Learning to Rank

Introduction:

Paper1: “From RankNet to LambdaRank to LambdaMART: An Overview”
LambdaMART is the boosted tree version of LambdaRank, which is based on RankNet. RankNet, LambdaRank, and LambdaMART have proven to be very successful algorithms for solving real world ranking problems: for example an ensemble of LambdaMART rankers won Track 1 of the 2010 Yahoo! Learning To Rank Challenge. In this paper the author has given a self-contained, detailed and complete description of them. All the three methods described are used to solve a class of supervised learning problems. In the approach for learning to rank this paper covers only the following approaches:

a) Pairwise approach (Used in RankNet and LambdaRank): In this case learning-to-rank problem is approximated by a classification problem — learning a binary classifier that can tell which document is better in a given pair of documents. The goal is to minimize average number of inversions in ranking.
b) Listwise approach (Used in LambdaMART): These algorithms try to directly optimize the value of one of the above evaluation measures, averaged over all queries in the training data. This is difficult because most evaluation measures are not continuous functions with respect to ranking model's parameters, and so continuous approximations or bounds on evaluation measures have to be used.

So this paper largely concentrates improves on the shortcomings or inefficiencies of each method. RankNet is improved to LambdaRank and LambdaRank is improved to LambdaMART iteratively.

Paper2: “Query-biased Learning to Rank for Real-time Twitter Search”
By incorporating diverse sources of evidence of relevance, learning to rank has been widely applied to real-time Twitter search, where users are interested in fresh relevant messages. In this paper, the author proposes to further improve the retrieval performance of learning to rank for real-time Twitter search, by taking the difference between queries into consideration. In particular, the paper concentrates on learning a query-biased ranking model with a semi-supervised transductive learning algorithm so that the query-specific features, e.g. the unique expansion terms, are utilized to capture the characteristics of the target query. This query-biased ranking model is combined with the general ranking model to produce the final ranked list of tweets in response to the given target query. Extensive experiments on the standard TREC Tweets11 collection show that the proposed query-biased learning to rank approach outperforms strong baseline, namely the conventional application of the state- of-the-art learning to rank algorithms like RankSVM and LambdaMART.

The paper is organized as follows: A survey is done on the existing learning to rank research on Twitter search. In the next section it gives a detail description about the method to build the learning to rank framework by combining the general learning to rank algorithms and the query-biased algorithm, which

is evaluated. Finally, the research is concluded and suggests future research directions, features by a semi-supervised learning algorithm.

Relationship: The aim of both the papers is to model an efficient algorithm to rank the web pages or documents such that retrieval is efficient. In the first paper the aim is to explain the three Learning to Rank algorithms (RankNet, LambdaRank and LambdaRank) in depth and how we go on improving them iteratively. In the second paper detail description is given to build the learning to rank framework by combining the general learning to rank algorithms (RankSVN and LambdaMART) and the query-biased algorithm for Real-time Twitter Search.

Paper1: "From RankNet to LambdaRank to LambdaMART: An Overview"

Main Contribution of the Paper:

The paper talks on the three methods RankNet, LambdaRank and LambdaMART. It is an iterative refining from RankNet to LambdaRank and combining LambdaRank and MART to LambdaMART for better Learning to Rank algorithm formulation. The detailed transition is described in the following sections.

Methods Proposed by the authors, Evaluation on how did the authors present a convincing case:

1) RankNet:

We consider models where the learning algorithm is given a set of pairs of samples [*A, B*] in R^d, together with target probabilities $\overline{P}_{AB}$ that sample *A* is to be ranked higher than sample *B*. This is a general formulation: the pairs of ranks need not be complete (in that taken together, they need not specify a complete ranking of the training data), or even consistent. We consider models $f : R^d \rightarrow R$ such that the rank order of a set of test samples is specified by the real values that *f* takes, specifically, $f(\mathbf{x}1) > f(\mathbf{x}2)$ is taken to mean that the model asserts that $\mathbf{x}1 \blacktriangleright \mathbf{x}2$.

Denote the modeled posterior $P(\mathbf{x}i \blacktriangleright \mathbf{x}j)$ by Pij, $i, j = 1, \ldots, m$, and let $\overline{P}_{ij}$ be the desired target values for those posteriors. Define oi = f (xi) and oj = f (xj) We will use the cross entropy cost function

$C_{ij} = C(o_{ij}) = -\overline{P}_{ij} \log P_{ij} - (1 - \overline{P}_{ij}) \log (1 - P_{ij})$; _____ (1)

where the map from outputs to probabilities are mod- eled using a logistic function (Baum & Wilczek, 1988)

$P_{ij} = e^{o_{ij}} / (1 + e^{o_{ij}})$ _____(2)

Cij then becomes

$C_{ij} = -\overline{P}_{ij} o_{ij} + \log (1 + e^{o_{ij}})$ _____(3)

Note that Cij asymptotes to a linear function; for problems with noisy labels this is likely to be more robust than a quadratic cost. Also, when $\overline{P}_{ij} = 1/2$ (when no information is available as to the relative rank of the two patterns), Cij becomes symmetric, with its minimum at the origin. This gives us a

principled way of training on patterns that are desired to have the same rank; we will explore this below. We plot Cij as a function of oij in the left hand panel of Figure 1, for the three values $\bar{P}$ = {0, 0.5, 1}.

