The theory underlying the YetiRank and YetiRankPairwise modes in CatBoost.

Winning The Transfer Learning Track of Yahoo!'s Learning To Rank Challenge with YetiRank

Andrey Gulin GULIN@YANDEX-TEAM.RU
Yandex
Moscow, Russia

Igor Kuralenok SOLAR@YANDEX-TEAM.RU
Yandex
Saint Petersburg, Russia

Dmitry Pavlov DMITRY-PAVLOV@YANDEX-TEAM.RU
Yandex Labs
Palo Alto, California

Editor: Olivier Chapelle, Yi Chang, Tie-Yan Liu

Abstract

The problem of ranking the documents according to their relevance to a given query is a hot topic in information retrieval. Most learning-to-rank methods are supervised and use human editor judgements for learning. In this paper, we introduce novel pairwise method called YetiRank that modifies Friedman's gradient boosting method in part of gradient computation for optimization and takes uncertainty in human judgements into account. Proposed enhancements allowed YetiRank to outperform many state-of-the-art learning to rank methods in offline experiments as well as take the first place in the second track of the Yahoo! learning-to-rank contest. Even more remarkably, the first result in the learning to rank competition that consisted of a transfer learning task was achieved without ever relying on the bigger data from the "transfer-from" domain.

Keywords: Learning to rank, gradient boosting, IR evaluation

1. Introduction

The problem of ranking, defined as finding the order of search results in response to a user query, has seen a surge of interest in the last decade, caused primarily by its commercial significance and the ever increasing population of users on the Web. In particular, the search market share of commercial search engines is widely believed to be directly dependent on the quality of search result ranking they produce.

A lot of approaches to solving the learning to rank problem were proposed, a majority of which are based on the ideas of machine learning. In the supervised learning context, every query-document pair has an editorial judgement, reflecting the degree of relevance of the document to the query, and is represented in a vector space of features, whose values are either stored in the offline document index or computed in real-time when the query is issued to the search engine.

The machine learning problem then consists of learning a function that ranks documents for a given query as close to the editorial order as possible. This editorial rank approximation task can be solved in a number of ways, including the pointwise approach (e.g., PRank (Crammer and Singer, 2001), (Li et al., 2008)), in which the relevance predictions are optimized directly, the pairwise approach (e.g., FRank (Tsai et al., 2007), RankBoost (Freund et al., 2003), RankNet (Burges et al., 2005), LambdaRank (Burges et al., 2006), and RankSVM (Joachims, 2002) and (Herbrich et al., 2000)), that optimizes the objective function on the pairs of documents retrieved for a query, and the listwise approach (e.g., AdaRank (Xu and Li, 2007), ListNet (Cao and yan Liu, 2007), SoftRank (Taylor et al., 2008) and BoltzRank (Volkovs and Zemel, 2009)) that generalizes the pairwise approach by defining the objective function that depends on the list of all documents retrieved for a query.

In this paper, we introduce a novel pairwise ranking method YetiRank, describe its properties and present results of its experimental evaluation. The main idea of YetiRank is that it models uncertainty in the editorial judgements. In some cases, as for instance with Yahoo!'s learning to rank task, the confusion matrix is not readily available, and has to be inferred. In section 4.2 we present one of the methods of estimating the confusion matrix based just on available query-document judgements. YetiRank also uses a special computation of the residual between the tree and the gradient, that results in even more "greedy" method than traditionally used, see section 3.4 for the details.

Modeling uncertainty in the editorial judgements and a special gradient computation allow YetiRank to improve the performance over LambdaRank which is one of today's leading pairwise ranking methods. YetiRank took the first place in the transfer learning track of Yahoo!'s Learning to Rank Challenge, outperforming all other methods, including listwise ones. YetiRank managed to win without making any use of the bigger data from the domain learning should have been transferred from. In the experimental results section, we estimate the contribution of modeling the uncertainty in the editorial judgements and the special gradient computation on the overall performance of YetiRank.

The paper is organized as follows. Section 2 introduces notation and defines the main ranking quality metrics we use: ERR and $NDCG$. In section 3, we describe our learning algorithm in detail, in particular, section 3.4 presents our procedure for computing the gradient and section 3.2 discusses how the uncertainty in editorial judgements is incorporated into the computation. Section 4 details the confusion matrix computation methodology. Section 5 presents the results of experimental evaluation showing YetiRank's high standing among the set of the top competing models. We conclude the paper in section 6 by summarizing our contributions and setting the directions of future work.

