January 2010

An Empirical Analysis on Point-wise Machine Learning Techniques using Regression Trees for Web-search Ranking

Ananth Mohan
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science

Department of Computer Science and Engineering

Thesis Examination Committee:
Dr. Kilian Weinberger, Chair
Dr. Kunal Agrawal
Dr. Robert Pless

AN EMPIRICAL ANALYSIS ON POINT-WISE MACHINE LEARNING TECHNIQUES USING REGRESSION TREES FOR WEB-SEARCH RANKING.

by

Ananth Mohan

A thesis presented to the School of Engineering
of Washington University in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Dec 2010
Saint Louis, Missouri

ABSTRACT OF THE THESIS

An Empirical Analysis on Point-wise Machine Learning Techniques Using Regression Trees for Web-search Ranking.

by

Ananth Mohan

Master of Science in Computer Science

Washington University in St. Louis, 2010

Research Advisor: Dr. Kilian Weinberger

Learning how to rank a set of objects relative to an user defined query has received much interest in the machine learning community during the past decade. In fact, there have been two recent competitions hosted by internationally prominent search companies to encourage research on ranking web site documents. Recent literature on learning to rank has focused on three approaches: point-wise, pair-wise, and list-wise. Many different kinds of classifiers, including boosted decision trees, neural networks, and SVMs have proven successful in the field. This thesis surveys traditional point-wise techniques that use regression trees for web-search ranking. The thesis contains empirical studies on Random Forests and Gradient Boosted Decision Trees, with novel augmentations to them on real world data sets. We also analyze how these point-wise techniques perform on new areas of research for web-search ranking: transfer learning and feature-cost aware models.

Acknowledgments

I would like to thank my research advisor, Dr. Kilian Weinberger, for introducing me to this new field. This thesis would not have been possible without his support and knowledge. The origins of this thesis came from work on an online competition hosted by Yahoo; I would like to thank Dr. Olivier Chapelle and the others who are responsible for organizing and hosting the competition. I'd also like to thank my partner for the Yahoo competition, Zheng Chen. I would also like to thank my thesis committee members: Dr. Kunal Agrawal and Dr. Robert Pless. And finally, I thank my parents for their support and this thesis is dedicated to them.

Ananth Mohan

Washington University in Saint Louis
Dec 2010

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Background

Learning to rank has received much interest in the machine learning community during the past decade. The learning to rank problem builds a model which is used to rank or order a set of objects relative to their relevance or importance to an user defined query. This construct has proven useful in many fields such as Information Retrieval [22], advertisements [18], social networking [9], and collaborative filtering [35]. There are two ranking paradigms, dynamic ranking and static ranking. Dynamic ranking orders objects relative to their importance to a specific query; the relation between an object and a query may be developed in real-time. Static ranking evaluates objects based on criteria that is independent of a query [28], such as PageRank [25] and click rate. Applying dynamic ranking (often called just ranking) to Web-search data sets in order to power web search engines has garnered much academic and industrial interest. In fact, in recent years there have been two open contents to promote research in the field of ranking web-search data sets; they were hosted by Yahoo and Yandex.

The past decade has seen three main approaches to web-search ranking: point-wise, pair-wise, and list-wise. Point-wise algorithms such as McRank [21] and Pranking [13] predict the exact relevance of a document by minimizing the regression error. Pairwise approaches such as RankBoost [14], FRank [32], and GBRank [38] learn a classifier that best predicts if one document is more relevant than another document. List-wise approaches tend to iteratively optimize any performance measure; these approaches include BoltzRank [34], AdaRank [36], PermuRank [37], and SoftRank [31]. Most of

these algorithms implement Gradient Boosted Regression Trees, which has arguably been the most popular learning technique for web-search ranking. However regression trees are not the exclusive learners; support vector machines [8], and neural networks [6] have also proven capable of significant performance.

However, recent areas of research have not solely focused on strict ranking performance. One area of interest is to increase ranking accuracy on one data set by learning from a different but related data set; this is called transfer learning. Current models train on the two models independently. In this case, potential knowlege from the bigger set is realized. One approach is to transfer knowledge from outdated and out of domain samples [12], another is to learn from unlabeled data [17]. In the context of web-search ranking, the data sets may come from two different but culturally similar countries. The recent Learning to Rank Challenge competition hosted by Yahoo was setup with precisely these two types of data sets. The other area of interest is to increase performance on real issues that effect production of web-search ranking. We look at ways of incorporating costs of the features (or attributes) into the model that learns to rank. Since many features of web-search documents are expensive to acquire (such as PageRank and bounce rate), the goal is to produce a model that is cost-efficient yet accurate. The cost of a decision tree can further be reduced by injecting early exits which weed out the lowest ranked data points efficiently [7].

This thesis provides a survey on techniques that use regression trees for web-search ranking. It will only focus on point-wise algorithms. The survey will provide empirical evidence that shows the performance of point-wise algorithms on two real world web-search ranking data sets, provided by the research teams at Yahoo and Microsoft. Traditional algorithms and novel augmentations to those algorithms will be explored. This thesis will also touch on how point-wise algorithms perform in regards to the new areas of research in the field that was previously mentioned. The goal of this thesis is to show empirical proof of the validity of point-wise learning classifiers relative to each other in terms of ranking accuracy and efficiency in aspects at test time.

1.2 Previous Work

Using point-wise algorithms for web-search ranking dates back to the early 2000's. AltaVista (now integrated into Yahoo) was one of the early pioneers of using gradient boosting in this field. Gradient boosting has since evolved to the other approaches listed, as well as to minimize more complex loss functions [39]. Recently, Burges et. al proposed treating point-wise regression as a multi-class classification problem [21]; they suggest that this framework leads to better ranking results. We ran experiments of our own to validate their claims. Recently, there have been attempts at combining two point-wise technique, Random Forests and Gradient Boosted Regression Trees, for better performance at ranking. One such way is BagBoost, which was one of the winners of the recent Yahoo Learning to Rank Competition[1]. However the implementers note that it takes an unrealistically long time to train its model. Another winner of the Yahoo competition combined the results of Random Forest and GBRT with linear regression [1]. Note that our own Random Forest and GBRT algorithm, IGBRT, and the two mentioned were all developed independently during the same time period.

1.3 Contributions

The main contributions of this thesis are as follows:

- We survey traditional point-wise algorithms and augmentations to them that are applied to ranking; this is done by showing empirical performance data on real world web-search ranking data sets.
- We investigate uses of point-wise algorithms for current topics of interest: transfer learning and feature-cost aware ranking models.

[1]See top competitor's presentation slides at http://learningtorankchallenge.yahoo.com/workshop.php

1.4 Thesis Outline

The thesis is organized as follows:

In the rest of Chapter 1, the notation and setup that will be used throughout the thesis are explained. Decision trees are reviewed and the data sets used for the experiments are introduced. We briefly mention the metrics which are used to analyze the accuracy of the ranking models.

In Chapter 2, two traditional point-wise machine learning techniques are analyzed. The first is Random Forests; we show that this technique can serve as a powerful alternative to the commercially proven algorithm - gradient boosted regression trees. We use Random Forests to explore the best way of handling missing features. Then we explore Gradient Boosting Regression Trees. As this is the most popular technique for web search ranking, it will serve as a base line for the other experiments. We introduce two novel modifications to GBRT, k-Stochastic GBRT and Norm k-Stochastic GBRT. kSGBRT results in great run-time speeds and accuracy. We introduce a novel algorithm: Initialized Gradient Boosted Regression Trees. This technique uses Random Forest as a starting point and boosts to reduce the resultant residuals.

In Chapter 3, it is shown how these traditional regression based techniques can be transformed into a classification problem for even better results. These involve "scoring" the four different classification problems. We show that the scoring method that is equivalent to Expected Relevance gives the best results.

In Chapter 4, the performance of all the techniques on the Yahoo and Microsoft ranking data sets are summarized. They are summarized in one large table so it is easy to see how conclusions were drawn. Our point-wise algorithms are also compared against an open-source pair-wise learner and published LETOR benchmarks.

In Chapter 5, we look at ways point-wise techniques can contribute to transfer learning. We experiment with knowledge transfer by increasing the size of a data set by populating it with candidate data points from another data set. We also add features to the smaller set from a model trained on the larger.

In Chapter 6, cost based decision trees are analyzed. Factoring in the cost of features into the model which learns to rank is a new area of research. We take into account the feature costs by injecting them into the loss function which the decision tree minimizes at each node. We analyze how well traditional point-wise techniques fit this new model in regards to three binary metrics.

In Chapter 7, we review the contributions and implications of this thesis. We indicate in what areas further research would benefit this field of study.

1.5 Notation and Setup

We will treat web-search ranking (WSR) as a plain regression problem: given an input variable, compute a real-valued output. The input to the classifier consists of two parts, a training set and one or more test sets. These sets contain a list of inputs, or data points. The classifier builds a model from the data points in the training set. The purpose of the classifier is to best guess the labels of the data points in the test sets by learning from the training set.

A data point in WSR are query-document pairs. A document x_i is a list of f features (also called attributes). We assume each feature j is described by a real-valued number: $x_i[j] \in \mathcal{R}^f$. It is common to view the individual documents x_i as a vector on some high dimensional space. These documents are matched with a query $q_i \in \mathcal{R}$. In the context of WSR, documents represent a web-site or url, and queries serve as an identifier for some user defined search string. The label y_i indicates some relation of document x_i to the query q_i. The labels are contained in a strict range, $y_i \in \{0, 1, 2, 3, 4\}$. The labels in the context of WSR indicate how relevant the document is to the query. A label of 4 indicates "very relevant" and 0 indicates "not relevant". These labels are typically humanly assigned to the query-document pairs. All together, the input to the classifier are n data points $D = \{(x_1, q_1, y_1), \ldots, (x_n, q_n, y_n)\}$. Note that the labels for the test data are not known, by convention these are assigned to -1.

Point-wise algorithms do not optimize over the queries. That is, we assume all the data points belong to the same query. Then the notation can be simplified to $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$. While the data sets being used do contain multiple

queries, ignoring them does not change any of the algorithms. This works because the relationship between an url and query is integrated into the features. For example, one feature may indicate how often the query string occurs in the url.

1.6 Decision and Regression Trees

The regression trees used in the algorithms are implemented following a modified CART (Classification and Regression Trees) [3] algorithm. It builds a regression tree that minimizes the squared-loss of the labels at each branch (or node) of the tree. We denote D to be the subset of data points on which the classifier h is trained on.

This squared loss of the labels at a node (also referred to as the impurity of a node) in the tree is denoted as:

$$I(D = \{(x_1, y_1), \cdots, (x_n, y_n)\}) = \sum_{i=1}^{n} (y_i - \bar{y})^2 \text{ for } \bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i^2 \tag{1.1}$$

At the start, all of the data is sent to the root node. Each node in the tree partitions the data D in that node into two groups $D_L \subset D$ and $D_R \subset D$ such that $D_L \cup D_R = D$ and $D_L \cap D_R = \emptyset$. The data is partitioned by choosing a feature f and split value v. Given these two values, the partitions are formed:

$$D_L = \{(x_i, y_i) \in D : x_i[f] \leq v\} \tag{1.2}$$

and

$$D_R = \{(x_i, y_i) \in D : x_i[f] > v\} \tag{1.3}$$

The feature f and value v are chosen such that the total loss $I(D_L) + I(D_R)$ is minimized. The algorithm recursively repeats on the two children nodes which contain D_L and D_R until either of two base cases are met: 1) depth of the node reached a terminal limit; 2) cardinality of the labels y_i in D is 1. Computing the split point v for a feature f which minimizes the total squared loss can be computed efficiently with dynamic programming; the pseudocode is in Appendix A.

Each node of the decision tree picks uniformly at random k features to consider splitting on. We will use the terms decision tree and regression tree interchangeably. Our implementation of CART takes three parameters:

1. A set of data points S. A subset $D \subset S$ will be used to build the tree, the rest will be used to get predictions on. Passing all of the data to the tree (rather than only the training set) eliminates the need to keep the trees in memory, which is not neccessary for our evaluation studies.

2. A number k that indicates how many features to consider when making a split.

3. A number d that indicates the maximum depth of the tree; $d = 1$ indicates a tree with only one node, and $d = \infty$ is a full tree.

Given these three parameters, our implementation of Cart considers all classifiers h of depth exactly d and split value k and returns one that approximately minimizes the loss function in equation 1.4.

$$L_h(D) = \sum_{x_i, y_i \in D}^{|D|} (h(x_i) - y_i)^2 \tag{1.4}$$

Our CART trees do not prune after building.

A single decision tree classifier is referred to as $h(\cdot)$. A single decision tree is usually not a powerful enough predictive model. Instead, algorithms that use ensembles of trees tend to generalize much better to the validation and training sets. A classifier that is built of many trees is denoted as $H(\cdot)$.

1.7 Data Sets

We used two real world data sets to evaluate our algorithms. One set of data sets is from the Yahoo Learning to Rank Challenge competition [2]. The other is from

[2] available at http://learningtorankchallenge.yahoo.com.