Combining Probabilities: The above model puts consistency requirements on the $\bar{P}$ ij, in that we require that there exist ideal outputs $\bar{o}$ i of the model such that

$\bar{P}$ ij = e ^ $\bar{o}$ ij/ (1 + e ^ $\bar{o}$ ij) _____(4)

Where $\bar{o}$ i j= $\bar{o}$ i – $\bar{o}$ j

This consistency requirement arises because if it is not met, then there will exist no set of outputs of the model that give the desired pair- wise probabilities. The consistency condition leads to constraints on possible choices of the $\bar{P}$'s. For example, given $\bar{P}$ ij and $\bar{P}$ jk, Eq. (4) gives

$\bar{P}$ ik = $\bar{P}$ ij $\bar{P}$ jk / (1 + 2 $\bar{P}$ ij $\bar{P}$ jk – $\bar{P}$ ij– $\bar{P}$ jk) _____(5)

This is plotted in the right hand panel of Figure 1, for the case $\bar{P}$ *ij* = $\bar{P}$ *jk* = P. We draw attention to some appealing properties of the combined probability $\bar{P}$ *ik*. First, $\bar{P}$ *ik* = *P* at the three points *P* = 0, *P* = 0.5 and *P* = 1, and only at those points. For example, if we specify that *P* (*A*► *B*) = 0.5 and that *P* (*B* ►*C*) = 0.5, then it follows that *P* (*A*► *C*) = 0.5; complete uncertainty propagates. Complete certainty (*P* = 0 or *P* = 1) propagates similarly. Finally confidence, or lack of confidence, builds as expected: for 0 < *P* < 0.5, then $\bar{P}$ *ik* < *P* , and for 0.5 < *P* < 1.0, then $\bar{P}$ *ik* > *P* (for example, if *P* (*A*► *B*) = 0.6, and *P* (*B* ► *C*) = 0.6, then *P* (*A* ►*C*) > 0.6). The following figures are plotted to explain the above:

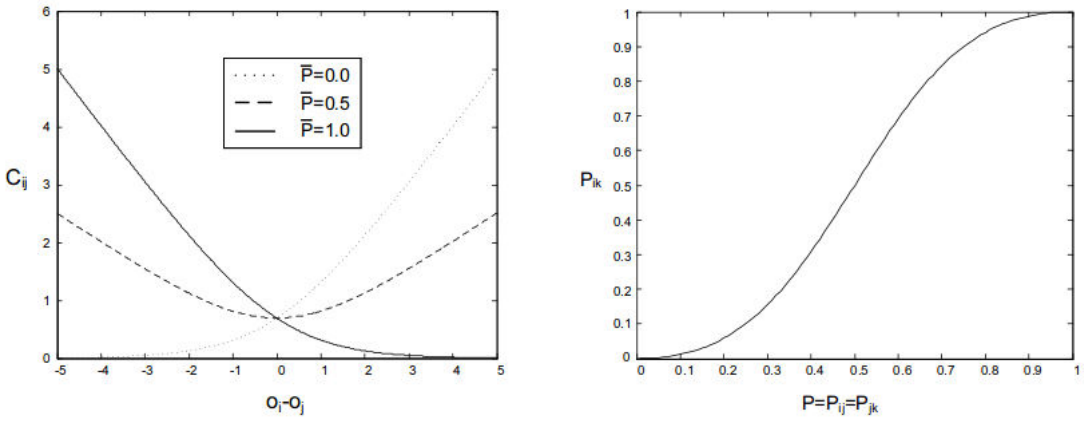

Figure 1. Left: the cost function, for three values of the target probability. Right: combining probabilities

For RankNet , the underlying model can be any model for which the output of the model is a differentiable function of the model parameters (typically we have used neural nets, but we have also implemented RankNet using boosted trees, which we will describe below).Let U i ►U j denote the event that U i should be ranked higher than U j (for example, because U i has been

labeled 'excellent' while U j has been labeled 'bad' for this query; note that the labels for the same urls may be different for different queries). Also let s i = f (x i) and s j = f (x j).The two outputs of the model are mapped to a learned probability that U i should be ranked higher than U j via a sigmoid function, thus:

P i j ≡ P(U i► U j) ≡1 / (1 + e $^{-\sigma (s i - s j)}$) _____(6)

Combining equations 1 and 6 we get:

C = ½ (1 − S i j)σ (s i − s j) + log (1 + $e^{-\sigma (s i - s j)}$) _____ (7)

∂C / ∂ s i= σ (½ (1 − S i j) – 1/(1 + $e^{\sigma (s i - s j)}$))= −∂C/∂ s j _____ (8)