2. General framework

Let $Q = \{q_1, \ldots, q_n\}$ be the set of queries, $D_q = \{d_{q1}, \ldots, d_{qm_q}\}$ be the set of documents retrieved for a query q, and $L_q = \{l_{q1}, \ldots, l_{qm_q}\}$ be the editorial relevance labels for the documents from the set D_q. Every document d_{qi} retrieved for the query q is represented in the vector space of features, such as text relevance, link relevance, historical click activity, and the like, describing the associations between the query and the document. We want to learn the ranking function $f = f(d_{qi})$, such that the ranking of documents d_{qi} for all queries from Q based on their scores x_{qi} under f, $x_{qi} = f(d_{qi})$, is as close as possible to the ideal ranking conveyed to us in the training data by the editorial judgements l_{qi}.

We use ERR and $NDCG$ ranking quality metrics to measure the deviation of the learned ranking induced by the function f from the ideal one. The expected reciprocal rank metric, or ERR, is defined for a given query q and the set of m_q documents retrieved for q as follows

$$ERR \quad = \quad \sum_{k=1}^{m_q} \frac{1}{k} l_{qk} \prod_{j<k} (1 - l_{qj}), \tag{1}$$

assuming that labels l_{qj} stand for probabilities of user satisfaction after observing document j for the issued query q.

In the same notation, the discounted cumulative gain (DCG) metric is defined as

$$DCG(q_i) \quad = \quad \sum_{k=1}^{m_q} \frac{l_{qk}}{\log_2{(k+1)}}, \tag{2}$$

where l_{qk} is the editorial relevance labels for the document retrieved for q at position k counting from the top after sorting the documents with respect to the scores $x_{qi} = f(d_{qi})$, and the denominator of the fraction under the sum serves the goal of limiting the influence of prediction on the metric as the position grows.

The $NDCG$ is a normalized version of the DCG metric, where the normalization is taken with respect to the maximum achievable DCG for a given query and the set of editorial judgements. The normalization used in $NDCG$ allows us to compare the results of ranking across different queries in a sound fashion. The full DCG and ERR are the averages of per query DCGs and ERRs respectively.

Both ERR and $NDCG$ are not continuous in relevance predictions x_{qi}, and, thus, cannot be directly optimized by the gradient boosting procedure. In section 3.2, we describe the differentiable pairwise loss function which we use instead of discontinuous ERR and $NDCG$.

3. Learning

In this work, we use Friedman's gradient boosting (Friedman, 2001) with trees as weak learners as a basis for our approach which we describe in a nutshell in sec. 3.1. In sec. 3.2 we introduce pairwise loss function we use, in sec. 3.3 we provide details on our LambdaRank implementation and finally in sec. 3.4 we show how one can improve gradient alignment by sacrificing learning time.

3.1. Stochastic gradient descent and boosting

The problem of finding the minimum of function $\mathbb{L}(h)$ on a linear combination of functions from a set H can be solved by the gradient descent, wherein the optimization starts with a function $h_0(d) \in H$ and proceeds with iterations $h_{k+1}(d) = h_k(d) + \alpha \delta h_k$, where α is a small step and δh_k is a function from H. The choice of the loss function is in our hands, e.g. if we set the loss to mean squared error then $\mathbb{L}(h) = \sum_{qi} (h(d_{qi}) - l_{qi})^2$. By differentiating $\mathbb{L}(h)$ we obtain the following gradient for each query/document pair:

$$d\mathbb{L}(h_k(d_{qi})) = 2(h_k(d_{qi}) - l_{qi}) dh_k(d_{qi}), \tag{3}$$

where $q = 1, \ldots, n$; $i = 1, \ldots, m_q$ varies across all labeled query-documents available for training. The idea of gradient boosting then consists of choosing an appropriate function δh_k from H such that adding it to the sum would most rapidly minimize the loss. In other words we want to select δh_k such that we would reduce our loss the most by moving along it for certain distance. To find such δh_k for our mean squared error loss we optimize the following function:

$$\underset{\delta h_k \in H}{\operatorname{argmin}} \sum_q \sum_{i=1}^{m_q} (\delta h_k(d_{qi}) + (h_k(d_{qi}) - l_{qi}))^2 \tag{4}$$

Solution of (4) is not normalized, length of δh_k is proportional to gradient, so moving along it slows down as descent progresses.