	Yahoo LTRC		MSLR MQ2007 Folds				
TRAIN	Set 1	Set 2	F1	F2	F3	F4	F5
# Features	700	700	136	136	136	136	136
# Documents	473134	34815	723412	716683	719111	718768	722602
# Queries	19944	1266	6000	6000	6000	6000	6000
Avg # Doc per Query	22.723	26.5	119.569	118.447	118.852	118.795	119.434
% Features Missing	0.68178	0.67399	0.37228	0.37331	0.37263	0.37163	0.37282
TEST	Set 1	Set 2	F1	F2	F3	F4	F5
# Documents	165660	103174	241521	241988	239093	242331	235259
# Queries	6983	3798	2000	2000	2000	2000	2000
Avg # Doc per Query	22.723	26.165	119.761	119.994	118.547	120.167	116.6295
% Features Missing	0.68113	0.67378	0.37368	0.36901	0.37578	0.37204	0.37215

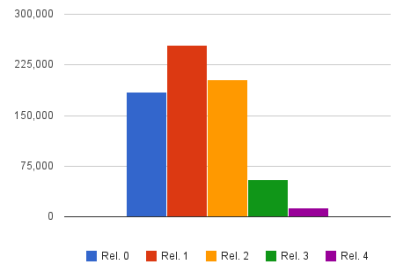
Table 1.1: Statistics of the Yahoo Competition and Microsoft LETOR data sets.

the MQ2007 LETOR data sets, provided by Microsoft [3]. The LETOR data sets are well known in the community and have many public baselines to compare with. The statistics of these two data sets are shown in table 1. The distributions of labels on the data sets are shown in Figure 1.1. Note that the size of the LETOR data sets are much larger than the two Yahoo data sets, and that the majority of documents are irrelevant - we denote irrelevant as having label 0, 1, or 2.

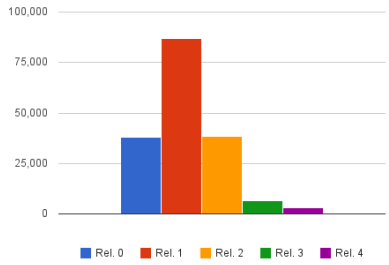
The Yahoo Learning to Rank Challenge (Yahoo LTRC) data comprises of two data sets; Yahoo Set 1 is much larger than Yahoo Set 2. Our implementation of Cart has not been extensively fine tuned and profiled, so running experiments on all of Yahoo Set 1 and the LETOR sets take a prohibitively long amount of time. To accommodate for this, some of the illustrative experiments runs ran on either Yahoo Set 2 or a subset of Set 1 that is of the same size of Set 2. It will be indicated when the subset of Set 1 was used. The Microsoft data comprises of five folds on a single data set. Each of the five folds, and both yahoo data sets, are partitioned into three splits: a train split, validation split, and test split.

The Yahoo data sets were used in a 2010 web search ranking competition. This competition helps us show that point-wise algorithms can prove competitive against more complicated pairwise and list-wise algorithms. The two data sets in the competition represent data from different countries. This setup is good for running experiments on transfer learning.

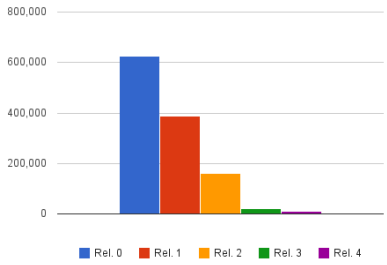
[3]available at http://research.microsoft.com/en-us/um/beijing/projects/letor/.



(a) Yahoo Set 1



(b) Yahoo Set 2



(c) Each Microsoft LETOR fold

Figure 1.1: Grade distribution on data sets.

1.8 Performance Metrics

There are three metrics we will judge our algorithms by: Root Mean Squared Error (RMSE), Normalized Discounted Cumulative Gain (NDCG) [19], and Expected Reciprocal Rank (ERR) [11]. Suppose the true labels of each data point x_i is y_i, and the predicted label by a classifier is $h(x_i)$.

RMSE is the simplest metric. We use it because it is a very common and well understood measurement of error. In our case it evaluates to

$$RMSE = \frac{1}{n}\sqrt{\sum_{i=1}^{n}(h(x_i) - y_i)^2}. \tag{1.5}$$

RMSE encapsulates how far off on average a prediction is; lower RMSE scores indicate a better learner.

NDCG and ERR take into account the ranking or relative ordering of the documents rather than the classifier's strict output value. The top documents contribute more to the score; in most ranking contexts, correctly ranking the top ten documents is much more important than correctly ranking the last ten. We denote the ordering of documents x_i from classifier h by π_h; if a classifier h ranks the fourth document as the second most relevant, then $\pi_h(2) = 4$. For point-wise algorithms, the ranking is calculated by simply sorting the documents x_i by $h(x_i)$ in descending order. The documents grouped within the same query are ranked and measured by NDCG and ERR; the final value is the averaged query score. NDCG and ERR encapsulate how well a classifier's ranking is to the true or best ranking; higher NDCG and ERR scores indicate a better learner.

The NDCG of a query is defined as

$$NDCG_\pi = \frac{1}{Z} \sum_{i=1}^{m} \frac{2^{y_{\pi(i)}} - 1}{log(1+i)} \tag{1.6}$$

with Z being the normalizing factor - the score of the best ordering. We use the NDCG@10, which only scores the top ten documents (i.e $m = 10$). If a query has no relevant documents, it is by default given a NDCG score of $\frac{1}{2}$.

The Expected Reciprocal Rank for the ordering of a query with m documents is defined as

$$ERR_\pi = \sum_{i=1}^{m} \frac{1}{i} R(y_{\pi(i)}) \prod_{j=1}^{i-1} (1 - R(y_{\pi(j)})) \tag{1.7}$$

with $R(y) = \frac{2^y - 1}{16}$. ERR models the expected payoff: the probability that a web-surfer is satisfied with document j is probability s/he is not satisfied with documents $1, 2...j-1$ and satsified with document $j = R(y_\pi(j))$. As such, summing up the discounted payoff of each of the m documents then averaging calculates the expected payoff. ERR applies a stronger discount factor to documents lower on the ranking list, so it is not necessary to insert in a cutoff point like with NDCG.

Chapter 2

Traditional Point-wise Algorithms

There are two point-wise algorithms that have proven successful particularly in the field of web-search ranking: Random Forests [5] and Gradient Boosted Regression Trees [16]. In fact, all of the top placed teams in the recent Yahoo Learning to Rank Competition used variants of either of these two algorithms with Decision Trees[4]. Despite the fact that Random Forests have been used by some competitors in the competition, there have not been many research papers using Random Forests for Learning to Rank. We will analyze how Random Forests compares against Gradient Boosting, and show augmentations to the GBRT algorithm that increases performance. Then we introduce an algorithm that combines Random Forests and Gradient Boosted Regression Trees.

2.1 Random Forest

Random Forests (RF) combine two ideas to make a strong classifier: bagging [4] and sampling features.

Bagging (ie Bootstrap Aggregating) averages together many different decision trees. The average of a set of decision trees is a set of predictions, where each prediction is the average predicted value from each of the decision trees. When bagging, each of the decision tree gets a randomly chosen list of $n = |D|$ data points to train on. These data points are sampled uniformly from D with replacement, so each classifier is slightly different as it is trained on a sample of a common data set. Doing this

[4]see competition proceedings at http://learningtorankchallenge.yahoo.com/workshop.php.

Algorithm 1 Random Forests

1: Input: data set $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$
Parameters: Sample amt $0 < K \leq f$, Number of trees M
2: **for** $t = 1$ to M **do**
3: $D_t \subseteq D$ # Sample with replacement, $|D_t| = |D|$.
4: $h_t = \text{Cart}(D_t, K, \infty)$ # Build full ($d = \infty$) Cart with $K \leq f$ randomly chosen features at each split.
5: **end for**
6: $h(\cdot) = \frac{1}{M} \sum_{t=1}^{M} h_t(\cdot)$.

reduces variance and helps against the problem of over-fitting to the training set. The more trees that are averaged, the better the final classifier will be. After a certain point, averaging more trees will not significantly increase accuracy since the predictions will converge to some value.

Sampling features adds an additional controlled layer of variance to bagging. At each node of the decision tree, only k uniformly randomly features are chosen to consider splitting on. Each decision tree is built to full (i.e. without a depth limit). The algorithm for Random Forests is shown in Algorithm 1. Note that the construction of each regression tree is independent of the others, making this algorithm easily parallelized; the algorithm scales very well as the number of processors increases.

For RF we simply set $k = 0.1f$. Performing cross validation on the Yahoo data sets yielded optimal k values close to this rule of thumb.

2.2 Treating Missing Features

Table 1.1 Shows that web-search ranking data sets are filled with missing features. There could be several reasons why the value of a feature may be missing: the feature has been computed yet, it is not applicable for the document, computations timed out, became outdated, etc. There is not much literature on how is best to handle these missing features. We explore different ways to handle these missing features: the missing features can either be ignored, or a value can be guessed for the missing feature. We explore six different approaches:

1. Fill in missing features with 0 (or $-\infty$).

2. Fill in missing features with 1 (or $+\infty$).

3. Fill in missing features with the mean value of that feature .

4. Fill in missing features with the median value of that feature.

5. Iteratively predict the missing features.

6. Ignore the missing feature and make a 3-way split in the node.

The values of the features on the Yahoo and LETOR data sets have been normalized to be between 0 and 1, exclusive. Thus setting a feature to 0 effectively always falls into the left subtree. Similarly setting it to 1 always falls into the right subtree. The mean and median values are calculated for each feature that has a value. That calculated value is substituted in for each missing feature.

We can also build a model to predict the missing values. A model is built from the known features, to predict a particular missing feature. All the features are predicted with this model. Once this is done, the predicted features can be integrated into the model. This new model then predicts the original missing features, and the predictions are again integrated in to a new model. This process repeats until convergence; the predictions for the features do not changed significantly during consecutive iterations. The pseudo-code for applying this technique is shown in in Algorithm 2.

Recall that decision trees typically have two splits at a node (D_L for data points $x_i[f] \leq v$, and D_R for data points $x_i[f] > v$). We experiment with adding another split to make a three-way split; the additional node contains data points D_M whose value is missing for the chosen feature. We denote missing features as having the value -1:

$$D_M = \{(x_i, y_i) \in D : x_i[f] = -1\} \tag{2.1}$$

The results on the two Yahoo data sets are summarized in Table 2.1. The table shows that splitting three ways or predicting the missing features does not work well for web-search data. Predictions were performed separately on the training and validation sets on Yahoo Set 2; it was not run on the test set of Set 2 or any of Set 1 because the data sets were large enough that the algorithm took a prohibitively long to converge, given our implementation of decision trees. The table also shows that substituting 0

Algorithm 2 Predict Missing Features

Input: data set $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$
Parameters: Sample amt $0 < K \leq f$, Number of trees M
Require: Denote document i is missing feature F by $x_i[f] = -1$
$M \leftarrow \{(i, f) : x_i[f] = -1 \forall i, f\}$ # record which documents x_i has feature f missing for all features
repeat
  for $F = 1$ to f **do**
    for $i = 1$ to n **do**
      $\hat{x}_i \leftarrow \{j | x_i[j] \neq -1\}$ # record which features aren't missing for document i
    end for
    $c \leftarrow \hat{x}_0 \cap \hat{x}_1 \cap ... \cap \hat{x}_n$ # get set of features that every document has in common
    for $i = 1$ to n **do**
      $\bar{x}_i \leftarrow [x_i[k] | k \in c]$ # reduce feature vector to only known features
    end for
    $H(\cdot) \leftarrow RF(\{(\bar{x}_i, x_i[F]) \forall i | x_i[F] \neq -1\}, k, M, \infty)$ # Build model on all known features where label is value of feature F
    for $i = 1$ to n **do**
      if $(i, F) \in M$ **then**
        $x_i[F] \leftarrow H(x_i)$ # Use model to predict feature F on documents with x[F] originally missing
      end if
    end for
  end for
until convergence # quit when H does not change during subsequent iterations

for missing values tends to yield the best performance. Experiments on the Microsoft LETOR data sets gave similar results. Based on this conclusion, substituting missing values for 0 is the approach we have taken for the remainder of the experiments.

2.3 Gradient Boosted Regression Trees

Gradient Boosted Regression Trees (GBRT) is another Machine Learning technique that has proven to be very powerful in the field of web-search ranking. All of the top competitors in the Yahoo Learning to Rank Challenge competition used some variation of GBRT . Similar to Random Forests, GBRT builds a classifier based on many decision trees. But unlike RF which can build many full trees ($d = \infty$) in parallel, GBRT builds many small trees incrementally with high bias. These weak learners are optimized to iteratively reduce the training rmse error. The classifier is not the average prediction as with Random Forests; instead it is the total (ie sum) prediction, while taking into account a learning rate.

Y! Set 1	Valid			Test		
approach	RMSE	NDCG	ERR	RMSE	NDCG	ERR
0	**0.76236**	0.77136	**0.4575**	**0.74791**	**0.77733**	**0.46304**
1	0.76311	**0.77147**	0.4574	0.74979	0.77643	0.46247
mean	0.77634	0.769	0.45553	0.78517	0.77026	0.45989
median	0.7853	0.76638	0.45522	0.79631	0.76899	0.45955
3-way split	0.76433	0.77005	0.45694	0.75089	0.77515	0.46241
Y! Set 2	Valid			Test		
0	**0.62621**	**0.77620**	0.45062	**0.63529**	**0.77447**	0.46194
1	0.62643	0.77494	**0.45087**	0.63612	0.77232	0.46187
mean	0.63039	0.77408	0.45062	0.64761	0.77386	**0.46207**
median	0.63933	0.77076	0.45020	0.65247	0.77207	0.46198
3-way split	0.63162	0.76966	0.44958	0.64066	0.76755	0.45911
predict	0.65470	0.75560	0.44497	0.76598	0.70330	0.43697

Table 2.1: Comparison of various methods to treat missing values on the Yahoo data sets.

We denote the boosted classifier by T which is comprised of many weak learners h_i. Suppose we have a loss function $L(T(x_1)...T(x_n))$ which reaches its minimum if $T(x_i) = y_i$ for all x_i. Each successive tree h_{i+1} is built to reduce the regression error of the previous trees $T = \sum_{i=1}^{|T|} \alpha h_i$.