This gradient is used to update the weights w k ∈ R (i.e. the model parameters) so as to reduce the cost via stochastic gradient descent 1 :

$$w_k \rightarrow w_k - \eta \frac{\partial C}{\partial w_k} = w_k - \eta \left(\frac{\partial C}{\partial s_i} \frac{\partial s_i}{\partial w_k} + \frac{\partial C}{\partial s_j} \frac{\partial s_j}{\partial w_k} \right)$$ _____(9)

where η is a positive learning rate (a parameter chosen using a validation set; typically, in our experiments, 1e-3 to 1e-5). Explicitly:

$$\delta C = \sum_k \frac{\partial C}{\partial w_k} \delta w_k = \sum_k \frac{\partial C}{\partial w_k} \left(-\eta \frac{\partial C}{\partial w_k} \right) = -\eta \sum_k \left(\frac{\partial C}{\partial w_k} \right)^2 < 0$$ _____(10)

The idea of learning via gradient descent is a key idea that appears throughout this paper (even when the desired cost doesn't have well-posed gradients, and even when the model (such as an ensemble of boosted trees) doesn't have differentiable parameters): to update the model, we must specify the gradient of the cost with respect to the model parameters w k , and in order to do that, we need the gradient of the cost with respect to the model scores s i . The gradient descent formulation of boosted trees (such as MART [8]) bypasses the need to compute ∂C/∂ w k by directly modeling ∂C/∂ s i.

Speeding up the RankNet training:

The above leads to a factorization that is the key observation that led to LambdaRank for a given pair of urls U i , U j (again, summations over repeated indices are assumed):

$$\begin{aligned} \frac{\partial C}{\partial w_k} &= \frac{\partial C}{\partial s_i} \frac{\partial s_i}{\partial w_k} + \frac{\partial C}{\partial s_j} \frac{\partial s_j}{\partial w_k} = \sigma \left(\frac{1}{2}(1 - S_{ij}) - \frac{1}{1 + e^{\sigma (s_i - s_j)}} \right) \left(\frac{\partial s_i}{\partial w_k} - \frac{\partial s_j}{\partial w_k} \right) \\ &= \lambda_{ij} \left(\frac{\partial s_i}{\partial w_k} - \frac{\partial s_j}{\partial w_k} \right) \end{aligned}$$

$$\lambda_{ij} \equiv \frac{\partial C(s_i - s_j)}{\partial s_i} = \sigma\left(\frac{1}{2}(1 - S_{ij}) - \frac{1}{1 + e^{\sigma(s_i - s_j)}}\right) \quad \text{(11)}$$

Note that since RankNet learns from probabilities and outputs probabilities, it does not require that the urls be labeled; it just needs the set I, which could also be determined by gathering pairwise preferences. Now summing all the contributions to the update of weight wk gives

$$\delta w_k = -\eta \sum_{\{i,j\} \in I} \left(\lambda_{ij} \frac{\partial s_i}{\partial w_k} - \lambda_{ij} \frac{\partial s_j}{\partial w_k}\right) \equiv -\eta \sum_i \lambda_i \frac{\partial s_i}{\partial w_k}$$

2) LambdaRank:

Although RankNet can be made to work quite well with the above measures by simply using the measure as a stopping criterion on a validation set, we can do better. The idea of writing down the desired gradients directly, rather than deriving them from a cost, is one of the ideas underlying LambdaRank: it allows us to bypass the difficulties introduced by the sort in most IR measures. The key observation of LambdaRank is thus that in order to train a model, we don't need the costs themselves: we only need the gradients (of the costs with respect to the model scores). The arrows (λ 's) mentioned above are exactly those gradients.

RankNet uses a neural net as its function class. Feature vectors are computed for each query/document pair. RankNet is trained on those pairs of feature vectors, for a given query, for which the corresponding documents have different labels. At runtime, single feature vectors are fpropped through the net, and the documents are ordered by the resulting scores. The RankNet cost consists of a sigmoid (to map the outputs to [0, 1]) followed by a pair-based cross entropy cost. Training times for RankNet thus scale quadratically with the mean number of pairs per query, and linearly with the number of queries. The ideas proposed in paper suggest a simple method for significantly speeding up RankNet training, making it also approximately linear in the number of labeled documents per query, rather than in the number of pairs per query. This is a very significant benefit for large training sets. In fact the method works for any ranking method that uses gradient descent and for which the cost depends on the pairs per query. This is a very very significant benefit for large training sets. In fact the method works for any ranking method that uses gradient descent and for which the cost depends on pairs of items for each query. Most neural net training, RankNet included, uses a stochastic gradient update which is known to give faster convergence. However here we will use batch per query (that is, the weights are updated for each query). We present the idea for a general ranking function f: R^n -> R with optimization cost $C : \mathbb{R}^n \rightarrow$ R. It is important to note that adopting batch training alone does not give a speedup: to compute the cost and its gradients we would still

need to fprop each pair.