We use a set of oblivious trees (those with the same condition in nodes of equal depth (Kohavi and Li, 1995)) with 6 total splits and 64 leaves as a set of functions H, and the solution h_k resulting from optimization is a linear combination of trees like this. On each descent iteration we add a tree aligned in a certain way with the gradient. We construct the tree greedily by adding one split a time. To make oblivious tree of depth 6 we add 6 splits. Tree construction is linear in the number of possible splits. We don't consider every possible split, it would be very costly for the method we use. For each feature we sort feature values, split sorted values into 32 equal size bins and use bin min/max values as possible splits. To improve stability and improve the quality of the final solution, we follow Friedman's recommendation (Friedman, 1999) and randomize the data we use at every iteration by sampling the query-document records for the purposes of learning tree splits. The final predictions in the leaves are still computed using full, unsampled data.

3.2. Pairwise ranking learning

YetiRank belongs to a family of pairwise learning methods that optimize the objective function defined on pairs of documents retrieved for a given query. In this case, the quadratic loss reviewed above in section 3.1 is replaced with log-loss on each pair of documents inspired by RankNet (Burges et al., 2005):

$$\mathbb{L} = -\sum_{(i,j)} w_{ij} \log \frac{e^{x_i}}{e^{x_i} + e^{x_j}}, \tag{5}$$

where index (i, j) goes over all pairs of documents retrieved for a query, w_{ij} is the weight of a given pair of documents, x_i and x_j are predictions of the ranking function for the query and documents i and j. In terms of section 3.1 $x_i = h_k(d_{qi})$ for current iteration k and some query q.

The treatment of the weight w_{ij} plays the key role in the effectiveness of the proposed solution and we define it here precisely.

We use the following intuition: the weight of the pair is dependent on the position of the documents in ranking so that only pairs that may be found near the top in ranking are important, these are the documents that have the high score. To figure out which documents may end up at the top we perform 100 random permutations of the scores according to the following formula:

$$\hat{x_i} = x_i + log \frac{r_i}{1 - r_i}, \tag{6}$$

where r_i is a random variable uniformly distributed in $[0, 1]$. Once the new scores $\hat{x_i}$ are sampled, and the documents are re-ranked according to the new scores, an accumulated weight of every pair of consecutive documents gets incremented with a score of $1/R$, where R is the rank of the highest ranked document of the two, and no other pairs of documents are affected. After all 100 trials are complete, each pair of documents that was observed occurring consecutively in at least one trial will have a non-zero accumulated weight, N_{ij}, equal to the sum of its reciprocal ranks in these trials, all of the rest of the pairs of documents will have a weight of 0. N_{ij} is symmetrical since we are incrementing both N_{ij} and N_{ji} for each pair.

For the final weight w_{ij} on the current iteration of the algorithm we adopt the following model:

$$w_{ij} = N_{ij} c(l_i, l_j), \tag{7}$$

where $c(l_i, l_j)$ is a function of manual labels of the documents i and j. A generic way to account for labels is to use $c(l_i, l_j) = l_i - l_j$ and take only pairs with positive $c(l_i, l_j)$. We modify it as follows:

$$c(l_i, l_j) = \sum_u \sum_v 1_{u>v} p(u|l_i) p(v|l_j), \tag{8}$$

where u and v run over all unique numeric values of editorial labels, and the confusion matrix $p(u|v)$ represents the probability that the true label v is confused with label u by the editors. This is the most important YetiRank's modification that incorporates our belief in the importance of modeling uncertainty in the editorial judgements. We discuss the treatment of the confusion matrix in more detail in section 4.

Note that N_{ij} and, hence, w_{ij} are dependent on the model scores that vary from one iteration to another.