We use the loss function:

$$L = L(T(x_1)...T(x_n)) = \frac{1}{2} \sum_{x_i, y_i \in T}^{|T|} (T(x_i) - y_i)^2 \tag{2.2}$$

GBRT updates the instance space $x_1...x_n$ with an approximated gradient step in equation 2.3. Here α denotes the learning rate.

$$T(x_i) \leftarrow T(x_i) - \alpha \frac{\partial L}{\partial T(x_i)} \tag{2.3}$$

Let the gradient $\frac{\partial L}{\partial T(x_i)}$ by denoted as r_i. To take the gradient step, the weak learner h added to T must be the tree that minimizes:

$$h_t \approx \underset{h}{\operatorname{argmin}} \sum_{i=1}^{n} (h(x_i) - r_i)^2 \tag{2.4}$$

Algorithm 3 Gradient Boosted Regression Trees (Squared Loss)

Input: data set $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, Parameters: α, M_B, d
Initialization: $r_i = y_i$ for $i = 1$ to n
for $t = 1$ to M_B **do**
  $T_t \leftarrow \text{Cart}((x_1, r_1), \ldots, (x_n, r_n), f, d)$ #Build Cart of depth d, with all f features and targets $\{r_i\}$
  for $i = 1$ to n **do**
    $r_i \leftarrow r_i - \alpha T_t(x_i)$ #Update the residual of each sample x_i.
  end for
end for
$H(\cdot) = \alpha \sum_{t=1}^{M_B} T_t(\cdot)$. #Combine the Regression Trees $T_1, \ldots, T_M$.

Our implementation of GBRT uses the Cart algorithm to find the regression tree h_t in each iteration. The pseudocode for the GBRT is described in Algorithm 3. In essence, each regression tree is build to model the residuals of the classifier, which is $r_i = y_i - T(x_i)$. The GBRT algorithm has three parameters:

1. Depth d to control the strength of the weak learners.
2. Learning rate α which controls the magnitude of each gradient step.
3. Iterations or trees M_B which dictates how many gradient steps to perform.

We used trees with depth $d = 4$ (8 terminal nodes), which has been shown to work well for boosting [29]. The number of iterations (or trees in the classifier) M_B and learning rate α are picked using cross validation. After a certain amount of boosting iterations, the classifier starts to over-fit to the training data or accuracy converges for the testing set. For the learning rate, smaller values of α tend to provide better accuracy but will require more trees. We found $M_B = 1000$ to be a good amount for Yahoo Set 2.

Figure 2.1 shows how Random Forests compares to Gradient Boosted Regression Trees with various learning rates. The bold black line represents Random Forests. For RF we chose the parameters $M_F = 10000$ and $K = 70$. Regardless of the learning rate, Random Forests always performed better than GBRT in regards to each metric. We do not expect lower learning rates to show significantly different behavior, and larger learning rates would just over-fit quicker. Note that because of its highly parallel nature, Random Forests can be computed much faster than GBRT (as shown

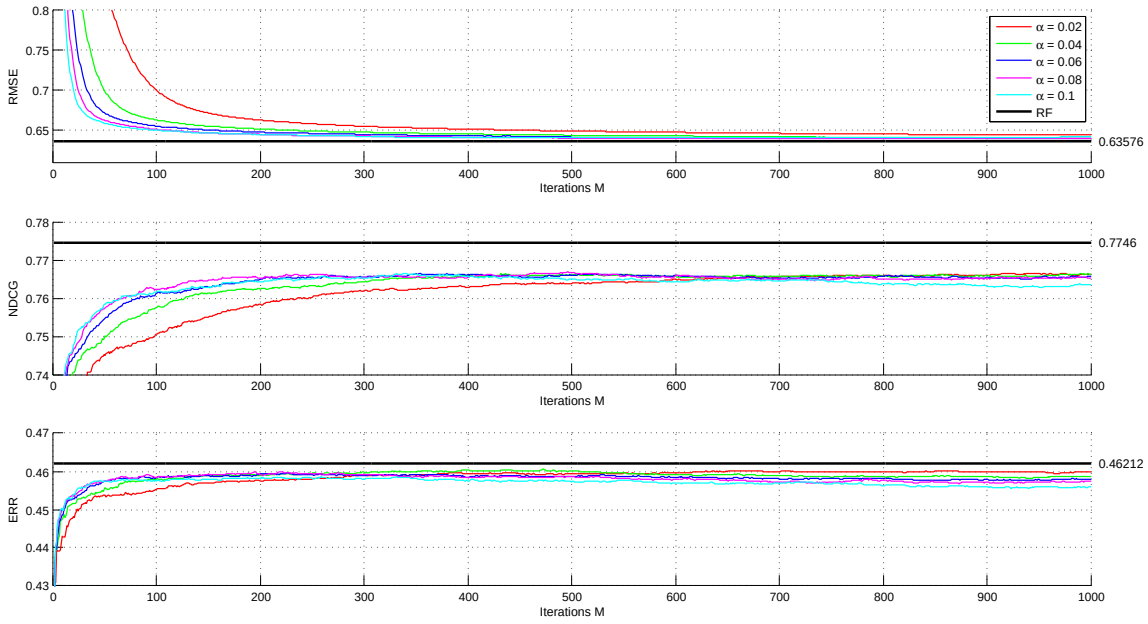


Figure 2.1: Results of GBRT compared to Random Forests on the Yahoo Set 2.

in Table 4). This figure helps show that Random Forest can be a strong alternative to GBRT.

2.3.1 Stochastic GBRT

Soon after Gradient Boosted Decision Trees was published, the inventor published a minor modification to the algorithm which reportedly offered a significant accuracy increase. This algorithm is called Stocastic Gradient Boosted Decision Trees [**?**] (SGBDT/SGBRT). Similiar to Random Forests, SGBRT incorporates randomization as an integral part to the classifier. With GBRT, each classifier h is trained on a the full training set D. In SGBRT, each classifier h is trained on a random subsample of the training data, $D_s \subset D$. The data is subsampled without replacement. We denote the sample rate as s; $s = 0.2$ indicates each tree in SGBRT sampled 20% from the training set.

Training on a subset of the data obviously leads to training-time boost, but it has been shown that it also increases accuracy. Friedman indicated that a sample rate

$s = 0.40$ had the best results on the data sets tested. Table **??** shows how SGBRT performed on Yahoo Set 2 with various sampling rates. The table shows that $s = .50$ performed the best on the test set, but lower sampling rates still had competitive results. In fact, $s = 0.20$ performed better than GBRT ($s = 1.00$).

	Yahoo LTRC Set 2 Val			Yahoo LTRC Set 2 Test		
S	RMSE	ERR	NDCG	RMSE	ERR	NDCG
0.10	0.63237	0.45054	0.76886	0.64067	0.45856	0.76602
0.20	**0.63082**	**0.45109**	0.77042	0.64085	**0.45908**	0.76525
0.30	0.63130	0.44962	0.76857	0.63998	0.45800	0.76562
0.40	0.63085	0.45104	0.77011	0.64035	0.45791	0.76640
0.50	0.63149	0.45022	0.77010	**0.63971**	0.45732	**0.76660**
0.60	0.63499	0.44943	0.76761	0.64490	0.45742	0.76223
0.70	0.63455	0.44981	0.76813	0.64506	0.45639	0.76153
0.80	0.63533	0.44796	0.76670	0.64408	0.45681	0.76230
0.90	0.63296	0.44909	0.76771	0.64370	0.45717	0.76355
1.00	0.63202	0.45042	**0.77188**	0.64068	0.45670	0.76495

Table 2.2: Performance of SGBRT on Yahoo Set 2.

2.3.2 k-Stochastic GBRT

Here we introduce k-Stochastic Gradient Boosted Regression Trees (kSGBRT), a novel approach to Gradient Boosting. Traditionally, boosted classifiers evaluate all f features to split on, and chooses the feature that reduces a loss function; usually entropy for decision trees [26], and squared loss for regression trees. With kSGBRT, each weak classifier only evaluates k randomly chosen features at each split point. There are two intuitive advantages for not evaluating every feature:

1. Faster training time.
2. Helps overcome myopic nature of CART algorithm.

The majority of of the computations involved for constructing a decision trees lies in finding the best feature and value to split each node on. Certainly if less features are considered, it takes less time to build the tree. The relation between the running time for various k values are shown in Table 2.3. Another aspect kSGBRT improves

on is myopia. Due to relations of attributes (or features), the best way to split on a node for the most information gain may depend on more than one feature. Since most decision tree algorithms only consider splitting on one attribute at a time, the algorithm is myopic if there are such interactions between the attributes [20]. Only considering a subset of features at each split can help utilize any intrinsic relations between attributes. An example for which greedy splits results in a sub-optimal tree is the XOR setup.

These incentives would not be worth it if kSGBRT significantly reduced accuracy. Figure 2.2 shows how kSGBRT with various low k values compares against GBRT with a strong step size. The black line represents GBRT, ie $k = f = 700$; the other lines have relatively low k values. The lowest k value $k = 20$ which considers $\approx 3\%$ of the features, lags behind GBRT only in regards to NDCG. Yet $k = 20$ and all the other SGBRT configurations performs well in regards to ERR.

These plots also show that k-stochastic boosting helps prevent against over-fitting. Notice that after $M = 500$, GBRT starts to over-fit in regards to NDCG and ERR. As the accuracy for GBRT goes down, the accuracy for $k \geq 40$ remains relatively constant. This is possibly explained by the myopic nature of GBRT.

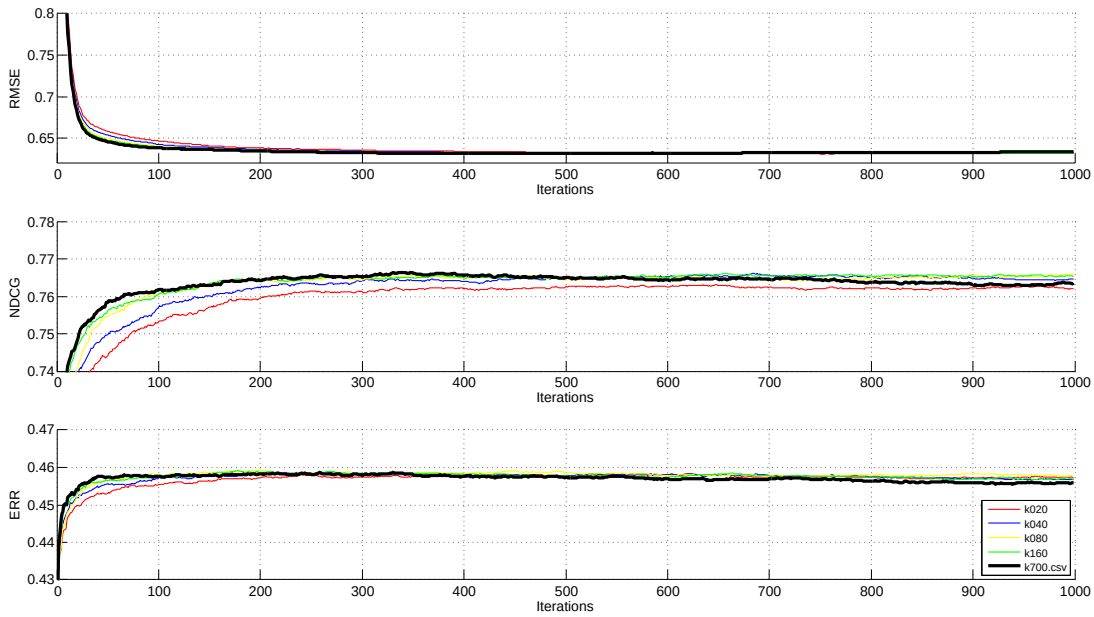


Figure 2.2: Results of kSGBRT on Yahoo Set 2 with various k values.

Table 2.3 compares the running time and accuracy of kSGBRT to GBRT. The runs only used one processor. The accuracy numbers reported on the test set are picked in correlation to the highest validation score for each metric. kSGBRT with $K = 100$, which uses approximately 14% of the features, had the best results on the test set. It is surprising that GBRT (i.e. the run that used all the features, $k = 700$) did not outperform kSGBRT in regards to any of the metrics on the test set for ERR, even when GBRT performed best on validation. This shows that GBRT and kSGBRT perform on par with one another.

Not only does kSGBRT provide good results compared to traditional GBRT, it also takes much less time to train. However, the training time does not scale proportionally to k. We attribute this to amount of I/O the boosting algorithm requires.

	Run-Time	Yahoo LTRC Set 2 Val			Yahoo LTRC Set 2 Test		
K	(mins)	RMSE	ERR	NDCG	RMSE	ERR	NDCG
10	65	0.63304	0.45072	0.76926	0.64289	0.45841	0.76280
20	76	0.63159	0.44860	0.76835	0.64168	0.45754	0.76249
40	97	0.63229	0.45051	0.76838	0.64139	0.45718	0.76488
60	117	0.63105	0.44913	0.76892	0.64139	0.45868	0.76624
80	131	0.63161	0.44878	0.76915	0.64101	0.45857	0.76461
100	153	0.63124	0.45050	0.76903	**0.63987**	**0.45933**	0.76501
120	173	0.63066	0.45050	0.77082	0.64077	0.45719	0.76413
140	187	**0.62968**	0.45110	0.77122	0.63926	0.45914	**0.76717**
160	206	0.63159	**0.45146**	0.76985	0.64082	0.45793	0.76540
700	456	0.63202	0.45042	**0.77188**	0.64068	0.45670	0.76495

Table 2.3: Run-Times and Performance of kSGBRT using one processor.