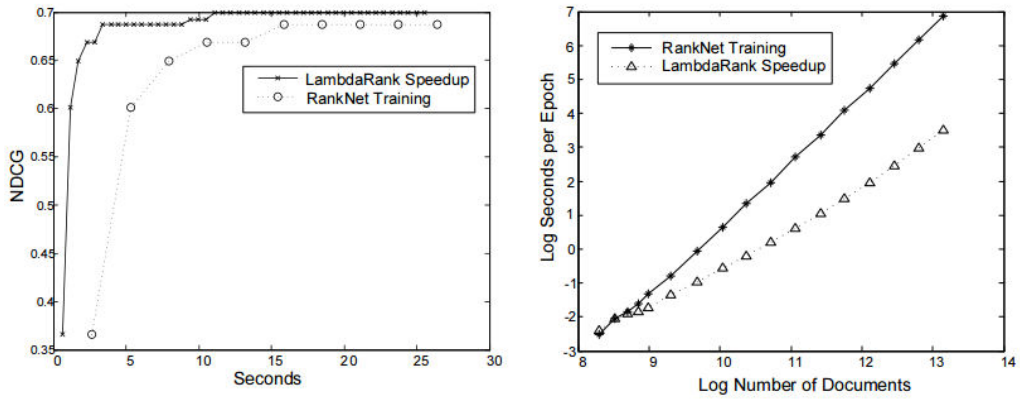

Figure 1: Speeding up RankNet training. Left: linear nets. Right: two layer nets.

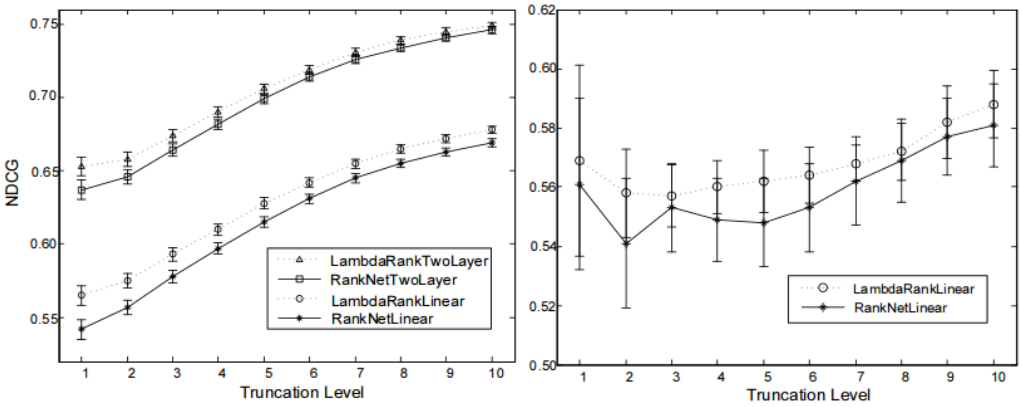

Figure 2: Left: Cubic polynomial data. Right: Intranet search data.

The key observation of LambdaRank is thus that in order to train a model, we don't need the costs themselves: we only need the gradients (of the costs with respect to the model scores). The arrows (λ 's) mentioned above are exactly those gradients. The λ 's for a given URL U 1 get contributions from all other URLs for the same query that have different labels. The λ 's can also be interpreted as forces (which are gradients of a potential function, when the forces are conservative): if U 2 is more relevant than U 1 , then U 1 will get a push downwards of size |λ | (and U 2 , an equal and opposite push upwards); if U 2 is less relevant than U 1 , then U 1 will get a push upwards of size |λ | (and U 2 , an equal and opposite push downwards). Experiments have shown that modifying Eq. (11) by simply multiplying by the size of the change in NDCG (|Δ NDCG |) given by swapping the rank positions of U 1 and U 2 (while leaving the rank positions of all other urls unchanged) gives very good results . Hence in LambdaRank we imagine that there is a utility C such that :

$$\lambda_{ij} = \frac{\partial C(s_i - s_j)}{\partial s_i} = \frac{-\sigma}{1 + e^{\sigma(s_i - s_j)}} |\Delta_{NDCG}| \qquad (12)$$

Since here we want to maximize C, equation (2) (for the contribution to the kth weight) is replaced by

$$w_k \rightarrow w_k + \eta \frac{\partial C}{\partial w_k}$$

so that

$$\delta C = \frac{\partial C}{\partial w_k} \delta w_k = \eta \left(\frac{\partial C}{\partial w_k} \right)^2 > 0 \qquad \text{(13)}$$

For a given pair, a λ is computed, and the λ 's for U 1 and U 2 are incremented by that λ , where the sign is chosen so that s 2 − s 1 becomes more negative, so that U 1 tends to move up the list of sorted urls while U 2 tends to move down. Again, given more than one pair of urls, if each url U i has a score s i , then for any particular pair {U i ,U j } (recall we assume that U i is more relevant than U j), we separate the calculation:

$$\delta s_i = \frac{\partial s_i}{\partial w_k} \delta w_k = \frac{\partial s_i}{\partial w_k} \left(-\eta \lambda \frac{\partial s_i}{\partial w_k} \right) < 0$$

$$\delta s_j = \frac{\partial s_j}{\partial w_k} \delta w_k = \frac{\partial s_j}{\partial w_k} \left(\eta \lambda \frac{\partial s_j}{\partial w_k} \right) > 0$$

Thus, every pair generates an equal and opposite λ , and for a given url, all the lambdas are incremented (from contributions from all pairs in which is it a member, where the other member of the pair has a different label). This accumulation is done for every url for a given query; when that calculation is done, the weights are adjusted based on the computed lambdas using a small (stochastic gradient) step.