Let's define the optimization problem that we will solve on each gradient boosting iteration. The total differential of the pairwise loss function (5) looks like this:

$$d\mathbb{L} = \sum_{(i,j)} w_{ij} \left((dx_i - dx_j) \frac{e^{x_j}}{e^{x_i} + e^{x_j}} \right), \tag{9}$$

In gradient boosting we are searching for the vector δx which minimizes $d\mathbb{L}$. Let's define $y_t = \sqrt{w_{ij}}(\delta x_i - \delta x_j)$, $a_t = \sqrt{w_{ij}} \frac{e^{x_j}}{e^{x_i}+e^{x_j}}$ where t is index of pair in equation (9). Let's search for y of fixed length:

$$\begin{aligned} \underset{y,|y|=const}{\operatorname{argmin}} \sum_t y_t a_t &= \underset{y,|y|=const}{\operatorname{argmin}} \left(1 + 2 \sum_t \frac{y_t a_t}{|a||y|} + 1\right) = \ldots = & (10) \\ &= \underset{y,|y|=const}{\operatorname{argmin}} \sum_t \left(y_t + \frac{|y|}{|a|} a_t \right)^2 & (11) \end{aligned}$$

Now we expand y_t and a_t, define $\lambda = \frac{|y|}{|a|}$ and lift the restriction on $|y|$ being constant. Then we get

$$\underset{\lambda,\delta x}{\operatorname{argmin}} \sum_{(i,j)} w_{ij} \left((\delta x_i - \delta x_j) + \lambda \frac{e^{x_j}}{e^{x_i} + e^{x_j}} \right)^2 \tag{12}$$

Solution of (12) is linear in λ in the sense that if we find a solution i.e. an optimal direction for some fixed λ we can scale it and get a solution corresponding to the same direction for any other λ. So the length of step along this direction can be tuned by scaling. The scaling factor is defined by λ and boosting step α (see section 3.1). We choose $\lambda = 1$. With $\lambda = 1$ the gradient boosting optimization problem reduces to:

$$\operatorname*{argmin}_{\delta x} \sum_{(i,j)} w_{ij} \left((\delta x_i - \delta x_j) + \frac{e^{x_j}}{e^{x_i} + e^{x_j}} \right)^2 . \tag{13}$$

3.3. LambdaRank approximation

For the sake of comparison we implemented an approximation of the LambdaRank method (Burges et al., 2006). We use the same 64 leaf oblivious tree as a weak learner, and modify only the loss function. Our LambdaRank approximation uses generic $c(l_i, l_j)$ and same N_{ij} as other methods in the paper.

In LambdaRank, finding solution of (13) is reduced to a pointwise optimization problem. Indeed, equation (13) can be interpreted as a set of springs attached to documents. Each spring pushes a better scoring document up and the other down. Instead of (13) LambdaRank optimizes the following function:

$$\operatorname*{argmin}_{\delta x} \sum_{(i,j)} w_{ij} \left[\left(\delta x_i + \frac{1}{2} \frac{e^{x_j}}{e^{x_i} + e^{x_j}} \right)^2 + \left(\delta x_j - \frac{1}{2} \frac{e^{x_j}}{e^{x_i} + e^{x_j}} \right)^2 \right] . \tag{14}$$

All spring potentials are added up, and the following weighted quadratic optimization problem is solved:

$$\begin{array}{rcl} Val_i & = & \sum_j w_{ji} \frac{1}{2} \frac{e^{x_j}}{e^{x_i}+e^{x_j}} - \sum_j w_{ij} \frac{1}{2} \frac{e^{x_j}}{e^{x_i}+e^{x_j}} \\ W_i & = & \sum_j w_{ij} + \sum_j w_{ji} \\ \\ & & \operatorname*{argmin}_{\delta x} \sum_i W_i (\delta x_i - \frac{Val_i}{W_i})^2 \end{array} \tag{15}$$

Equation (15) can be solved with traditional gradient descent described in section (3.1).

3.4. Better gradient alignment

Solving minimization problem (13) for the set δx_i gives us a direction in which the step of gradient boosting needs to be performed. Unlike LambdaRank 3.3 we can optimize equation (13) directly without reducing it to the pointwise problem. Note that in each gradient descent iteration we are looking for the tree structure and the values in its leaves. Since each document with score x_i belongs to one and only leaf in the tree, we may define and denote by $Leaf_h(x)$ a function which maps documents to the leafs for a tree h. Let the predicted value of the score of leaf n be c_n. Then equation (13) can be rewritten as follows:

$$\mathbb{L} \; = \; \sum_{ij} w_{ij} \left(c_{leaf(d_i)} - c_{leaf(d_j)} + \frac{e^{x_j}}{e^{x_i} + e^{x_j}} \right)^2 , \tag{16}$$

In this form we get the regular least squares optimization problem and we can use matrix algebra to solve it. In particular, we can introduce matrix A and vector b, such that each row of A and the element of b correspond to a single term of equation (16).