2.3.3 Averaged k-Stochastic GBRT

Here we discuss averaging or several runs of k-stochastic boosted classifiers. Remember that Random Forests function on the ideas of bagging and feature sampling. Sampling the features serves to control the variation of each decision tree. We can interpret the k value for kSGBRT to serve the same purpose. Bagging does not fit into the context of kSGBRT as the training data is not sampled. However, we can still average together many runs with the same k value. We denote the number of runs to be averaged as b. The weak law of large numbers says that this classifier

		Run-Time	Yahoo LTRC Set 2 Val			Yahoo LTRC Set 2 Test		
K	B	(mins)	RMSE	ERR	NDCG	RMSE	ERR	NDCG
10	1	61	0.63503	0.44884	0.76693	0.64558	0.45729	0.76237
10	2	96	0.63142	0.44902	0.76791	0.64015	0.45892	0.76519
10	5	198	0.63026	0.45062	0.77049	0.63915	0.45971	0.76664
10	10	359	0.62977	0.45183	0.77081	0.63850	0.46043	0.76795
10	20	677	0.62930	0.45151	0.77207	0.63826	0.46052	0.76856
10	40	1326	0.62924	0.45170	0.77279	0.63806	0.46014	0.76818
10	80	2546	0.63095	0.45051	0.77078	0.64044	0.46032	0.76682
10	100	3535	0.63217	0.44936	0.76969	0.64182	0.46063	0.76653
20	1	67	0.63151	0.44917	0.76852	0.64120	0.45851	0.76469
20	2	119	0.63086	0.45091	0.77013	0.63953	0.45963	0.76612
20	5	241	0.62927	0.45098	0.77135	0.63804	0.46001	0.76823
20	10	438	0.62875	0.45091	0.77262	0.63754	0.46005	0.76758
20	20	828	0.62863	0.45157	0.77253	0.63728	0.46043	0.76843
20	40	1565	0.62847	0.45188	0.77296	0.63743	0.46023	0.76793
20	80	3135	0.63029	0.45010	0.77122	0.64011	**0.46096**	0.76669
20	100	3839	0.63131	0.44956	0.77022	0.64120	0.46054	0.76580
40	1	96	0.63172	0.44918	0.76694	0.64112	0.45747	0.76562
40	2	158	0.62961	0.45116	0.77147	0.63914	0.45872	0.76725
40	5	326	0.62809	0.45195	0.77240	0.63738	0.45935	0.76852
40	10	592	0.62837	0.45191	0.77175	0.63670	0.45983	0.76843
40	20	1105	**0.62804**	**0.45228**	**0.77308**	**0.63675**	0.45960	**0.76907**
40	40	2141	0.62851	0.45135	0.77192	0.63779	0.45984	0.76837
40	80	4234	0.63108	0.44966	0.77056	0.64112	0.46071	0.76731
40	100	5623	0.63219	0.44942	0.77018	0.64239	0.46048	0.76668
700			0.63202	0.45042	0.77188	0.64068	0.45670	0.76495

Table 2.4: Avg kSGBRT with various k and b.

that is the average of b classifiers with small variations will converge to the expected value as $b \to \infty$. We call this classifier that averages many runs of kSGBRT as Avg kSGBRT.

Table 2.4 summarizes the run-times and accuracy for Norm kSGBRT on Yahoo Set 2. The configurations of the runs were $b = \{1, 2, 5, 10, 20, 40, 80, 100\}$ and $k = \{10, 20, 40\}$. The run-times reported are using one processor. We observed that averaging 5 or more times $b \geq 5$ tends to have significantly better performance than $b < 5$. The table shows that $k = 40$, $b = 20$ performed the best; and that setting outperformed GBRT and kSGBRT.

2.4 Initialized Gradient Boosted Regression Trees

We propose another novel augmentation to GBRT, called Initialized Gradient Boosted Regression Trees (IGBRT). The traditional GBRT algorithm does not initialize the initial classifier to any useful value: $T(x_i) = 0$. Since our loss function from 2.2 is concave, the gradient steps will step towards the global minimum, regardless of what each $T(x_i)$ starts off as. However, our gradient step procedure only approximates the gradient step. A true gradient step entails a very small learning rate, which requires an impractical amount of iterations or trees to reach the convergent global minimum. For a "smart" starting point, IGBRT initializes the GBRT classifier with the results of Random Forests. There are a couple of reasons why Random Forests would be a good starting point:

1. RF has been shown to be an accurate classifier.
2. RF parallelizes well and as such does not require much time to train, given adequate resources.
3. RF is resistant against over-fitting; increasing the iterations M will not lower performance.
4. RF in a sense only has one true parameter (k), which can be set using a rule of thumb. As such RF does not require any parameter tuning.

The only change to the GBRT algorithm is what the initial boosted classifier predicts $T(x_i)$. We set each $T(x_i)$ to be the prediction of Random Forests for the data point x_i. We let $k = 10\%f$, which is $k = 70$ for the Yahoo data sets. The pseudocode for IGBRT is shown in Algorithm 4.

Figure 2.3 shows how IGBRT (initialized with RF,$M_F = 10000$) compares to Random Forests on Yahoo Set 2. The bold black line represents Random Forests. Remember that on this data set, Random Forests have been shown to outperform all settings of GBRT. The various traces are IGBRT initialized with the Random Forest output. The figure shows that IGBRT with any configuration outperforms Random Forests in regards to RMSE, and NDCG. For ERR, there is a brief range for $\alpha = 0.1$ where RF performs better; this may indicate that the learning rate is too strong for the ERR

Algorithm 4 Initialized Gradient Boosted Regression Trees (Squared Loss)

Input: data set $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, Parameters: α, M, d, K_{RF}, M_{RF}
$F \leftarrow RandomForests(D, K_{RF}, M_{RF})$
Initialization: $r_i = y_i - F(x_i)$ for $i = 1$ to n
for $t = 1$ to M **do**
  $T_t \leftarrow$ Cart($\{(x_1, r_1), \ldots, (x_n, r_n)\}, f, d$) #Build Cart of depth d, with all f features, and targets $\{r_i\}$
  for $i = 1$ to n **do**
    $r_i \leftarrow r_i - \alpha T_t(x_i)$ #Update the residual of each sample x_i.
  end for
end for
$H(\cdot) = F(\cdot) + \alpha \sum_{t=1}^{T} T_t(\cdot)$. #Combine the Regression Trees $T_1, \ldots, T_M$ with the Random Forests F.

metric. However outside that range for $\alpha = 0.1$, and any range for NDCG, IGBRT beats RF.

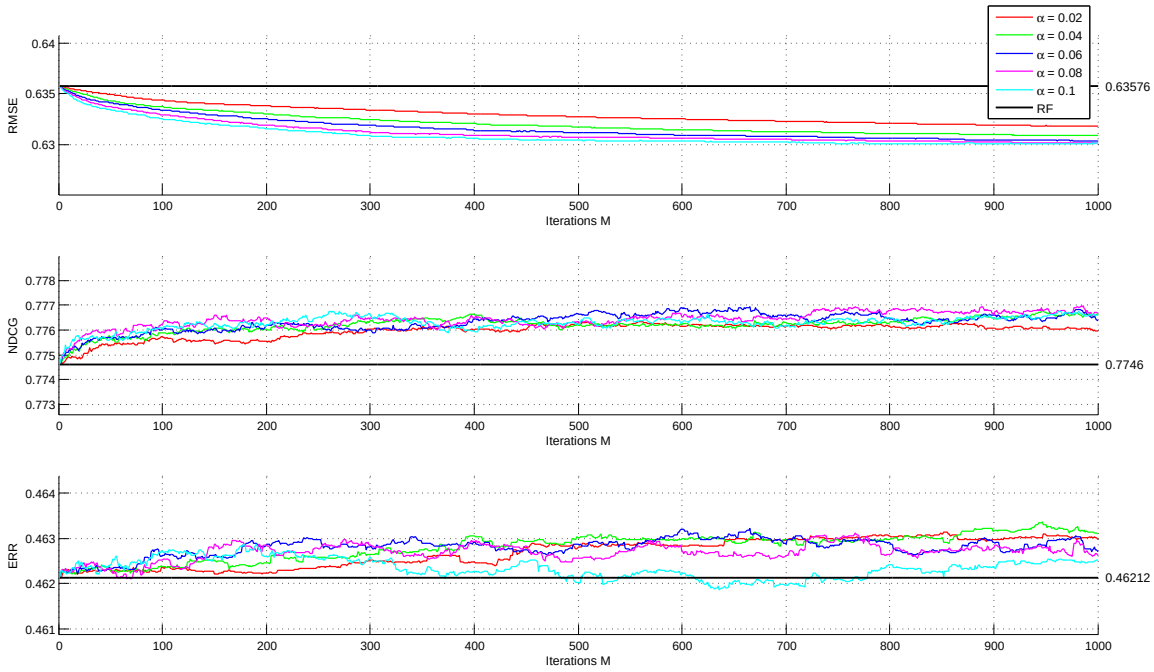


Figure 2.3: Results of IGBRT on Yahoo Set 2 with various learning rates.

Chapter 3

Classification Paradigm for Regression Algorithms

The algorithms thus far have used regression to calculate the relevance of a document. Recently is has been shown that approximating the relevance of a document using classification can lead to even better performance [21]. We follow their procedures to see if we can produce similar results on the Yahoo Challenge and Microsoft LETOR data sets. Note that it has been proven that both the ERR error [24] and NDCG error [21] is bounded by the classification error. Thus the classification error (number of misclassified documents) can be used as a surrogate loss function to minimize.

3.1 Description

Remember that in the web-search ranking setup, labels have relevances $y_i \in \{0, 1, 2, 3, 4\}$. The algorithms so far have learned a regressor to approximate the relevance: $T(x_i) \approx y_i$. In a classification setting, we wish to learn the probability that a document x_i is of a certain class, $P(y_i = c)$. We denote this as $P_c(x_i)$, for $c \in \{0, 1, 2, 3, 4\}$. Given these 5 probabilities, a final score S_i can be describe the relative relevance of the document:

$$S_i = \sum_{c=0}^{5} P_c(x_i) * F(c) \tag{3.1}$$

with $F(c)$ being a function that weights relevance c. Using $F(c) = c$ becomes Expected Relevance. Since ERR and NDCG only evaluate the ordering of the data points, any monotone increasing function of $F(c)$ can be used.

Algorithm 5 Gradient Boosted Regression Trees Using Classification

Input: data set $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, Parameters: α, M, d, W
Initialization: $r_{i,c} = (y_i \leq c)$ for $i = 1$ to n, $c = 0$ to 4
for $t = 1$ to M **do**
  for $c = 0$ to 4 **do**
    $T_{t,c} \leftarrow \text{Cart}((x_1, r_{1,c}), \ldots, (x_n, r_{n,c}), f, d)$ #Build Cart of depth d, with all f features and targets $\{r_i\}$
    for $i = 1$ to n **do**
      $r_{i,c} \leftarrow r_{i,c} - \alpha T_{t,c}(x_i)$ #Update the residual of each sample x_i.
    end for
  end for
end for
$P_c(\cdot) = \alpha \sum_{t=1}^{T} T_{t,c}(\cdot)$. # $P_c(\cdot)$ models if the relevance of a doc. is less than c
$H(\cdot) = \alpha \sum_{c=0}^{4} W[c] * (P_c(\cdot) - P_{c-1}(\cdot))$. # Subtract the cumulative probabilities to get actual probabilities, multiply them by a weight vector W

To calculate each $P_c(x_i)$ value, the inventors of this technique take advantage of the natural ordering of the relevance labels and generate four binary classification problems. Each binary problem c predicts whether or not the relevance of document x_i is less than or equal to c. We denote a classifier which models this cumulative probability $P(y_i \leq c)$ as $T_c(\cdot)$. This effectively makes four binary classification problems, where the labels have been transformed to 0 or 1; the output of the classifier can be interpreted as the probability the label is 1. Given these four cumulative probabilities, the original probabilities $P_c(x_i)$ can be computed:

$$Prob(y_i = c) = P_c(x_i) = Prob(y_i \leq c) - Prob(y_i \leq c - 1) \tag{3.2}$$

$$P_c(x_i) = T_c(x_i) - T_{c-1}(x_i) \tag{3.3}$$

with $T_4(\cdot) = 1$ and $T_{-1}(\cdot) = 0$. Algorithm 5 shows the pseudocode for gradient boosting with classification. The main change is that four classifiers need to be trained, each with the labels $r_{i,c} = y_i \leq c$. Given the predictions from these classifiers which represent the cumulative probabilities, the actual probabilities can be extracted. Then the final prediction is a weighting of the actual probabilities.

3.2 Scoring Functions

We explored three fixed scoring functions:

1. $F(c) = c$ (Expected Relevance)
2. $F(c) = c^2$
3. $F(c) = 2^c$

These three scoring functions are compared in Table 3.1. The classifier used was Random Forests with $k = 70$ and $M_F = 1000$. Intuitively, one may think the scoring methods 2^c and c^2 would work better than the Expected Relevance as they place higher importance on labels with high relevance, and NDCG and ERR are effected the most by the top relevant documents. However, the Expected Relevance scoring scheme worked the best on every metric on both data sets. Note that since the other scoring methods distort the predicted label in favor of widening the relevance gaps, the RMSE (ie classification error) metric is not comparable across these scoring functions.

Scoring	Yahoo LTRC Set 2 Val			Yahoo LTRC Set 2 Test		
Function	RMSE	ERR	NDCG	RMSE	ERR	NDCG
$F(c) = c$	0.62900	**0.45245**	**0.77631**	0.63736	**0.46113**	**0.77157**
$F(c) = c^2$	1.67218	0.44724	0.76118	1.71985	0.45884	0.75979
$F(c) = 2^c$	2.00306	0.44620	0.76131	2.03567	0.45884	0.76069
Scoring	Yahoo LTRC Set 1 Val			Yahoo LTRC Set 1 Test		
Function	RMSE	ERR	NDCG	RMSE	ERR	NDCG
$F(c) = c$	0.76318	**0.45679**	**0.77175**	0.74884	**0.46280**	**0.77705**
$F(c) = c^2$	2.03004	0.45639	0.76672	2.04136	0.46219	0.77204
$F(c) = 2^c$	2.28912	0.45633	0.76594	2.29483	0.46174	0.77178

Table 3.1: Comparing various scoring functions for classification.