3) LambdaMART:

LambdaMART combines MART [8] and LambdaRank. To understand LambdaMART we first review MART. Since MART is a boosted tree model in which the output of the model is a linear combination of the outputs of a set of regression trees, we first briefly review regression trees. Suppose we are given a data set {x i , y i }, i = 1, . . . , m where x i ∈ R^d and the labels y i ∈ R. For a given vector x i we index its feature values by x i j , j = 1, . . . , d. Consider first a regression stump, consisting of a root node and two leaf nodes, where directed arcs connect the root to each leaf. We think of all the data as residing on the root node, and for a given feature, we loop through all samples and find the threshold t such that, if all samples with x i j ≤ t fall to the left child node, and the rest fall to the right child node, then the sum is minimized. Here L (R) is the sets of indices of samples that fall to the left (right), and μ L (μ R) is the mean of the values of the labels of the set of samples that fall to the left (right). (The dependence on j in the sum appears in L, R and in μ L , μ R). S j is then computed for all choices of feature j and all choices of threshold for that feature, and the split (the choice of a particular feature j and threshold t) is chosen that gives the overall minimal S j . That split is then attached to the root node. For the

two leaf nodes of our stump, a value γ l , l = 1, 2 is computed, which is just the mean of the y's of the samples that fall there. In a general regression tree, this process is continued L − 1 times to form a tree with L leaves.

$$S_j \equiv \sum_{i \in L} (y_i - \mu_L)^2 + \sum_{i \in R} (y_i - \mu_R)^2$$

MART is a class of boosting algorithms that may be viewed as performing gradient descent in function space, using regression trees. The final model again maps an input feature vector x ∈ R d to a score F(x) ∈ R. MART is a class of algorithms, rather than a single algorithm, because it can be trained to minimize general costs (to solve, for example, classification, regression or ranking problems). Note, however, that the underlying model upon which MART is built is the least squares regression tree, whatever problem MART is solving. MART's output F(x) can be written as:

$$F_N(x) = \sum_{i=1}^{N} \alpha_i f_i(x)$$

where each f i (x) ∈ R is a function modeled by a single regression tree and the α i ∈ R is the weight associated with the ith regression tree. Both the f i and the α i are learned during training. A given tree f i maps a given x to a real value by passing x down the tree, where the path (left or right) at a given node in the tree is determined by the value of a particular feature x j , j = 1, . . . , d and where the output of the tree is taken to be a fixed value associated with each leaf, γ kn , k = 1, . . . , L, n = 1, . . . , N, where L is the number of leaves and N the number of trees. Given training and validationsets, the user-chosen parameters of the training algorithm are N, a fixed learning rate η (that multiplies every γ kn for every tree), and L. (One could also choose different L for different trees). The γ kn are also learned during training.

To implement LambdaMART we just use MART, specifying appropriate gradients and the Newton step. The gradients γ^- i are easy: they are just the λ i . As always in MART, least squares is used to compute the splits. In LambdaMART, each tree models the λ i for the entire dataset (not just for a single query). To compute the Newton step, we collect some results from above: for any given pair of urls U i and U j such that U i U j , then after the urls have been sorted by score, the λ i j 's are defined as:

$$\lambda_i = \sum_{j:\{i,j\} \in I} \lambda_{ij} - \sum_{j:\{j,i\} \in I} \lambda_{ij}$$

Algorithm: LambdaMART

set number of trees N, number of training samples m, number of leaves per tree L, learning rate η

for $i = 0$ to m **do**

  $F_0(x_i) = \text{BaseModel}(x_i)$ //If BaseModel is empty, set $F_0(x_i) = 0$

end for

for $k = 1$ to N **do**

  for $i = 0$ to m **do**

    $y_i = \lambda_i$

    $w_i = \frac{\partial y_i}{\partial F_{k-1}(x_i)}$

  end for

  $\{R_{lk}\}_{l=1}^{L}$ // Create L leaf tree on $\{x_i, y_i\}_{i=1}^{m}$

  $\gamma_{lk} = \frac{\sum_{x_i \in R_{lk}} y_i}{\sum_{x_i \in R_{lk}} w_i}$ // Assign leaf values based on Newton step.

  $F_k(x_i) = F_{k-1}(x_i) + \eta \sum_l \gamma_{lk} I(x_i \in R_{lk})$ // Take step with learning rate η.

end for

Finally it's useful to compare how LambdaRank and LambdaMART update their parameters. LambdaRank updates all the weights after each query is examined. The decisions (splits at the nodes) in LambdaMART, on the other hand, are computed using all the data that falls to that node, and so LambdaMART updates only a few parameters at a time (namely, the split values for the current leaf nodes), but using all the data (since every x i lands in some leaf). This means in particular that LambdaMART is able to choose splits and leaf values that may decrease the utility for some queries, as long as the overall utility increases.

Shortcomings of the paper1:

1) No proofs, no experimental results are presented in the paper for RankNet/ LambdaRank/ LambdaMART. (I have included the experimental results in the above paper)
2) No comparison to any other methods(other than the given three methods) for learning to rank
3) One approach is not considered – Pointwise Approach.
   Pointwise approach: In this case it is assumed that each query-document pair in the training data has a numerical or ordinal score. Then learning-to-rank problem can be approximated by a regression problem — given a single query-document pair, predict its score.