$$A = \begin{pmatrix} & & \vdots & & \\ 0 & 1 & -1 & 0 & \dots \\ 1 & 0 & -1 & 0 & \dots \\ & & \vdots & & \end{pmatrix} \tag{17}$$

$$b = \begin{pmatrix} \vdots \\ -\frac{e^{x_3}}{e^{x_2}+e^{x_3}} \\ -\frac{e^{x_3}}{e^{x_1}+e^{x_3}} \\ \vdots \end{pmatrix} \tag{18}$$

Solving the equation (16) can be then reformulated as a problem of finding minimum of $(Ac-b)^T w(Ac-b)$, and the solution for c can be found as $c = (A^T wA)^{-1} A^T wb$, where w is a square diagonal matrix with w_{ij} on the main diagonal. We can save some computation by precomputing $A^T wA$ and $A^T wb$ for each query in advance and projecting them on the leaves for each candidate tree. Algorithms for doing this are presented below.

Proposed method increases tree split search complexity to $O(N_{Query} * N^2_{QueryDoc})$ from $O(N_{Query} * N_{QueryDoc})$ for methods with the pointwise reduction like in equations (15). N_{Query} is query count and $N_{QueryDoc}$ is document per query count. Also it adds matrix inversion with $O(N^3_{LeafCount})$ complexity. Matrix inversion can take significant time especially when using small training sets and complex trees.

Algorithm 1 Precompute per query $M^q = A^T A$ and $v^q = A^T b$ for query q

```
1: $M^q = 0$ $(m_q$ x $m_q)$
2: $v^q = 0$ $(m_q)$
3: for all $(i, j)$ do
4:     $v^q_i -= w_{ij} \frac{e^{x_j}}{e^{x_i}+e^{x_j}}$
5:     $v^q_j += w_{ij} \frac{e^{x_j}}{e^{x_i}+e^{x_j}}$
6:     $M^q_{ii} += w_{ij}$
7:     $M^q_{ij} -= w_{ij}$
8:     $M^q_{ji} -= w_{ij}$
9:     $M^q_{jj} += w_{ij}$
10: end for
```

4. Confusion Matrix

The ideas of expert judgement variations have been known for quite some time now (Voorhees, 1998). Given the variability inherent to the editorial judgements, it makes sense to model the judgement as a random variable with fixed probabilities of a mark u being confused with v. Below we address several questions related to confusion matrix: whether it depends on

Algorithm 2 Selecting split to add to the tree

```
Require: $M^q$ - precomputed $A^TA$ for query q
Require: $v^q$ - precomputed $A^Tb$ for query q
Require: $doc^q$ - the documents retrieved for query q
 1: for each candidate split of tree h do
 2:   $M^{all} = 0$ {(leafCount * leafCount)}
 3:   $v^{all} = 0$ {(leafCount)}
 4:   for each query q do
 5:     for $i = 1...m_q$ do
 6:       $v^{all}[leaf_h(doc^q[i])] += v^q_i$
 7:       for $j = 1..m_q$ do
 8:         $n_i = leaf_h(doc^q[i])$
 9:         $n_j = leaf_h(doc^q[j])$
10:         $M^{all}[n_i][n_j] += M^q_{ij}$
11:       end for
12:     end for
13:   end for
14:   $c = (M^{all})^{-1}v^{all}$ {c is leaf values}
15:   $tree\ score = c^TM^{all}c - 2v^{all}c$
16: end for
```

instructions given to the experts only or it also varies from one group of editors to another, how to use confusion matrix for model training and how to infer the confusion matrix based just on the labeled query-document pairs.

4.1. Editors Do Vary

In most of the manual data collection approaches to learning to rank problem that we are aware of, the judges follow the shared set of instructions to assign labels to query-document pairs. It is clear that the confusion probability $P(true\ mark\ u|editor\ mark\ v)$ depends on the value of u as well as the set of instructions. In Yandex practice, we obtain labels from editors, who, for instance, may reside and label data in different geographical regions but follow the common set of instructions. Similarly, in the framework of the transfer learning track of Yahoo!'s learning to rank competition, the instructions were likely defined by the company and then used by two groups of editors for data labeling: one for a smaller regional data and the other one for bigger "transfer-from" data. Here we explore the question of variability of grades in a situation like this.