Note that output of $F(c) = c$ is just the product of two row vectors, one of them having fixed values and the other representing the 5 cumulative probabilities. When Expected Relevance is used, the scoring function S_i becomes

$$S_i = \sum_{c=0}^{4} c * P_c(x_i) \tag{3.4}$$

This can be viewed as multiplying the vector containing the cumulative probabilities with certain weights:

$$
\begin{aligned}
S_i &= \sum_{c=0}^{4} c * P_c(x_i) \\
&= \sum_{c=0}^{4} c * (T_c(x_i) - T_{c-1}(x_i)) \\
&= 0(T_0(x_i) - 0) + 1(T_1(x_i) - T_0(x_i)) + 2(T_2(x_i) - T_1(x_i)) + 3(T_3(x_i) - T_2(x_i)) + \\
&\quad 4(T_4(x_i) - T_3(x_i)) \\
&= -1T_0(x_i) + -1T_1(x_i) + -1T_2(x_i) + -1T_3(x_i) + 4
\end{aligned}
$$

Thus the score for expected relevance can be viewed as the product of two vectors:

$$S_i = \mathbf{w}^{\mathrm{T}}\mathbf{h}_i \tag{3.5}$$

with

$$\mathbf{w}^{\mathrm{T}} = \begin{bmatrix} -1 & -1 & -1 & -1 & 4 \end{bmatrix} \tag{3.6}$$

and

$$\mathbf{h}_i^{\mathrm{T}} = \begin{bmatrix} T_0(x_i) & T_1(x_i) & T_2(x_i) & T_3(x_i) & 1 \end{bmatrix} \tag{3.7}$$

Given this framework, the weight vector $\mathbf{w}$ that minimizes the squared error from a training set

$$Error = \sum_{i=1}^{n} (\mathbf{w}^{\mathrm{T}}\mathbf{h}_i - y_i)^2 \tag{3.8}$$

can be calculated simply by performing Ordinary Least Squares. Ordinary Least Squares solves for the unknown parameters in a linear regression model by reducing the squared error. We solved for this weight vector using the data points in the training and validations sets on the Yahoo data sets with Random Forests ($k = 70, M_F = 1000$); they are shown in Table 3.2. Surprisingly, the derived weight vectors which minimize the classification error on the training and validation sets yield very similar weights to Expected Relevance.

Yahoo Set	input	OLS derived weight vector
1	Training	[-1.1660 -1.2503 -1.1383 -1.0671 4.4104]
1	Validation	[-1.1701 -0.9857 -1.0417 -1.0594 4.1269]
1	Training+Val	[-1.1528 -1.1804 -1.0710 -1.0577 4.2854]
2	Training	[-1.1793 -1.2658 -1.2370 -0.9097 4.3323]
2	Validation	[-1.1753 -1.2546 -1.2212 -0.9157 4.3151]
2	Training+Val	[-1.1926 -1.0391 -0.9726 -1.1955 4.2854]
	Expected Relevance	[-1 -1 -1 -1 4]

Table 3.2: Weight Vectors for classification derived from OLS on various data sets.

Table 3.3 compares the accuracy results using the scores from weight vectors calculated by OLS on training and/or validation sets against Expected Relevance.

Weight	Yahoo LTRC Set 2 Val			Yahoo LTRC Set 2 Test		
Vector	RMSE	ERR	NDCG	RMSE	ERR	NDCG
Expected Relevance	0.62900	**0.45245**	0.77631	0.63736	0.46113	0.77157
OLS on training	0.63144	0.45205	0.77572	0.63972	0.46115	0.77136
OLS on validation	**0.62772**	0.45238	**0.77653**	**0.63585**	0.46112	**0.77258**
OLS on training+val	0.62911	0.45232	0.77612	0.63727	**0.46123**	0.77144
Weight	Yahoo LTRC Set 1 Val			Yahoo LTRC Set 1 Test		
Vector	RMSE	ERR	NDCG	RMSE	ERR	NDCG
Expected Relevance	0.76318	**0.45679**	**0.77175**	0.74884	**0.46280**	**0.77705**
OLS on training	0.76463	0.45659	0.77151	0.75028	0.46275	0.77675
OLS on validation	**0.76102**	0.45669	0.77138	**0.74670**	0.46269	0.77696
OLS on training+val	0.76417	0.45663	0.77150	0.74981	0.46278	0.77676

Table 3.3: Expected Relevance vs Ordinary Least Squares.

On Yahoo Set 2, the best performer in regards to each metric varies. Only the scoring function from OLS using the training set did not perform the best in regards to any metric. The scores are for the most part very similar; the performance gains from choosing a scoring scheme other than Expected Relevance may not be significant.

On Set 1, Expected Relevance scores the best on every metric except RMSE. This is to be expected as OLS by definition chooses the weight vector which minimizes the squared regression error, and thus the RMSE. Remember that is has been proven that the ERR and NDCG is bounded by the classification error. However the results indicate that reducing the RMSE (ie the classification error) by Ordinary Least Squares will not consistently increase ERR or NDCG. In this case, the derived weight vectors could be overfitting to the input data.

We can conclude from these experiments that Expected Relevance is a very good scoring scheme for translated the probabilities $P(y_i = c)$ into a predicted class. Given the class probabilities (or class cumulative probabilities), doing anything fancier than Expected Relevance is not likely to reap significant rewards.

Chapter 4

Performance Results

This chapter analyzes the run-time performance and accuracy results of the experiments that have been described.

4.1 Run-Time

Table 4.1 summarizes the training times of Random Forests ($M_F = 1000, 10000$), GBRT ($M_B = 1000$), and IGBRT (initialized with RF, $M_F = 10000$ and boosted for $M_B = 500$ iterations). They were run on the Yahoo Set 2 data set, using a Intel Xeon L5520 @ 2.227 GHz computer. The implementations were parallelized; each algorithm used $p = 8$ processors. Random Forests has each of the eight processors build $\lceil M/8 \rceil$ trees. GBRT splits the features into partitions and then each processor can independently calculate the loss for splitting on $\lceil f/8 \rceil$ features.

With M is fixed, Random Forests runs much faster than GBRT. When $M_{RF} = 10000 = 10M_B$ (which is was the parameter for our final classifiers), Random Forests takes slightly longer to train than GBRT. But remember that, when using rules of thumb, Random Forests does not require any parameter sweeping. GBRT on the other hand, may require several runs to find an acceptable M_B and d. Classification algorithms tend run around four times longer than their respective regression algorithms; this is to be expected as the classification paradigm converts the regression problem into four binary classification problems (which ofcourse can be run in parallel).

Running Time	RF	RF	GBRT	I-GBRT
Iterations M	10000	1000	1000	10000/500
Regression	130m	16m	103m	181m
Classification	768m	77m	388m	842m

Table 4.1: Run-Times for training the learners.

All of the learners can be trained on Yahoo Set 2 using one computer on a single day. However, our implementation takes much longer for Yahoo Set 1 and the five Microsoft LETOR Folds.

4.2 Accuracy

Table 4.2 shows how all of the algorithms (Random Forests, Gradient Boosted Decision Trees, and Initialized Gradient Boosted Regression Trees) compare with one another. Performance of a pairwise technique, SVMrank, is also included on the Yahoo data sets. Public LETOR baselines using other algorithms are shown in Table 4.3, these scores are of NDCG@10, the same metric that has been used through out this thesis.

For Random Forest, we chose parameters $M = 10000$ and $k = 0.1f$. For GBRT, we used 1000 trees with a learning rate $\alpha = 0.1$. We chose a strong learning rate because given our implementation of decision trees, smaller learning rates would have to run for a prohibitively long amount of time on the large data sets Yahoo Set 1 and the five Microsoft folds. For IGBRT, we initialized from the output of Random Forests, then boosted for 500 additional iterations with the same parameters as described for Random Forest and GBRT individually. For SGBRT, we picked $s = 0.20$ for all the data sets. k-SGBRT uses $k = 0.1f$. For Norm k-SGBRT, the same k value was used, with $b = 20$ for the Yahoo Set 2, and $b = 10$ for the rest. Because it takes a very long time for the Norm k-SGBRT runs to finish, we chose a stronger step size for it, $\alpha = 0.2$. The same settings were used for regression and classification. For SVMrank we trained with the parameter $c = 1$ (calculated quickly with cross validation), which indicates the trade-off between the margin and training error.

The three point-wise algorithms were evaluated using both the traditional regression framework and the relatively newly introduced classification framework. The algorithms were ran on the Yahoo data sets and five Microsoft Learning to Rank data sets. Recall their statistics from Table 1.1. Each of the respective learners were trained on the training set, and evaluated on the validation and test sets. The best metrics on the validation data sets are reported. The parameters on the test set are from chosen from the parameters that performed the best on its respective validation set.

The table shows the following trends:

1. Random Forests can outperform GBRT.
2. IGBRT reliably outperformed RF.
3. Classification tended to outperform Regression.
4. IGBRT performed the best on most of the data sets.

In regards to NDCG, Random Forests(R) performed better than GBRT(R) in four of the seven data sets. With classification, Random Forests(C) only performed better on three of the data sets. So the classification paradigm seems to work better for GBRT than RF. With ERR, Random Forests performed much better than GBRT only under the classification setting; with regression RF outperformed GBRT on only three of the seven data sets. Even so, these results show that Random Forests can be a good alternative to GBRT for web-search ranking, especially when using classification. Random Forests have not been a focus of study on recent research for learning to rank in the past decade [5]; these results show that further research on Random Forests for Learning to Rank warrants some merit.

IGBRT(R) immediately starts to overfit on Yahoo Set 1, in regards to NDCG. This shows that IGBRT does not always improve upon RF, so care must be taken when using this technique. However it did outperform RF on all of the other data sets; IGBRT did perform the best on six of the seven data sets, in regards to NDCG. With ERR, GBRT does outperform IGBRT on three of the seven data sets. Norm kSGBRT

[5]See a large (but not exhaustive) list of papers on Learning to Rank released during the last decade at http://research.microsoft.com/en-us/um/beijing/projects/letor/paper.aspx.

with classification, NkSGBRT(C), does not improve over kSGBRT(C) as much as the regression setting. This can be attributed to the strong step-size we used for Norm k-SGBRT. SGBRT performed better than k-SGBRT on most of the sets.

Pair-wise and list-wise approaches have become the standard in current learning to rank models [5]. SVMrank is a open-source, pairwise learner that implements support vector machines. Every point-wise learner we have described performs better than the pairwise learner SVMrank on six of the seven data sets in regards to both ERR and NDCG. This helps show that point-wise learners can have comparable performance to the pair-wise approaches, and possibly list-wise. Unfortunately we were unable to find any open-source implementations of learning to rank models that use a list-wise approach.

In the Yahoo Learning to Rank Competition, IGBRT with classification would have scored eleventh place on Set 1, and with regression would have scored fourth place on Set 2 [6]. This is particularly impressive as most of the top competitors used either the pair-wise or list-wise approach, which tend to learn a classifier that optimizes the NDCG and/or ERR metrics. The metric that the competition ranked submissions by was ERR.

Regarding the five LETOR folds, our point-wise algorithms beat all the baselines shown except for Fold 1. Fold 1 is the only fold where the other approaches are not completely beaten by the point-wise algorithms. On this fold, RankSVM performed the best, which IGBRT(C) being the second best performing algorithm.

[6]See top scoring entries at http://learningtorankchallenge.yahoo.com/leaderboard.php.

ERR	R./	Yahoo LTRC		LETOR MQ2007 Folds				
method	C.	Set 1	Set 2	F1	F2	F3	F4	F5
GBRT	R	0.45304	0.45670	0.35914	0.35539	0.35579	0.36039	0.37076
SGBRT	R	0.45566	0.45859	0.36150	0.35826	0.35225	0.36364	0.37292
kSGBRT	R	0.45439	0.45718	0.35747	0.35496	0.35004	0.35088	0.36309
NkSGBRT	R	0.46043	0.46026	0.35933	0.35806	0.35352	0.36304	0.37052
RF	R	0.46349	0.46212	0.35481	0.35458	0.34775	0.35853	0.36853
I-GBRT	R	0.46301	**0.46303**	0.35787	0.35985	0.35383	0.36491	0.37422
GBRT	C	0.45448	0.46008	**0.36264**	0.36168	**0.35990**	0.36498	**0.37549**
SGBRT	C	0.45814	0.45982	0.36133	0.36007	0.36009	0.36764	0.37532
kSGBRT	C	0.45886	0.46027	0.36005	0.35691	0.35580	0.35052	0.36956
NkSGBRT	C	0.45966	0.46062	0.35861	0.35581	0.35265	0.36022	0.36792
RF	C	0.46308	0.46200	0.35868	0.35677	0.35003	0.36364	0.37052
I-GBRT	C	**0.46360**	0.46246	0.36232	**0.36198**	0.35486	**0.36744**	0.37430
SVMrank		0.43037	0.43175					

NDCG	R./	Yahoo LTRC		LETOR MQ2007 Folds				
method	C.	Set 1	Set 2	F1	F2	F3	F4	F5
GBRT	R	0.75215	0.76495	0.47358	0.47414	0.46877	0.47703	0.48105
SGBRT	R	0.76174	0.76602	0.47261	0.47477	0.46559	0.47723	0.48180
kSGBRT	R	0.75756	0.76488	0.47143	0.46918	0.46463	0.47460	0.47534
NkSGBRT	R	0.76843	0.77020	0.47332	0.47643	0.46856	0.47989	0.48403
RF	R	0.77799	0.7746	0.46893	0.47505	0.46577	0.47638	0.48427
I-GBRT	R	0.77799	**0.77633**	0.47263	0.48247	0.47242	0.48327	**0.49088**
GBRT	C	0.75470	0.7704	**0.47785**	0.47877	0.47105	0.48308	0.48783
SGBRT	C	0.76735	0.7696	0.47717	0.47722	0.47264	0.48333	0.48678
kSGBRT	C	0.76817	0.77154	0.47344	0.47304	0.46795	0.47647	0.48183
NkSGBRT	C	0.76875	0.77070	0.47247	0.47397	0.46701	0.47636	0.48013
RF	C	0.77768	0.77281	0.47161	0.47602	0.46608	0.47963	0.48335
I-GBRT	C	**0.77896**	0.77499	0.47766	**0.48258**	**0.47268**	**0.48484**	0.48981
SVMrank		0.73475	0.73462					

Table 4.2: Performance of the algorithms on the Yahoo and Microsoft web-ranking data sets.