Ideas for how to extend the work:

[5] contains a relatively simple method for linearly combining two rankers such that out of all possible linear combinations, the resulting ranker gets the highest possible NDCG (in fact this can be done for any IR measure). The key trick is to note that if s^1_i are the out- puts of the first ranker (for a given query), and s^2_i those of the second, then the output of the combined ranker can be written $\alpha\, s^1_i + (1 - \alpha)\, s^2_i$, and as α sweeps through its values (say, from 0 to 1), then the NDCG will only change a finite number of times. We can simply (and quite efficiently) keep track of each change and keep the α that gives the best combined NDCG. However it should also be noted that one can achieve a similar goal (without the guarantee of optimality, however) by simply training a linear (or nonlinear) LambdaRank model using the previous model scores as inputs. This usually gives equally good results and is easier to extend to several models.

Paper2: "From RankNet to LambdaRank to LambdaMART: An Overview"

Main Contribution of the Paper:

In this paper, the authors have constructed a combined learning to rank framework by integrating a general ranking model with the query-biased model that takes the query differences into account. In their proposed combined framework, the general ranking model is learned from training queries by the conventional learning to rank approach. In addition, they propose to learn a query-biased ranking model by a transductive learning algorithm(given later) based on the pseudo relevance information. Finally, the query-biased model is combined with the general model to produce the final tweet ranking for the target queries.

So there are the following two main contributions of this paper as follows:

1) The authors have propose a semi-supervised learning algorithm to build a query-biased ranking model. Instead of treating all queries equally, they train a query-biased model based on the queries intrinsic features through an iterative learning process.
2) They also propose a learning to rank framework to combine the query-biased model with a general ranking model learned from the common features.

To test the above two main contributions, extensive experiments on the standard TREC Tweets11 collection demonstrate the effectiveness of our proposed query-biased learning to rank approach.

Methods Proposed by the authors:

There is one Combined Learning to Rank Framework that is proposed by the authors which is divided into the two following parts:

1) General Ranking Model: Learned from the training instances, represented by the features common to different queries.
2) Query-Biased Model: Learned from the query-specific features by a semi-supervised learning algorithm.

The above two models are integrated by a linear combination as follows:

$$\text{Score final}(d, Q) = \text{Score LT R}(d, Q) + \beta \cdot \text{Score QLT R}(d, Q)$$

Where Score final (d, Q) is the final score of tweet d for the given query Q; Score LT R (d, Q) is the score given by the general ranking model; Score QLT R (d, Q) is the score given by the query-biased model. The setting of the parameter β is obtained by training on different folds of cross-validation.

1) General Ranking Model
   a) Tweet Features Extraction: The features are organized around the basic entities for each query tweet tuple to distinguish between the relevant and irrelevant messages.
   b) Learning to Rank Algorithms: The author takes into consideration RankSVM(Pairwise approach) and LambdaMART(Listwise approach) into consideration.
2) Query-biased Ranking Model – Following are presented the two main algorithms for query base ranking models
   a) Query-specific Tweet Representation - The tweet representation algorithm.
      Figure1:

Input

D: initial retrieved tweets returned by a content-based retrieval model for a batch of query topics
N: the maximum number of terms to be selected to represent a document

Output

W: the selected term set

Method

Do the following for each query:
(1) Get R, the top-k ranked tweets from D
(2) Compute the KL divergence weight for each unique term in R
(3) Extract the N terms with the highest weights as the term features
(4) Represent each tweet retrieved for the given query with the selected terms and their KL divergence weights

All the unique terms in the top-30 tweets are taken as candidate terms, and the 10 terms with highest KL divergence weights are chosen as the query-specific features. Thus, the selected words and their corresponding KL diver- gence weights are used as attributes and values to represent the given tweets. Our arbitrary choice of selecting the top-10 terms from the top-30 tweets is mainly due to the fact that this setting was found to provide the best query expansion effectiveness in the TREC 2011 Microblog track, as reported in [1]. The KL divergence weight of a candidate term t in the top-k ranked tweets in the initial retrieval is computed as follows:

$w(t, Rk) = P(t|Rk) \log_2 P(t|Rk) / P(t|C)$

where $P(t|Rk)$ is the probability of generating the candidate term t from the set of top-k ranked tweets Rk , and $P(t|C)$ is the probability of generating t from the entire collection C.

b) Transduction for Query-biased Modeling - The query-biased transductive learning algorithm. Figure2:

Input

D: initial retrieved tweets returned by content-based retrieval models for a batch of target queries
U: a set of unlabeled tweets
I: a set of labeled tweets, if any
F: a set of term features representing the tweets
k: the number of top-ranked and bottom-ranked tweets to be initially added to I, if I is empty
IN: the number of iterations

Output

L: a ranked list for each query in the target queries

Method

If I is empty, add the top and bottom-k tweets in D into I as positive and negative examples, respectively

1. Do the following for IN iterations
   (1) Learn a ranking model M from I using F to represent the tweets
   (2) Use M to rank tweets in U represented by F
   (3) For all the topics in the training dataset, select the very top-ranked and bottom-ranked tweets T from U
   (4) Add T to I and remove T from U
2. Get the best effectiveness and get the number of iterations for the batch of queries or per-query in the test set
3. Gradually add the labeled examples for the query in the test set
4. Use the generated data to train a model M
5. Return a ranked list for each of the target queries by applying M

Transductive learning [2] is a semi-supervised method for classification by utilizing limited data using transduction.

