would also like to note that it seems possible to take the best from each approach: RankBoost can use VSIM as one of the possible weak ranking functions it can choose on each round. The end result would be an algorithm that resorts to a simple memory-based correlation when the amount of data is small and gradually shifts to discriminative approaches as the amount of training data grows.

7. Conclusion

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.

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 $\mathcal{X}$, n ranking features, and a feedback function of size $|\Phi|$ that ranks a subset of the instances $\mathcal{X}_\Phi \subseteq \mathcal{X}$. Given this input, RankBoost generally runs in $O(|\Phi|)$ time, and a naive implementation of the weak learner we present runs in $O(n|\Phi|\mathcal{X}_\Phi)$ time. We have shown two improvements in efficiency, as summarized in Theorem 2 of Section 4. If we use binary weak rankings and we search for the best weak ranking using the third method in Section 3.2, then we can implement the weak learner in time $O(n|\mathcal{X}_\Phi| + |\Phi|)$. 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 rankings, we can implement the weak learner in $O(n|\mathcal{X}_\Phi|)$ time.

These two restriction are both natural and useful. Binary rankings are quite simple and this makes them easy to design, analyze, and compute efficiently. Although a single such ranking 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 the weak ranking. 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 and was consistently better than the vector similarity method in medium to large problem settings.

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 URLs) and large feedback (10,000 URLs). 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).

One important inherent feature of RankBoost, being a boosting algorithm, is its ability to combine different approaches for ranking. While the task of combining general ranking features given non-bipartite feedback can be rather involved, the boosting-for-ranking framework that we introduced in this paper offers a principled and efficient algorithmic infrastructure. Therefore, RankBoost can also be used as a tool for building a hybrid ranking system that combines different ranking algorithms, yielding a high precision and recall ranking meta-algorithm.

Current directions. There are numerous 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 6.2.6): 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. We could compare RankBoost's ordering to AdaBoost's ordering of the instances by classification weight to see if minimizing ranking loss is superior to minimizing classification error.

As for the RankBoost algorithm itself, the first method for setting α_t is the most general and requires numerical search. Schapire and Singer (1999) 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. In a recent paper (Iyer et al., 2000), two versions of RankBoost were compared to traditional information retrieval approaches. The experiments in the paper indicate that RankBoost can provide an alternative approach of combining term weights, however, RankBoost's performance greatly depends on the quality of the feedback that is provided.

Various versions of RankBoost might turn out to be useful in learning problems that at first sight do not seem to be related to ranking. For instance, Walker, Rambow and Rogati (2001) recently used RankBoost successfully to train a sentence-generation system. In other work, Collins (2000), describes experiments using the RankBoost for a natural-language processing task, specifically, to re-rank the candidate parses produced by a probabilistic parser. The paradigm suggested by Collins can be applied to other settings in which the results of an approximate or exact search yield an ordered list of candidates with the "correct" element appearing somewhere down the ordered list. This list can then be re-ranked by applying RankBoost to a fresh set of features.

Finally, we would like to note that this work is part of a general research effort on learning algorithms for ordinal data. We hope that this work will spark further interest in such problems which are challenging and relatively unexplored. Indeed, recent work (Crammer and Singer, 2001, 2002, Lebanon and Lafferty, 2002) on ranking problems indicate that some of the techniques explored in this paper can be carried over to online learning of ranking functions and exponential models for ranking, which makes a connection to work in statistics.

Acknowledgements

Special thanks to David Karger for substantial contributions to this work. Thanks also to William Cohen, Matt Levine, and David Lewis for helpful discussions, and to the anonymous referees of an earlier draft of this paper for their careful reading and useful comments. Part of this research was conducted while all authors were employed by AT&T Labs. In addition, R. Iyer was supported by an NSF Graduate Fellowship. The research of Y. Freund, R. Schapire, and Y. Singer was supported by the US-Israel Bi-national Science Foundation under grant number 1999-038.

Appendix A. 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 6.2.4). 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 $\Pr[\text{rank}(t_k) = i]$, the probability, over all total orders consistent with H, that the kth good movie on H's list occurs at position i on the list. Here we calculate this probability.

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

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

Let r be the number of *good* movies that definitely appear 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,

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

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

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

Then,

$$\Pr[\text{rank}(t_k) = i] = \frac{\binom{i-R-1}{k-r-1}\binom{Q-i+R}{q-k+r}}{\binom{Q}{q}} \ . \tag{22}$$

We prove (22) as follows. Let $j = k - r$. Then when t_k is listed at position i, t_k is the jth good movie appearing within the list of Q tied movies. Define the random variable Y_j to be the rank of t_k within the list of tied movies. For example, if t_k is the second movie listed then $Y_j = 2$. Then

$$\Pr[\text{rank}(t_k) = i] = \Pr[R + Y_j = i] = \Pr[Y_j = \ell] \ , \tag{23}$$

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. (Balls of the same color are indistinguishable.) 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

$$\Pr\left[Y_j = \ell\right] = \frac{\binom{\ell-1}{j-1}\binom{Q-\ell}{q-j}}{\binom{Q}{q}}. \tag{24}$$

Substituting $\ell = i - R$ from (23) into this equation gives (22), the desired result. ■

References

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.

Peter L. Bartlett. The sample complexity of pattern classification with neural networks: the size of the weights is more important than the size of the network. *IEEE Transactions on Information Theory*, 44(2):525–536, March 1998.

John S. Breese, David Heckerman, and Carl Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 43–52, 1998.

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.

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

Michael Collins. Discriminative reranking for natural language parsing. In *Proceedings of the Seventeenth International Conference on Machine Learning*, 2000.

K. Crammer and Y. Singer. A new family of online algorithms for category ranking. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2002.

Koby Crammer and Yoram Singer. Pranking with ranking. In *Advances in Neural Information Processing Systems 14*, 2001.

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

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.

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.

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

David Haussler, Michael Kearns, Nick Littlestone, and Manfred K. Warmuth. Equivalence of models for polynomial learnability. *Information and Computation*, 95(2):129–161, December 1991.

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.

Raj D. Iyer, David D. Lewis, Robert E. Schapire, Yoram Singer, and Amit Singhal. Boosting for document routing. In *Proceedings of the Ninth International Conference on Information and Knowledge Management*, 2000.

Guy Lebanon and John Lafferty. Cranking: Combining rankings using conditional probability models on permutations. In *Proceedings of the Nineteenth International Conference on Machine Learning*, 2002.

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.

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

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

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

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

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.

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

Marilyn A. Walker, Owen Rambow, and Monica Rogati. SPoT: A trainable sentence planner. In *Proceedings of the 2nd Annual Meeting of the North American Chapter of the Associataion for Computational Linguistics*, 2001.