NDCG	MSLR MQ2008 Folds				
method	F1	F2	F3	F4	F5
RankSVM	0.4818	0.4266	0.4461	0.4163	0.4485
ListNet	0.47670	0.42700	0.44780	0.42140	0.44720
AdaRank-NDCG	0.4677	0.4243	0.4416	0.4058	0.4451
AdaRank-MAP	0.4665	0.4217	0.4415	0.4065	0.4311

Table 4.3: Public LETOR baselines.

Chapter 5

Transfer Learning

The two data sets from the Yahoo LTRC competition represent two different countries. Previous work has shown that learning on the two sets jointly can have improvements in performance [10]. We ran experiments on the two sets individually: the goal is to transfer knowledge from one data set (the bigger one), to increase ranking accuracy on the other data set.

We wish to perform better on Set 2, by the help of Set 1. We took two approaches:

1. Use Set 1 to increase the training set of Set 2.
2. Use Set 1 to add features to Set 2.

A machine learning model always performs better if it has more data to train on, assuming the data is drawn from the same distribution that the model will be tested on. By definition, Set 1 and Set 2 are from different populations, and thus from different distributions. We looked at all the documents in Set 1, and compute how similar they are to those in Set 2. Set 2 is then populated with the most similar documents from Set 1, and a model is trained on this new data set.

We did so by partitioning Set 1 into five folds. A model is built on four folds of Set 1 and all of Set 2. The documents in Set 1 are assigned the label 0, and documents in Set 2 assigned 1. The model predicts the labels of the documents in the remaining fold. The predictions can be interpreted as the probability a document belongs to the population of Set 2. Note that a Leave-One-Out approach could be used instead of five folds (which is effectively leave 20% out). Leave-One-Out would lead to more accurate results, but would take considerably longer to run.

The top P% of Set 2 candidates from Set 1 are added to Set 2. We then retrain the model to rank the documents from the validation and test sets of Set 2. The results are shown in Table 5.1.

Top P%	# Docs. Added	Yahoo LTRC Set 2 Val			Yahoo LTRC Set 2 Test		
		RMSE	ERR	NDCG	RMSE	ERR	NDCG
0.0	0	**0.62847**	**0.45042**	0.**77383**	**0.63723**	**0.46133**	**0.77267**
0.1	700	0.62998	0.44921	0.77331	0.63873	0.46078	0.77227
0.2	1400	0.63241	0.44930	0.77310	0.64094	0.46099	0.77147
0.4	2800	0.63599	0.44714	0.77002	0.64530	0.46046	0.77006
0.6	4200	0.64011	0.44596	0.76608	0.64979	0.45840	0.76777
0.8	5600	0.64320	0.44604	0.76608	0.65336	0.45631	0.76548
1.0	7000	0.64499	0.44573	0.76694	0.65568	0.45533	0.76390
5.0	35000	0.65245	0.44343	0.76115	0.66314	0.45375	0.76243
10.0	70000	0.65227	0.44325	0.76216	0.66284	0.45402	0.76263
15.0	105000	0.65215	0.44291	0.76112	0.66280	0.45440	0.76272
20.0	140000	0.65217	0.44362	0.76184	0.66255	0.45429	0.76245
30.0	210000	0.65182	0.44355	0.76355	0.66237	0.45467	0.76267
40.0	280000	0.65140	0.44322	0.76214	0.66201	0.45512	0.76297
50.0	350000	0.65173	0.44484	0.76228	0.66239	0.45508	0.76227
75.0	525000	0.65158	0.44436	0.76319	0.66241	0.45507	0.76201
100.0	700000	0.65157	0.44476	0.76274	0.66224	0.45426	0.76224

Table 5.1: Transfer Learning - Increasing size of Set 2.

The table shows that adding documents to the smaller data set does not help. Even with only adding a small fraction of the most likely documents, all of the metrics start to suffer in accuracy. The accuracy continues to drop as more documents are added, which is to be expected. Note that the base line accuracy (adding no documents to Set 2) is lower than previously mentioned. This is because the transfer learning models are trained on only the features that are common to both Set 1 and Set 2. In this case that is 441 features.

The other approach we took is to add features to the documents of Set 2. Let $H(\cdot)$ be a model trained on Set 1. For a given document x_i in Set 2, we computed ten possible feature additions:

- Q: $H(x_i)$
- P: $P(H(x_i) = c)$ for $c \in 0, 1, 2, 3, 4$
- C: $P(H(x_i) \leq c)$ for $c \in 0, 1, 2, 3$

Table 5.2 shows the results of adding various combinations of the described features to Set 2. Adding a single feature, the Q feature, is the only configuration that yields better results than the baseline. Adding that single feature does not seem to get significantly better accuracy on the test set.

Added	Yahoo LTRC Set 2 Val			Yahoo LTRC Set 2 Test		
Feature(s)	RMSE	ERR	NDCG	RMSE	ERR	NDCG
baseline	**0.62693**	0.45035	0.77468	**0.63589**	0.46210	0.77378
Q	0.62694	**0.45094**	**0.77574**	0.63600	**0.46227**	**0.77409**
P01234	0.62716	0.45001	0.77529	0.63615	0.46168	0.77351
C0123	0.62704	0.45032	0.77432	0.63609	0.46110	0.77249
Q+P	0.63710	0.44753	0.76575	0.63570	0.46166	0.77321
Q+C	0.63648	0.44856	0.76760	0.63603	0.46195	0.77295
Q+P+C	0.63553	0.44962	0.76967	0.63605	0.46201	0.77348

Table 5.2: Performance changes from adding features.

It is worth noting that the two approaches can be combined: add documents from Set 1 that are most likely from Set 2's distribution to Set 2, then use a model from Set 1 to add features to the augmented Set 2. Considering that each approach individually did not yield good results, combining them is not likely to either.

Chapter 6

Cost-Based Decision Trees

In practice, the features which describe a document are not equally cheap to compute. Features cost a variable amount to compute, in terms of time, resources needed, computations involved, etc. For instance, it would be much simpler to compute the click rate of an url (via a database lookup) than a feature which requires some computations on the query. The decision trees discussed so far indiscriminately choose which feature (or attribute) each node will be split on, and assumes all the features are immediately available. But in certain circumstances, the features of a document may be extracted only when neccessary. This is the setting of a new area of research - designing models that are accurate, yet cost-effective [27].

In this section we show how the Random Forests and Gradient Boosted Regression Trees perform with cost-effectiveness in mind. We do so by adding the cost of the feature to the impurity function for which the nodes decide how to split partition the data points. In effect this adds a penalty to the split function for chosing a particular feature. Harsher penalties are assigned if the feature is expensive to acquire. We are interested in seeing if this addition of a feature penalty has drastic effects on the performance of the classifiers.

In this context, we will give the documents binary labels. A label of 1 is referred to as "relevant", 0 as "irrelevant". We treat relevances of 0,1, and 2 as "irrelevant", 3 and 4 as "relevant".

6.1 Incorporating Feature Costs into the Loss Function

Suppose each feature f has an associated cost to compute, $c(f)$. Recall that the CART algorithm we are using minimizes the regression error of the two children nodes, $I_{f,v}(D) = I(D_L) + I(D_R)$. To this we add the cost of the feature, multiplied by some constant λ:

$$I_{f,v}(D) = I(D_L) + I(D_R) + \lambda c(f) \tag{6.1}$$

If $\lambda = 0$, then the loss equation is identical to the squared loss as before. We refer to decision trees which incorporate the feature costs ($\lambda \geq 0$) as cost-aware, and those which do not ($\lambda = 0$) as cost-unaware or cost-oblivious. If λ is very large, then the choice of feature to split on will be chosen exclusively by its cost, instead of how well it splits the data points. We use $\lambda \leq 1$, so it will not have a dominating impact on the loss function. It needs to be ensured that the metric the costs are measured in is comparable to the total regression loss of the labels. This depends on what values the labels can take, and the number of documents the model is trained on. We also assume that the cost of constructing a decision tree is 1, regardlesss of what features it will use. The primary interest in assigning costs to features, is to have an idea of how costly it is to classify a single document. The goal is to build a model such that its accuracy is acceptably high, and its cost to classify a document is acceptably low.

We denote the set of features used by the nodes in a decision tree h as $n(h)$:

$$n(h) = \{f_1, f_2, \cdots, f_{h_f}\} \tag{6.2}$$

and then the cost of a decision tree h is total cost of the features use by h:

$$c(h) = \sum_{k=1}^{|n(h)|} c(f) \text{ for } f \in n(h) \tag{6.3}$$

and finally the cost to classify a data point x_i by a boosted classifier $H = h_1, h_2, \cdots, h_T$ is the average tree cost:

$$c_H(x_i) = \frac{1}{T} \sum_{t=1}^{T} c(h_t) \tag{6.4}$$

The trees in this model minimize the loss function in Equation 6.1 with one slight modification. Once a decision tree in the classifier has used a feature, that feature becomes free for the subsequent trees to use. So once h_i uses feature f, the cost for f changes to $c(f) = 0$ for later trees $h_j, j > i$. This is because once a feature has been computed, it does not need to be recomputed. Suppose a tree in the classifier decides to split by PageRank. It takes some computation to calculate this value, but the value only needs to be calculated once. Thus cost of PageRank becomes, in a sense, "free to use" for the later trees in the classifier.

As such, the Random Forests algorithm needs to be modified. They can no longer be made completely in parallel; each tree needs to know which features the previous trees have used. The algorithm can be changed into a completely stage-wise cascade like GBRT, having each tree parallelized by the features. Since the full tree is being made, this will make better use of the processors than GBRT. However, parallelizing by trees instead of features is still more efficient. A combination of the traditional Random Forest algorithm with independent trees and the GBRT cascade can be made, in a way that still achieves good parallelism. Each processor will make one tree in parallel. They can then communicate which features were used and wait until all the processors are finished, and continue with the next p trees. There can be some cost inefficiencies here; in each stage, the trees aren't aware of which features the other $p - 1$ have chosen. We assume the inefficiencies from this is negligible.

One final augmentation we do is injecting early exits into the classifier. Every X iterations, the bottom Y% of data points are "weeded out" or "frozen". The data points are not passed to the subsequent trees to classify. The bottom data points are the ones that the classifier gives the lowest predictions for. In other words, these data points that the classifier deems most irrelevant. The intuition is that, the bottom data points are the ones that are most unlikely to become relevant over time. Thus it is not needed to pass these data points onto all of the decision trees. After a certain

amount of iterations, these data points are weeded out from the test sets. Weeding out data points serves to reduce the overall average cost of classifying a data point.

6.2 Metrics

We introduce two more metrics that are well known for problems with binary labels: Area Under the ROC Curve (AUC) [2], and Precision.

ROC refers to data points as positive or negative. In our context, positive data points are those which are relevant (label 0), and negatives are irrelevant data points (label 0). AUC can be interpreted as "if document x is positive and document y is negative, what are the chances the predicted label for x is greater that the predicted label of y.". An AUC of 100% or 1.0 indicates a perfect classifier; if a classifier predicts x is more relevant than y, then there is a 100% chance x is actually more relevant than y. An AUC of $\frac{1.00^2}{2} = 0.50$ or 50% can always be achieved with random guessing. In the context of web-search ranking, it makes sense to look at the partial area under the curve (PAUC). We use PAUC@[0,5] (which is AUC under the first 0% to 5% of the curve), which indicates a false positive rate of more than 5% cannot be tolerated. The optimal PAUC@[0,5] score is $0.05 * 1 = 0.05$, and a random guessing score with PUAC@[0,5] is $\frac{(0.05)^2}{2} = 0.00125$. AUC can be viewed as PUAC@[0,100]. The AUC can be computing by considering all positive (label 1) and negative (label 0) document pairs, and returning the percentage of those which have the predicted positive value greater than the predicted negative value. For PAUC@[i,j], only the pairs of which the positive example falls into the i'th to j'th percentile are considered.

The other metric, precision, measures how many relevant documents are the top predictions. We look at the top five documents. The precision of a query for the top 5 documents is defined as

$$PREC@5_\pi = \frac{1}{5} \sum_{i=1}^{5} y_{\pi(i)} \tag{6.5}$$

The final PREC@5 score is the averaged PREC@5 query score. The optimal PREC@5 is not neccessarily 1, all queries may not have at least five relevant documents. This is the case with the data set we ran the experiments on.

The experiments were ran on a subset of Yahoo Set 1. The training, validation, and test splits have the same sizes as the respective data sets in Yahoo Set 2.

6.3 Results

For RF and GBRT we ran 8 different configurations: $\lambda = 0$ and $\lambda = 1$ with weed-out (WO) percentages $WO = \{0, 5, 10, 20\}$. The data points are weeded out every 100 iterations/trees. A WO percentage of 0 is the traditional classifier. Weeding out these data points routinely is also referred to as early exiting; i.e. we place an early exit every 100 iterations. Note that there are more involved early exiting procedures, as described by Chapelle et al [7].