Moreover, The authors have defined two variants of the query-biased learning as follows:

i) Per-query learning: for each query, a query-biased model is learned with the unique term features. Thus, it is able to distinguish the special aspects of the individual target queries. This is equivalent to the algorithm in Figure 2 when there is only one test query. However, this method may suffer from the problem of sparseness since the learning is based only on the pseudo relevance set of single queries.

ii) Batch learning: Instead of building the query-biased modelfor each target query, this method treats the selected highly weighted terms of different queries as the

common features for a batch of target queries. As this method takes the pseudo relevance sets of a batch of queries as the training data, it is expected to be able to deal with the sparseness problem.

Evaluation on how did the authors present a convincing case:

The authors present a convincing case by doing experiments on TREC Tweets11 collection, which is used for evaluating the participating real-time Twitter search systems over 50 official topics in the TREC 2011 Microblog track.They were successfully able to crawl 13,401,964 unique tweets.

The choice of the hyperparameter C, the regularization parameter for RankSVM, plays an important role in the retrieval performance. C can be selected by carrying out cross-validation experiments through a grid search from 1 to 40 on the training queries.

Parameter.	Values
Max Number of Leaves	2, 3, 5, 7, 10
Min Percentage of Obs. per Leaf	0.05, 0.12, 0.15, 0.75
Learning rate	0.01, 0.04, 0.045, 0.05, 0.07, 0.2
Sub-sampling rate	0.05, 0.1, 0.3, 0.5, 0.7, 1
Feature Sampling rate	0.1, 0.3, 0.5, 0.7, 1

The LambdaMART algorithm has five parameters(given above) that need to be tuned to achieve the best results. They applied a greedy boosting algorithm for the parameter tuning. Step 1 is to optimize each of the five parameters, while keep the other four parameters to the default values. Then, they set the parameter that has the highest optimized retrieval performance to its optimized value. This process repeats until all the parameters are optimized.

The aim of the experiments is to evaluate the effectiveness of the proposed learning to rank framework which combines the general learning to rank approach and transductive query-biased method for real time Twitter search. In particular, two state-of-the-art learning to rank approaches, namely Ranking SVM (RankSVM) [3], a classical pair wise learning to rank algorithm and LambdaMART [31], a tree-based listwise learning to rank algorithm.

They compare their proposed combined learning to rank framework, denoted as CombLTR, to the general ranking model (LTR), which represents the conventional application of the state-of-the-art learning to rank algorithms. Using the LTR approach, they conducted experiments in different granularities of the partitioning of the train-test topics, namely the 2-fold (LTR 2) 5-fold (LTR 5),10-fold (LTR 10) and leave-one-out (LTR L1) cross-validations, respectively. Cross-validation is a technique to estimate the performance of a predictive model. In K-fold cross-validation, the 50 topics is partitioned equally into K subsets. Among them, one subset is chosen to test the model, while the others are used for training.

The results summary:

RankSVM			LambdaMART		
Leave-one-out					
LTR_L1	CombLTR_L1_Batch	CombLTR_L1_PerQ	LTR_L1	CombLTR_L1_Batch	CombLTR_L1_PerQ
0.4016	0.4020, 0.10%	0.4020, 0.10%	0.3878	**0.4116, 6.14%**∗	**0.4116, 6.14%**∗
10-fold					
LTR_10	CombLTR_10_Batch	CombLTR_10_PerQ	LTR_10	CombLTR_10_Batch	CombLTR_10_PerQ
0.4061	0.4218, 3.87%	0.4116, 1.35%	0.3986	0.4143, 3.94%	**0.4211, 5.64%**∗
5-fold					
LTR_5	CombLTR_5_Batch	CombLTR_5_PerQ	LTR_5	CombLTR_5_Batch	CombLTR_5_PerQ
0.3980	**0.4313, 8.37%**∗	0.4109, 3.24%	0.3980	0.4102, 3.07%	0.4027, 1.81
2-fold					
LTR_2	CombLTR_2_Batch	CombLTR_2_PerQ	LTR_2	CombLTR_2_Batch	CombLTR_2_PerQ
0.3816	**0.4150, 8.75%**∗	0.3912, 2.52%	0.3946	0.3932, -0.35%	0.4007, 1.55%

From the above observation, the authors showed that CombLTR Batch significantly out performs RankSVM on 5 and 2-fold cross-validations, and outperform LambdaMART on the 10-fold cross-validation; while CombLTR_PerQ outperforms LambdaMART on 10-fold and leave-one-out cross-validations. Comparing to Rank- SVM, CombLTR_PerQ shows no notable improvement. In addition, a surprising observation is that, using RankSVM, CombLTR has better performance with fewer training data available, i.e. during the 2-fold and 5-fold cross-validations. The opposite is observed when using LambdaMART. They suggest that this is possibly due to the fact that RankSVM has a better ability in dealing with the sparseness problem than LambdaMART in the learning process. Besides, the granularity of cross-validation is believed as a factor affecting the retrieval effectiveness, just as Table 2 shows that CombLTR_Batch outperform RankSVM on 5 and 2-fold cross-validations, while CombLTR_PerQ outperforms LambdaMART on 10-fold and leave-one-out cross-validations.