The only way to get data on confusion matrix is an experiment on real editors. To model variability, we took 100 editors working on labeling the data for Yandex's *.ru* domain, and the corresponding 154,000 query-document judgements, and 23 editors working on the Ukrainian data along with their 56,000 query-document judgements. Every query-document pair mentioned above was judged at least twice, and allowed us to construct confusion matrices for each of these domains that we present in table 1. Analysis of entries in this table clearly illustrates the differences between the groups of editors in terms of their labeling mistakes. For instance, the "excellent" grade is much closer to "perfect" in the set

	Bad	Poor	Good	Exc.	Perf.
Bad	0.75	0.22	0.02	0	0
Poor	0.34	0.54	0.11	0.01	0
Good	0.07	0.13	0.73	0.06	0.01
Exc.	0.04	0.04	0.52	0.32	0.08
Perf.	0.03	0.02	0.05	0.08	0.83

	Bad	Poor	Good	Exc.	Perf.
Bad	0.88	0.09	0.02	0	0
Poor	0.26	0.65	0.07	0.01	0
Good	0.05	0.08	0.78	0.07	0.01
Exc.	0.03	0.02	0.24	0.60	0.10
Perf.	0.03	0.02	0.03	0.05	0.86

Table 1: Probability to change editors mark (rows) to true mark (columns) for Russian (left) and Ukrainian (right) editors.

	Bad	Poor	Good	Exc.	Perf.
Bad	0.869	0.103	0.02	0.001	0.007
Poor	0.016	0.878	0.1	0.005	0.002
Good	0.003	0.098	0.85	0.046	0.004
Exc.	0	0.01	0.094	0.896	0
Perf.	0	0	0.019	0.016	0.965

Table 2: Inferred confusion matrix for yahoo LTRC track 2

of Ukrainian judgements than in the set of Russian judgements. This observation leads us to a conclusion that to attempt a straight mixing of the two data sets available for transfer learning without accounting for label confusion is likely a subpar idea.

As one may see from the tables, the confusion matrices are not symmetric. This may be caused by many reasons. One of them is that the separate instruction for each level of relevance, these instructions having different levels of preciseness. If J_1 was defined in a more precise way then J_2, then $p(J_1|J_2)$ could be different from $p(J_2|J_1)$. For example "excellent" mark is way less clear for editors then "good" in our instruction and people set "excellent" marks for better then average "good" documents but this opinion is rarely supported by other experts.

4.2. Inferring confusion matrix from data

Often the confusion matrix between judgement labels is either too expensive or simply impossible to estimate directly as is the case in learning to rank challenge. In this case we have no chance of getting correct confusion matrix. On the other hand we can try to infer this matrix from the data we have. Optimal inferring method depends heavily on the type of data available.

To infer confusion matrix from the contest data we define buckets of query-document pairs that are extremely close to each other (ideally mapped to the same point in feature space) $b = \{d_i : \forall d_j \in b, |d_i - d_j| < \epsilon\}$ and treat these buckets as several judgments for single query-document pair. We can compute likelihood of such bucket with confusion matrix $p(true\ mark|editor\ mark)$ if we know the true mark j for the bucket and assume

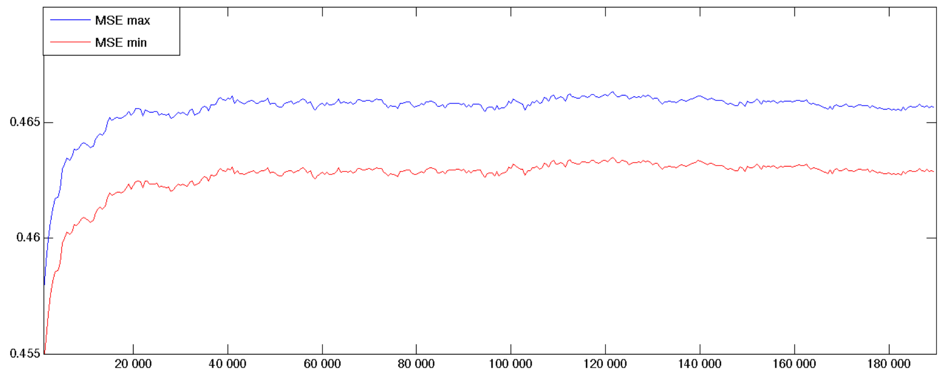


Figure 1: ERR test scores for MSE on track 1 data.