We assigned costs to the features by partitioning the features into five groups, and assigning an exponentially increasing cost value to each group.

6.3.1 Cost-aware Random Forests

First we will analyze cost-aware Random Forests ($M_F = 2000, k = 70$). Figure 6.1 shows how Random Forests performs with the metrics PREC@5, PAUC@[0,5], and AUC. The three subplots in the figure are using each of the tree metrics, respectively. Be sure to note that the x-axis in all of the plots represents the average cost classify a document; it is no longer the number of iterations/trees. Each of these plots will be analyzed in turn.

We will analyze the PREC@5 plot first. This is the first (top) plot in the figure. This plot shows that early exiting (another term for weeding out) does not affect the performance in any significant way. Freezing the predictions of the bottom 20% documents every 100 iterations has the same precision and cost of never freezing any documents. It is not surprising that the early exits do not effect the precision, as

these documents are the most unlikely to be the top 5 relevant documents - which are the only documents that effect precision. But inserting early exits also does not effect the cost. This is most likely because, after 100 iterations, all the features which will be used, have been used. This suggests that Random Forests is not a good classifier to insert early exits into. The next thing to note in the plot is that the $\lambda = 0$ runs and $\lambda = 1$ runs converge to the same precision. This is also surprising; it is intuitive to think that $\lambda = 0$ would result in better accuracy, as it has more freedom in which to build the model. But with Random Forests, that extra freedom to choose the more expensive features does not result in better predictions in regards to the top 5 documents. Thus taking into account the feature costs does not hurt the accuracy, which is exactly one of the attributes which is sought for in a cost-aware model. Not only do $\lambda = 0$ and $\lambda = 1$ converge to the same precision, $\lambda = 1$ converges much sooner. On this data set, $\lambda = 1$ reaches its peak at about cost=3000, and $\lambda = 1$ reaches its peak at around cost=16000. So for the same accuracy, the cost-unaware model costs more than five times less than the cost-aware model. The final thing to note: the cost of building a single cost-unaware tree is the more than building the set of cost-aware trees which give maximum precision. All these observations suggest that Random Forests can be a good efficient cost-aware model.

The next metric is PAUC@[0,5], measured in the second (middle) plot in the figure. $\lambda = 1$ converges to about 0.008, $\lambda = 0$ to a value slightly higher than note. This is much better than the PAAC@[0,5] value for random guessing (0.00125), but still much less than the perfect PAUC@[0,5] value 0.05. With this metric, early exiting has a big impact. Early exiting results in a big drop in performance; it seems in essence to lower the "accuracy ceiling", in a manner of speaking. This happens for both $\lambda = 0$ and $\lambda = 1$. This is most likely due to noise in the data set. Early exits throw away to most predicted irrelevant documents. However since the accuracy is decreasing, some of the documents thrown away are actually relevant. Noise in the data set leads the classifier to believe the documents are irrelevant, and thus the classifier will predict a certain set of data points wrongly, which is what this metric measures.

The last metric is AUC, measured in the third (bottom) plot in the figure. The AUC plot shows trends very similar to those discussed in regards to the PREC@5 plot. The $\lambda = 0$ and $\lambda = 1$ runs converge to the same accuracy. $\lambda = 1$ converges much quicker and costs about 5 times less than the cost-aware model for the same accuracy. Early

exiting does not hurt accuracy or decrease cost. But early exiting did hurt accuracy with PAUC@[0,5]. It does not effect the accuracy for PAUC@[0,100] because the data points thrown away can still be classified correctly for false positive rates above 5%. Note that the accuracy converges to a PAUC of about 0.8. The data points which bring down this score are probably the ones which effect the PAUC@[0,5] metric so strongly.

6.3.2 Cost-aware Gradient Boosted Regression Trees

For boosting, we used SGBRT: $\alpha = 0.1, M_B = 1000, k = .1f$. Figure 6.2 shows how Gradient Boosted Decision Trees performs with the same metrics as mentioned previously, PREC@5, PAUC@[0,5], and AUC.

First we analyze the PREC@5 plot. Here the cost-aware runs ($\lambda = 1$) converge much quicker than the cost-unaware runs ($\lambda = 0$). In fact, at their peaks, the cost-aware runs cost about 6 times less than the cost-aware runs. However, the cost-aware runs do achieve a slightly higher accuracy than the cost-aware runs. The highest PREC@5 for $\lambda = 1$ is around 0.27, while the highest PREC@5 for $\lambda = 0$ is closer to 0.28. Both of these precisions are higher than those achieved by the Random Forests models. Not only is the accuracy higher, but the cost is cheaper. The peak $\lambda = 1$ runs have a cost around 2000, which is half of the peak cost of the cost-aware Random Forests runs. Similar to Random Forests, early exiting does not have any effect here. This is surprising, as freezing a percentage of the documents could have a big impact on the cost of the classifier. Because of these reasons, it seems that GBRT is a better cost-aware classifier than Random Forests in regards to precision.

The next metric is PAUC@[0,5]. This plot has a different partition of the models than the previous plots. Previously, the cost aware and unaware models were divided by cost; the cost-unaware model would be on the left side of the cost spectrum, and the cost-aware on the right. With this plot, the models are partitioned by the PAUC value. The cost-aware model almost always has a lower PAUC@[0,5] value than the cost-aware model. Like Random Forests, early exiting does does effect this metric, albeit not as strongly. The baseline run (no early exiting, the red dotted line), has a "ceiling" higher than the other cost-aware runs that do insert early exits. This is attributed to

noise in the data set; predicted irrelevant data points are weeded out when they are actually relevant. For the cost-aware runs, early exiting does have an effect; runs with higher weed-out percentages have higher accuracy as the cost (i.e. number of trees) increases. However, this positive effect is unintended. Inserting early exists serve to lower the average cost to classify a data point. Here it keeps the accuracy steady. It does so by throwing away data points which otherwise would contribute to over-fitting as the number of iterations increases. With the PREC@5 metric, the accuracy of the cost-aware model slowly approaches the accuracy of the cost-unaware model as the cost increases. That trend does not follow here; the accuracy of the cost-aware model almost immediately reaches and surpasses the cost-unaware model. This shows that GBRT may not be a good cost-aware model in regards to PAUC@[0,5]. However, the peak PAUC@[0,5] value for the cost-aware runs is greater than the cost-aware Random Forest runs, especially those that have early exits.

The last metric is AUC. This plot has the same trend as the PREC@5 plot; the accuracy of the $\lambda = 0$ runs slowly approach the $\lambda = 1$ runs has the cost increases. In this case the $\lambda = 0$ runs surpass the $\lambda = 1$ runs. Again early exits have no effect, for reasons described in the AUC analysis of Random Forest cost-based models. The cost-aware models start off much more accurate than the cost-unaware models and convert to around the same value as cost-aware Random Forest models. Even though they converge to the same value, GBRT does so with much less cost than RF. Cost-aware GBRT peaks with cost=500; cost-unaware reaches this value at cost=4000. For Random Forests, the cost-aware models peak at cost=2500, and the cost-aware models at cost=16000. This shows that GBRT is a better cost-based model than Random Forests when comparing AUC. However, the cost-unaware model still outperforms the cost-aware model. This is to be expected, as the cost-unaware model has the freedom to choose any feature to split on, which can result in "better" decision trees.

6.3.3 RF and SGBRT comparison

Table 6.1 compares RF and SGBRT for cost-effectiveness. The metrics are reported as they were in Table 4.2. The highest scores from the validation set are reported. The scores for the test set are from the parameters (i.e. M) that achieved the highest validation score. The table shows that Random Forests does worse than GBRT in

regards to each metric. However, cost-aware RF does outperform cost-unaware RF in regards to precision and AUC. So if one were to use a RF model, it would not hurt to transition into a cost-aware model.

RF	WO	Yahoo Set 1S Test		
λ	%	PREC@5\cost	PAUC@[0,5]\cost	AUC\cost
0	0	0.24893\16045.0	**0.00853\17760.0**	0.76907\17127.0
0	05	0.24885\15972.1	0.00645\16927.4	0.76668\16885.1
0	10	0.24859\15903.4	0.00524\16173.9	0.76356\16209.5
0	20	0.24765\15777.8	0.00167\14998.0	0.75724\15916.5
1	0	**0.25235\5287.0**	0.00805\6647.0	**0.76955\6183.0**
1	05	0.25184\4757.2	0.00582\3637.8	0.76679\5031.4
1	10	0.25175\4573.5	0.00205\2899.7	0.76387\4343.9
1	20	0.24927\3494.8	0.00093\2427.0	0.75933\3631.7
SGBRT	WO	Yahoo Set 1S Test		
λ	%	PREC@5\cost	PAUC@[0,5]\cost	AUC\cost
0	00	0.26536\16192.0	0.01067\10315.0	0.79535\14450.0
0	05	0.26587\14703.6	0.01071\8011.5	0.79550\13486.8
0	10	0.26578\13462.0	**0.01072\7864.9**	0.79603\12613.1
0	20	**0.27006\11482.4**	0.01063\7571.8	**0.79620\11468.8**
1	00	0.26133\1297.0	0.00834\1080.0	0.78317\1609.0
1	05	0.26142\969.4	0.00817\617.4	0.78310\1326.0
1	10	0.26099\886.4	0.00789\603.3	0.78192\1332.0
1	20	0.26031\761.1	0.00790\520.0	0.78210\865.4

Table 6.1: Performance of cost-aware classifiers.

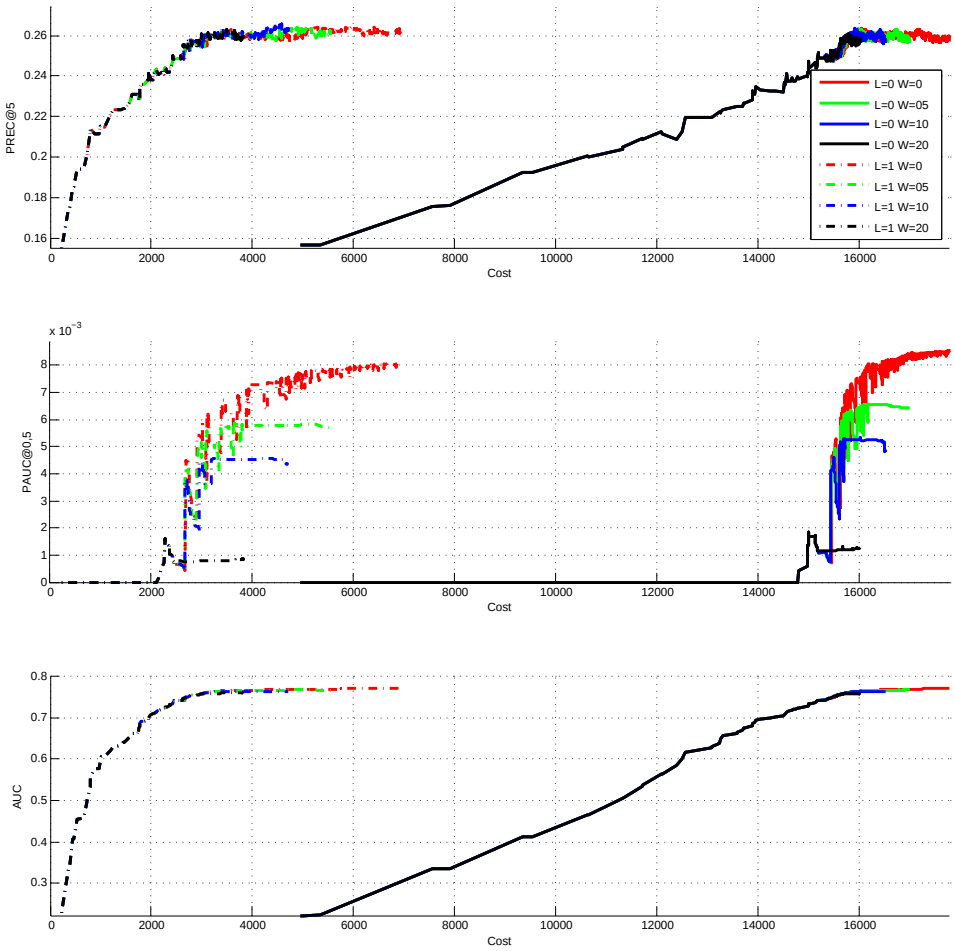


Figure 6.1: Cost-aware Random Forests.