Shortcomings of the paper2:

1) There is no explanation of RankSVM and how it works internally.
2) There is no explanation of LambdaMART and how it works internally. This we have covered in the in depth understanding of LambdaMART in paper 1.
3) The robustness of the proposed approach can be improved by introducing an adaptive halting criterion of the iterative process.

Ideas for how to extend the work:

The typical adaptive halting algorithm is essentially a simple four-step procedure: solve–estimate–mark–refine. Namely, given a subdivision of the computational domain, one solves the finite element problem (which involves the assembly of the stiffness matrix and the solution of the resulting linear system), then computes an a posteriori error estimate; based on this estimate, a marking strategy is applied to identify for further refinement a set of elements in the current subdivision. Typically, it is assumed that the solution of the linear systems on each level is exact; with a few exceptions. For large problems, especially in three dimensions, the exact solution of the linear systems may not be obtained efficiently, if at all, by using sparse direct solvers. In such cases, a competitive, or only, alternative is provided by

iterative linear solvers, such as the Conjugate Gradient (CG) method, which compute approximations to the exact finite element solution at each iteration of the linear solver. The key issue in this context is accuracy: a highly accurate approximation of the solution to the linear system at each level is inefficient and, most likely, unnecessary, while a poor approximation may affect the quality of the a posteriori estimators, thereby yielding different, possibly extensive refinements. The latter, in turn, affects the quality of the solution at the next step and, potentially, the convergence of the adaptive algorithm itself.

Summary for the two papers:

Paper1: "From RankNet to LambdaRank to LambdaMART: An Overview"
We have studied simple and effective methods for learning to rank. LambdaRank is a general approach: in particular it can be used to implement RankNet training, and it furnishes a significant training speedup there. We studied LambdaRank in the context of NDCG target costs.

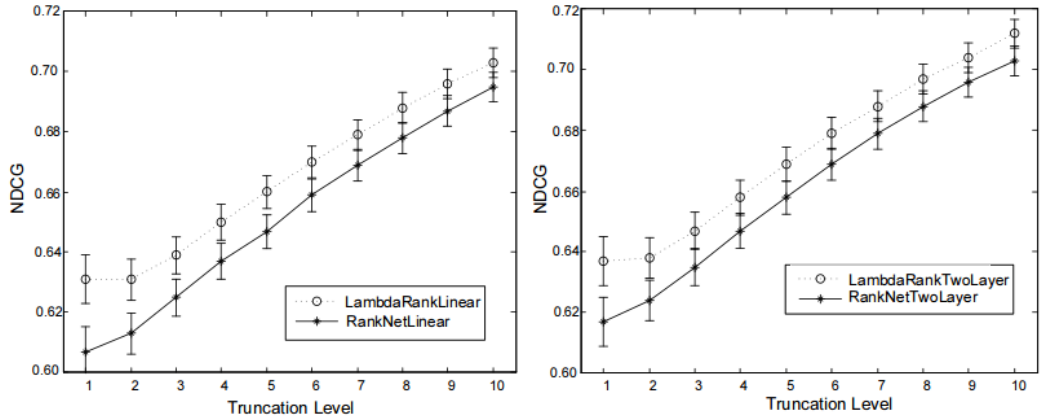

Figure 3: NDCG for RankNet and LambdaRank. Left: linear nets. Right: two layer nets

LambdaMART inherits significant advantages from both MART and LambdaRank. It has the flexibility and the interpretability of boosted trees, and we have shown that replacing the first tree with a previously trained model significantly improves accuracy for the model adaptation problem. From LambdaRank it inherits the property of direct optimization of the IR measure at hand, and in addition produces models that have significantly better behavior regarding the speed/accuracy tradeoff.

Paper2: "Query-biased Learning to Rank for Real-time Twitter Search"
The authors proposed method -> combined learning to rank framework, has significantly improved the retrieval effectiveness when comparing with the content-based retrieval models and the popular Learning to rank approaches like RankSVM and LambdaMART individually. In addition, despite the noise during the learning of the query-biased ranking model, a series of experiments on Tweets11 demonstrate the benefit brought by the method put forward in this paper, especially when there is only limited amount of training queries available.

REFERENCES

[1] G. Amati, G. Amodeo, M. Bianchi, A. Celi, C. D. Nicola, M. Flammini, C. Gaibisso, G. Gambosi, and G. Marcone. Fub, iasi-cnr, univaq at trec 2011. In TREC, 2011.

[2] V. N. Vapnik. Statistical learning theory. New York: Wiley, 1998.

[3] T. Joachims. Optimizing search engines using clickthrough data. In KDD, pages 133–142. ACM, 2002.

[4] Chin-Tien Wu ∗ and Howard C. Elman. Convergence Criteria For Iterative Methods In Solving Convection-Diffusion Equations On Adaptive Meshes.

[5] Q. Wu, C.J.C. Burges, K. Svore and J. Gao. Adapting Boosting for Information Retrieval Measures. Journal of Information Retrieval, 2007.