	ERR	NDCG
MSE	0.4607	0.7766
Equal pair weights	0.4598	0.7813
Weighted pairs	0.4624	0.7800
LambdaRank	0.4629	0.7809
Better gradient alignment	0.4635	0.7845
YetiRank	0.4638	0.7870

Table 3: Maximal scores on track 2 test set.

the judgments independence:

$$p(b|j) = \prod_{i=1}^{|b|} p(j|j(d_i)) = \prod_{i=1}^{|J|} p(j|j_i)^{|\{d_t \in b : J(d_t) = J_i\}|} = \prod_{i=1}^{|J|} p(j|j_i)^{n_b(j_i)} \tag{19}$$

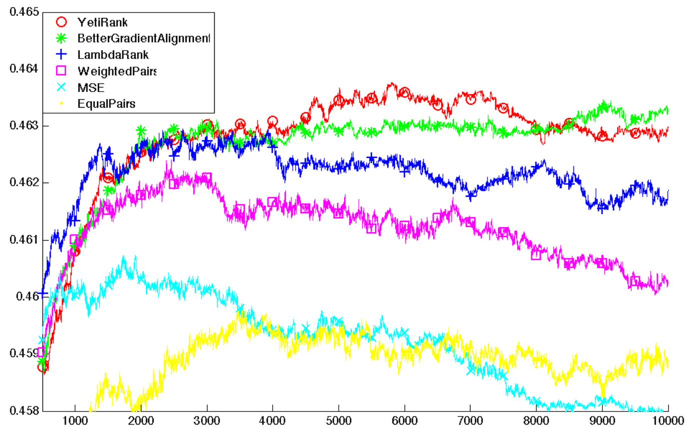
where $n_b(j_i)$ – number of judgments with mark j_i in bucket b. Searching for the confusion matrix maximizing likelihood of all observed buckets b_i solves the problem:

$$C^* = \underset{C}{\operatorname{argmax}} \sum_{i=1}^{|B|} \log p(b_i|j_i) = \underset{C}{\operatorname{argmax}} \sum_{i=1}^{|B|} \sum_{k=0}^{|J|} n_{b_i}(j_k) \log p(j_i|j_k) \tag{20}$$

C – is the confusion matrix. The open question is how to define "true" mark for the bucket. We did this by randomly choosing mark from those in the bucket, weighted by the amount of such mark in the bucket. Table 2 contains the result of this procedure on the Yahoo!'s contest second track data.

5. Experiments

In this section we measure influence of our improvements on final scores. Experiments were performed on the real-world data from the transfer learning track of Yahoo! ranking



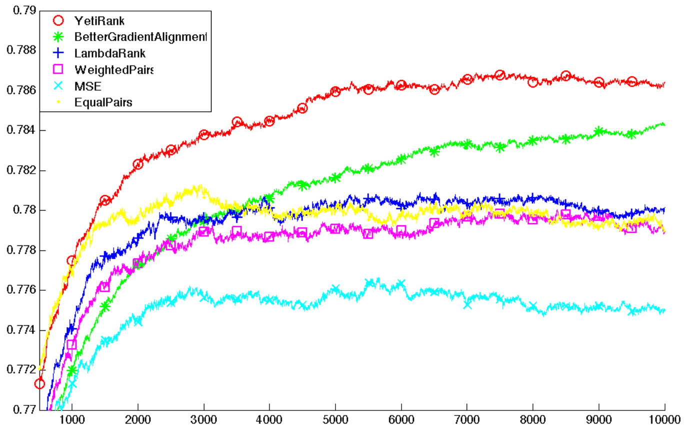


Figure 2: ERR (top) and NDCG (bottom) test scores for proposed algorithms as a function of the number of iterations/trees.

challenge at ICML-2010. We've used track 2 data to compare different methods. Proposed algorithm in its presented form does not converge on track 1 data within reasonable number of iterations, see fig. 1 so it is computationally difficult to find maximum scores for track 1 data. Second problem we found in track 1 data is a number of almost same documents with different marks. Our model assigns same score to such documents and order becomes random, we plotted 2 graphs on fig. 1 to indicate minimum and maximum possible scores that could be achieved by permuting same score documents.

There were 6 experimental setups on track 2 data:

- plain MSE gradient boosting as a baseline;
- pairwise model with equal weights assigned to all pairs, $w_{ij} = 1$ if $l_i > l_j$ and $w_{ij} = 0$ otherwise;
- pairwise model with generic $c(i, j)$ and $N(i, j) = 1$, so we account only label difference, pair position is ignored;
- pairwise model a la LambdaRank from section 3.3;
- better aligned model from section 3.4;
- and finally one for YetiRank.