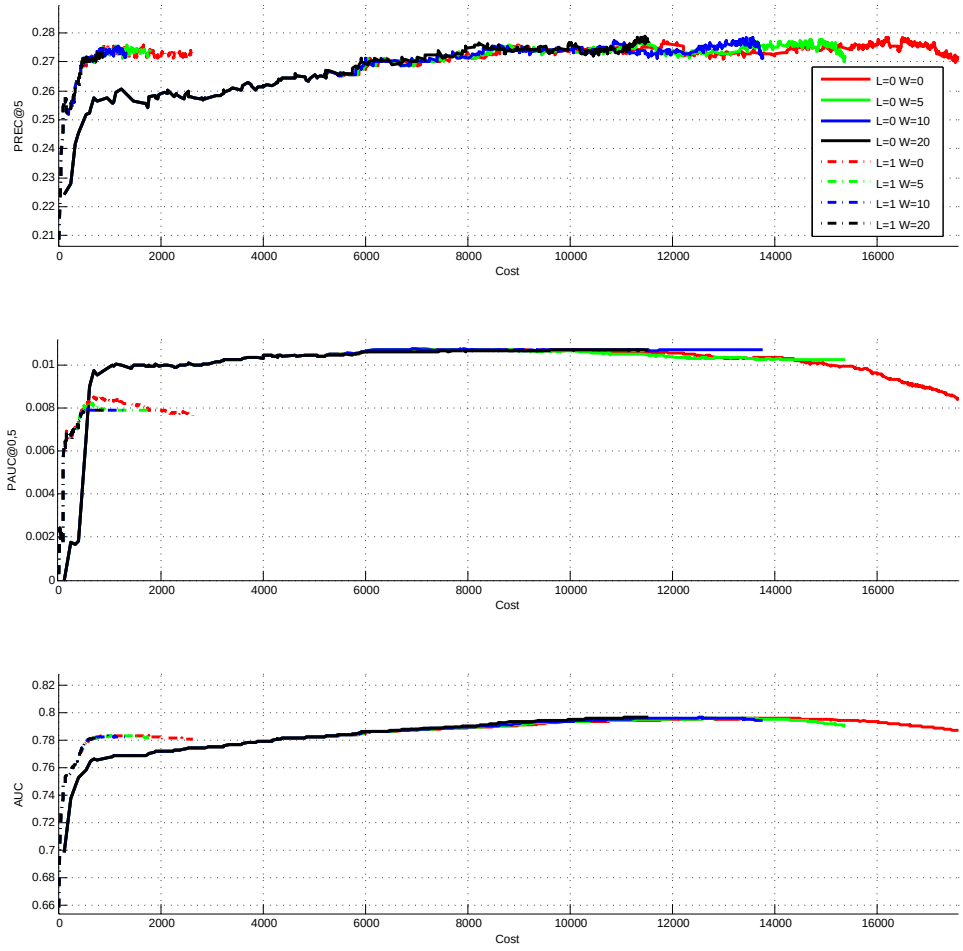


Figure 6.2: Cost-aware SGBRT.

Chapter 7

Conclusions

Here we discuss the things that can be learned from the experiments ran. We discuss in which areas future study would be beneficial.

7.1 Implications

We surveyed the two main point-wise algorithms: Random Forests and GBRT. This thesis has analyzed how the two algorithms performed with web-search ranking in mind. Our experiments show empirical evidence that RF tends to work better than GBRT on real-world data sets. Not only in terms of ranking metrics, but also training time. kSBRT can be a good alternate to GBRT; it runs faster as is not as prone to over-fitting. Combining RT and GBRT produces our newly introduced point-wise algorithm, IGBRT, which performs better than the two individually. Finally, formulating the problem with classification almost always performs better than regression. We showed that scoring the classification problem with Expected Relevance works the best. This provides insight on which point-wise algorithms is appropriate for a problem, given time and accuracy constraints.

None of our experiments for transfer learning proved successful. This shows that more sophisticated approaches need to be researched, or the data sets provided are not conducive for transfer learning. Our experiments show that Random Forests and GBRT can be good cost-aware models by simply adding a penalty for splitting on a particular feature. GBRT gave better accuracy with regards to cost than Random

Forests. Unfortunately there are no public baselines/implementations for cost-aware point-wise or list-wise models to compare to.

Note that the algorithms presented are applicable to any supervised machine learning problem, even though we analyzed them with web-search ranking in mind.

7.2 Future Work

It has been shown that Random Forests are competitive to GBRT in regards to both performance and training time. Yet Random Forests has not been a part of the learning to rank models proposed by the machine learning community. It would be intriguing to see if Random Forests would still be competitive in a controlled, real world production setting. We have shown that initializing GBRT with RF gives great results. In this context GBRT learns the residual error of RF on a data set. We did not explore other ways of combining these two classifiers, such as learning a weight for each classifier for a given document. Similarly, Table 4.2 shows that sometimes regression does perform better than classification. It would be interesting to further explore the relationship between these regression and binary classification problems, and if the two can be combined for a classifier that is stronger than the two individually.

More work needs to be done on Transfer Learning. None of the winning teams on the Yahoo LTRC competition were able to increase performance on Set 2 by learning from Set 1. This suggests that more sophisticated techniques need to be researched. There is a lot of noise in Web-search ranking data sets. Despite the fact that point-wise ranking aims to minimize the regression error, the data indicates that the RMSE never even comes close to 0. This became a problem when we ran tests on early exiting. In Figure 6.1 we show the area under the ROC curve with early exiting. We originally only allows for 1% false positives, PAUC@[0,1]. However, because of the noise in the data sets, this metric did not prove to be useful; the false positive rate had to of been much more relaxed, but this wouldn't be appropriate in a real world setting. It would be useful to research if there is any to better handle, or ideally reduce, the noise in web-search data sets.

Appendix A

Decision Tree Optimizations

There are several ways to speed up the building of a decision tree. Pre-sorting the features can help the decision trees scale better for large data sets [30] [23]. The decision tree data structure can be made over a distributed system to train on a very large data set which would not fit into local memory [1] [33].

At each node of the tree, the CART algorithm finds the optimal feature f and value v to split on, such that the total impurity of the data points which fall to the left and right subtrees is minimal.

Recall the impurity of a set of documents is defined in equation 1.1. This impurity (ie squared loss) function can be simplified:

$$\begin{aligned} I(\cdot) &= \sum_{i=1}^{n}(y_i - \bar{y})^2 \\ &= \sum_{i=1}^{n}(y_i^2 - 2\bar{y}y_i + \bar{y}^2) \\ &= (\sum_{i=1}^{n} y_i^2) - (2\bar{y}\sum_{i=1}^{n} y_i) + (\bar{y}^2 \sum_{i=1}^{n} 1) \\ &= (\sum_{i=1}^{n} y_i^2) - 2n\bar{y}^2 + n\bar{y}^2 \\ &= (\sum_{i=1}^{n} y_i^2) - n\bar{y}^2 \end{aligned}$$

If f is fixed, the squared loss for $I = I(D_L) + I(D_R)$ for each v can be computed using dynamic programming. Without loss of generality, the only split values v that may change the value of the loss function are each $x_i[f]$ for $i < n$. Split values $\frac{x_i[f]+x_{i+1}[f]}{2}$ increase the margin and should generalize better.

If we know the squared loss for one v split value, we can easily compute the loss for the numerically next v value with the help of some auxilary variables.

The psuedocode for calculating the best split point v for a fixed feature f is shown in Algorithm 6. We assume the data has not been pre-sorted. The best value to split on for each feature can be calculated in parallel. This offers great speedup for incremental techniques such as boosting. With Random Forests, the processors are better used to build the trees in parallel.

Algorithm 6 Compute Best Split Point with D.P.

Input: data set $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, feature f
Initialization: $\bar{y}_L = \bar{y}_R = s = r = 0$
$D \leftarrow sorted(D)$
for $i = 1$ to n **do**
  $r \leftarrow r + y_i * y_i$
  $\bar{r} \leftarrow \bar{r} + y_i$
end for
$\bar{r} \leftarrow \bar{r}/n$
for $i = 1$ to $n - 1$ **do**
  $s \leftarrow s + y_i * y_i$
  $r \leftarrow s - y_i * y_i$
  $\bar{y}_L \leftarrow (i * \bar{y}_L + y_i)/(i + 1)$
  $\bar{y}_R \leftarrow ((n - i) * \bar{y}_R - y_i)/(n - i - 1)$
  $L \leftarrow s - \bar{y}_L * \bar{y}_L$
  $R \leftarrow r - \bar{y}_R * \bar{y}_R$
  $Impurity(x_i[f]) \leftarrow L + R$
end for
return $\underset{v}{\text{argmin}}\ Impurity(v)$

Bibliography

[1] Kanishka Bhaduri, Ran Wolff, Chris Giannella, and Hillol Kargupta. 1 distributed decision tree induction in peer-to-peer systems, 2008.

[2] Andrew P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30:1145–1159, 1997.

[3] Leo Breiman. *Classification and Regression Trees*. Wadsworth and Brooks/Cole Advanced Books and Software, Monterey, CA, 1984.

[4] Leo Breiman. Bagging predictors. In *Machine Learning*, pages 123–140, 1996.

[5] Leo Breiman and E. Schapire. Random forests. In *Machine Learning*, pages 5–32, 2001.

[6] Chris Burges, Tal Shaked, Erin Renshaw, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *In ICML*, pages 89–96, 2005.

[7] Berkant Barla Cambazoglu, Hugo Zaragoza, Olivier Chapelle, Jiang Chen, Ciya Liao, Zhaohui Zheng, and Jon Degenhardt. Early exit optimizations for additive machine learned ranking systems. In *WSDM*, pages 411–420, 2010.

[8] Yunbo Cao, Jun Xu, Tie yan Liu, Hang Li, Yalou Huang, and Hsiao wuen Hon. Adapting ranking svm to document retrieval. In *In Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 186–193. ACM Press, 2006.

[9] S. Chakrabarti. Learning to rank in vector spaces and social networks. *Internet Mathematics*, 4(2):267–298, 2007.

[10] O. Chapelle, P. Shivaswamy, S. Vadrevu, K. Weinberger, Y. Zhang, and B. Tseng. Multi-task learning for boosting with application to web search ranking. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1189–1198. ACM, 2010.

[11] Olivier Chapelle and Ya Zhang. Expected reciprocal rank for graded relevance abstract.

[12] Depin Chen, Yan Xiong, Jun Yan, Gui-Rong Xue, Gang Wang, and Zheng Chen. Knowledge transfer for cross domain learning to rank. *Information Retrieval*, 13:236–253, 2010. 10.1007/s10791-009-9111-2.

[13] K. Crammer and Y Singer. Pranking with ranking. In *Advances in Neural Information Processing Systems 14*, pages 641–647. MIT Press, 2001.

[14] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer. An efficient boosting algorithm for combining preferences. *Journal of Machine Learning Research*, 4:933–969, 2003.

[15] Jerome H. Friedman. Stochastic gradient boosting. *Computational Statistics and Data Analysis*, 38:367–378, 1999.

[16] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

[17] Wei Gao, Peng Cai, Kam-Fai Wong, and Aoying Zhou. Learning to rank only using training data from related domain. In *Proceeding of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '10, pages 162–169, New York, NY, USA, 2010. ACM.

[18] Thore Graepel, Joaquin Quionero Candela, Thomas Borchert, and Ralf Herbrich. Web-scale bayesian click-through rate prediction for sponsored search advertising in microsofts bing search engine.

[19] Kalervo Jrvelin and Jaana Keklinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems*, 20:2002, 2002.

[20] Igor Kononenko, Edvard Šimec, and Marko Robnik-Šikonja. Overcoming the myopia of inductive learning algorithms with relieff. *Applied Intelligence*, 7:39–55, January 1997.

[21] P. Li, C.J.C. Burges, and Q. Wu. Learning to rank using classification and gradient boosting. In *Advances in Neural Information Processing Systems 20*, 2008.

[22] T. Lui, T. Joachims, L. Hang, and Z. Chengxiang. Introduction to special issue on learning to rank for information retrieval. *IR*, 13(3):197–200, 2009.

[23] Manish Mehta, Rakesh Agrawal, and Jorma Rissanen. Sliq: A fast scalable classifier for data mining. pages 18–32, 1996.

[24] A. Mohan, Z. Chen, and K. Weinberger. Web-search ranking with initialized gradient boosted regression trees (under review). 2010.

[25] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[26] J. R. Quinlan. Induction of decision trees. *Mach. Learn*, pages 81–106, 1986.

[27] Vikas C. Raykar, Balaji Krishnapuram, and Shipeng Yu. Designing efficient cascaded classifiers: Tradeoff between accuracy and cost, 2010.

[28] M. Richardson, A. Prakash, and E. Brill. Beyond pagerank: machine learning for static ranking. *Proceedings of the 15th international conference on World Wide Web*, 2006.

[29] Joseph Sexton and Petter Laake. Logitboost with errors-in-variables. *Computational Statistics and Data Analysis*, 52(5):2549 – 2559, 2008.

[30] John Shafer, Rakeeh Agrawal, and Manish Mehta. Sprint: A scalable parallel classifier for data mining. pages 544–555. Morgan Kaufmann, 1996.

[31] Michael Taylor, John Guiver, Stephen Robertson, and Tom Minka. Abstract softrank: Optimizing non-smooth rank metrics.

[32] M. Tsai, T. Liu, H. Chen, and W. Ma. Frank: A ranking method with fidelity loss. Technical Report MSR-TR-2006-155, Microsoft Research, November 1999.

[33] S. Tyree, K. Weinberger, K. Agrawal, and J. Paykin. Parallel boosted regression trees for web search ranking (under review). 2011.

[34] Maksims N. Volkovs and Richard S. Zemel. Boltzrank: Learning to maximize expected ranking gain.

[35] S. Xiaoyuan and M. T. Khoshgoftaar. A survey on collaborative filterting techniques. *Advances in Artificial Intelligence*, 2009.

[36] Jun Xu. A boosting algorithm for information retrieval. In *In Proceedings of SIGIR07*, 2007.

[37] Jun Xu, Tie yan Liu, Min Lu, Hang Li, and Wei ying Ma. Directly optimizing evaluation measures in learning to rank. In *In SIGIR 08: Proceedings of the 31th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM Press, 2008.

[38] Z. Zheng, H. Zha, K. Chen, and G. Sun. A regression framework for learning ranking functions using relative relevance judgements. *SIGIR '07*, 2007.

[39] Zhaohui Zheng, Hongyuan Zha, Tong Zhang, Olivier Chapelle, Keke Chen, and Gordon Sun. A general boosting method and its application to learning ranking functions for web search. In *Inf. Proc. Sys. Conf*, pages 1697–1704, 2008.

Vita

Ananth Mohan

Date of Birth	July 3, 1987
Place of Birth	St. Louis, Missouri
Degrees	B.S. Computer Science, May 2009 M.S. Computer Science, December 2010
Professional Societies	Association for Computing Machines

Dec 2010

Point-wise Web-Search Ranking, Mohan, M.S. 2010