Figure 2 illustrates how ERR and $NDCG$ metrics respectively behave as a function of the number of trees learned in gradient boosting process. We were using 0.02 shrinkage parameter value with slight variation across methods to ensure comparable convergence rate. The optimization uses the inferred confusion matrix presented in table 2 and described in section 3.4. Note that both ERR and $NDCG$ metrics show improvement after 5000+ iterations over LambdaRank.

The optimal model we built contained 6000 trees total, thus requiring 6000*7 flops per document (6 comparisons for every oblivious tree and one addition). Training takes around 8 hours on a single Linux box with 4 cores.

We also want to underscore that the bigger "transfer-from" data of the contest was not used at all in the optimization, yet yielded the best overall model among all participants. This is not due to the lack of trying, in particular, we tried learning the confusion matrix on the bigger data and also mix the two data sets together, none of which produced a better result. We hypothesize that the bigger data set was too dissimilar to the smaller one and finer instruments needed to be used to try and obtain a better model.

6. Conclusion

In this paper, we presented a YetiRank algorithm for learning the ranking function that performed well on the Yahoo! learning to rank competition task winning the first place in the transfer learning track and likely deserves more attention and further exploration. In our experiments, we established that the gradient boosting approach employing decision trees as a base learner still remains the best approach, however two modifications make it even more powerful: the special gradient alignment computation and using the additional information about the uncertainty in judgments. We demonstrated the effect of each of these modifications on the quality of ranking. We also established that several groups of experts differ in their confusion about various labels, and proposed a way of accounting for this confusion in the modeling process. We further explored ways of learning the confusion matrix between the relevance labels from the query-document labeled data directly and used it in learning to rank competition. One of the possible directions for future research consists of exploring methods of noise reduction in building the transition matrix.

References

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

Chris Burges, Robert Ragno, and Quoc V. Le. Learning to rank with nonsmooth cost functions. In *NIPS*, pages 193–200, 2006.

Zhe Cao and Tie yan Liu. Learning to rank: From pairwise approach to listwise approach. In *Proceedings of the 24th International Conference on Machine Learning*, pages 129–136, 2007.

Koby Crammer and Yoram Singer. Pranking with ranking. In *Advances in Neural Information Processing Systems 14*, pages 641–647. MIT Press, 2001.

Yoav Freund, Raj Iyer, Robert E. Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. In *JOURNAL OF MACHINE LEARNING RESEARCH*, volume 4, pages 170–178, 2003.

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

Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2001.

Ralf Herbrich, Thore Graepel, and Klaus Obermayer. *Large margin rank boundaries for ordinal regression*. MIT Press, Cambridge, MA, 2000.

Thorsten Joachims. Optimizing search engines using clickthrough data. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 133–142, New York, NY, USA, 2002. ACM. ISBN 1-58113-567-X.

Ron Kohavi and Chia-Hsin Li. Oblivious decision trees graphs and top down pruning. In *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 2*, pages 1071–1077, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1-55860-363-8. URL http://portal.acm.org/citation.cfm?id=1643031.1643039.

Ping Li, Christopher Burges, and Qiang Wu. Mcrank: Learning to rank using multiple classification and gradient boosting. In J.C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 897–904. MIT Press, Cambridge, MA, 2008.

Michael Taylor, John Guiver, Stephen Robertson, and Tom Minka. Softrank: optimizing non-smooth rank metrics. In *WSDM '08: Proceedings of the international conference on Web search and web data mining*, pages 77–86, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-927-9.

Ming-Feng Tsai, Tie-Yan Liu, Tao Qin, Hsin-Hsi Chen, and Wei-Ying Ma. Frank: a ranking method with fidelity loss. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 383–390, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-597-7.

Maksims N. Volkovs and Richard S. Zemel. Boltzrank: learning to maximize expected ranking gain. In *ICML'09: Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1089–1096, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-516-1.

Ellen M. Voorhees. Variations in relevance judgments and the measurement of retrieval effectiveness. In *SIGIR '98: Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 315–323, New York, NY, USA, 1998. ACM. ISBN 1-58113-015-5.

Jun Xu and Hang Li. Adarank: a boosting algorithm for information retrieval. In *SIGIR'07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 391–398, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-597-7.