Improving LinkedIn Search Using Learn-to-Rank Models

Ganesh Venkataraman
Senior Software Engineer, LinkedIn
Email: gvenkataraman@gmail.com

***Abstract*—Social network search is personalized. In traditional web search, results are a function of query and documents. In social networks, it is a function of query, document and user. Relevancy of the result set could vary quite a bit based on the user's network, profile information etc. Traditional Learn-to-Rank techniques have focused for most part on result set as a function of query and documents. In this work, we will examine results as a function of (query, document, user). We will take a specific set of queries, collect training data and run them on different learn-to-rank models. We will examine ways to avoid position bias and draw insights into why some models perform better than others, when do we over fit/under fit etc.**

***Index Terms*—Search Ranking, Learn to Rank, Machine Learning**

I. Introduction

LinkedIn is the largest professional network serving over 248 million users worldwide. Users visit the site for various reasons including connecting with other people, finding jobs, recruiting candidates, reading news, reading professional content etc. As the user base increases and the number of possibilities of using the site increase, search becomes the central to finding what users want. Search is also central for users to "be found" in LinkedIn [8].

Traditional information retrieval methods focus on ranking as a function of *(query, document)*. This does not always work with social networks. The query *developer* could have very different meaning depending on whether you work in real estate or software engineering. To solve the ambiguity, we could perform:

1) Explicit disambiguation like *Wikipedia*.
2) Diversity in top 10 result set like *Google*.
3) Make an educated guess on the user intent and serve the appropriate results

The third approach is well suited for LinkedIn. Using the user specific data, one could find the intent of the user and rank the results based on intent. Hence ranking in social networks is a function of *query, documents* and ***user***.

User searches can fall into two broad categories - *navigational* and *exploratory*. Navigational searches are those where the information need of the user can only be met by a single result or in other words there is only one right result. Example, when searching for "john smith Microsoft", the user probably is searching for a specific "john smith" who works for Microsoft. On the other hand, a query like "java" may mean multiple things. The user may be looking to recruit someone good at Java or connect with someone who is good at Java or something else. In case of such queries, non-personalization features also start playing an important role. In essence, our feature set must contain a mix of both personalization and non-personalization features.

Over the last decade or so, learn to rank models have started becoming very popular in both academia [5] and industry [1]. A good survey of more recent work can be found in [1].

Learn to rank techniques are already in use in production at LinkedIn [8]. LinkedIn allows users to add skills to their profile. Typical example of skills for a software engineer would be - "Java", "map reduce", "scrum" etc. Users can also *endorse* skills for other users in their network. Skills are an integral part of a user's profile. It also helps recruiters find users with specific skills. For purposes of this work, we will restrict ourselves to searches where at least one keyword is a skill. The contributions of this work are as follows:

1) We will detail how we obtained the truth data for training.
2) We will run a set of machine learning algorithms and perform analysis on their performance.
3) We will report the relative performance for each of the techniques tried and draw insights into why one performed better/worse than the other.

Note that we have not mentioned feature engineering. Though feature engineering was done for this project, we cannot disclose information about the features and hence it is not mentioned as part of the contributions. The remainder of the report is organized as follows: Section II deals with a brief survey of learn-to-rank techniques and their challenges. Section III talks about how to collect specific training data to address some of the challenges pointed out in section II. We will present the experimentation framework and insights in section IV, present results in section V and draw conclusions in section VI.

A. Disclaimers

1) Please do **not** make this report public.
2) The report should not be interpreted as the current state-of-art at LinkedIn. The live production code may use completely different set of techniques. This includes different set of features, model as well as evaluation metrics. This work is exploratory - but based on real search data.
3) We cannot release exact feature weights and/or any production metrics.

II. Preliminaries

A. Conventions

Unless explicitly stated otherwise, the following conventions will be used in the paper.

- d_i refers to retrieved document i.
- q refers to query.
- u refers to the user.
- Φ refers to the ranking function.
- l_i refers to relevance label assigned to a particular result set for a particular query. These can be done using human judgment or derived using click data.

B. Information Retrieval Measures

We will establish certain basic ranking measures which will be used in training and/or testing the result sets. CTR@1 refers to the percentage of queries where the first retrieved result was clicked. Reciprocal Rank (RR) refers to the reciprocal of the rank of the least clicked result. For example, if the 5^{th} result were clicked, RR equals 0.20. Mean Reciprocal Rank refers to the mean of the RR's over the enter sample set. Let l_i refer to relevance label. DCG (Discounted Cumulative Gain) [1] for a given set of search results is given by:

$$DCG@T = \sum_{i=1}^{m} \frac{2^{l_i} - 1}{log(1+i)} \tag{1}$$

T refers to the truncation level for the query. Typically T = 10. NDCG is the normalized version given by:

$$NDCG@T = \frac{DCG@T}{maxDCG@T} \tag{2}$$

maxDCG@T denotes the maximum DCG@T attainable.

C. Pairwise Approach

Training data for learn to rank models can be collected using click through data (typically from search logs) or from manual judgment. We will focus on the former in this project. One of the first work on optimizing using clickthrough data was published by Jorchims [5]. Assume that we have a result set shown to a user with documents *(d_1, d_2, ..., d_n)*. Assume that document d_3 is clicked. Let $\Phi(q, d, u)$ represent a ranking function. Ranking is a function of query q, document d and user u. This can be taken as an implicit judgment that $\Phi(q_k, d_3, u_l) > \Phi(q_k, d_1, u_l)$ and similar relations for (d_3, d_2), (d_3, d_4) etc. Let the ranking function Φ be parametrized by $\vec{w}$. To make things explicit, we will refer to the ranking function as $\Phi(q, d, u, \vec{w})$. Let P represent the set of such pairs. In general, pairwise ranking approaches tries to build a ranking function that satisfies a set of constraints:

$$(q_k, u_l, d_i, d_j) \epsilon P : \quad \Phi(q_k, d_i, u_l, \vec{w}) > \Phi(q_k, d_j, u_l, \vec{w}) \tag{3}$$

One direct approach is to maximize the number of constraints in equation (3) to be satisfied. This turns out to be NP-hard [3]. We will approximate the solution by introducing slack variables $\xi_{i,j,k,l}$ for each constraint in equation (3). Slack constraints transform it into a ranking SVM shown below:

$$\begin{aligned} minimize: \ V(\vec{\xi}, \vec{w}) &= \frac{1}{2}\vec{w}\vec{w}^t + C\sum \xi_{i,j,k,l} \\ s.t. \ \Phi(q_k, d_i, u_l, \vec{w}) &> \Phi(q_k, d_j, u_l, \vec{w}) + 1 - \xi_{i,j,k,l} \forall P \\ \xi_{i,j,k,l} &\geq 0 \end{aligned} \tag{4}$$

Equation (4) can be re-written in standard SVM format by rearranging the constraints as:

$$\Phi(q_k, d_i, u_l, \vec{w}) - \Phi(q_k, d_j, u_l, \vec{w}) \geq 1 - \xi_{k,l,i,j} \tag{5}$$

This transforms into standard SVM used for classification and we could use several libraries for implementation [6]. Note that that equation(3) can also be cast as logistic regression problem with result $\Phi(q_k, d_i, u_l, \vec{w})$ assigned label '1' and $\Phi(q_k, d_j, u_l, \vec{w})$ assigned label '0'.

D. Listwise Approach

Listwise approaches use the same training data as pairwise approach but treat each list as an instance instead of explicitly generating the pairs [2]. Listwise approach offers the following advantages over pairwise approach:

1) Objective of training is to model a ranking function which dictated order of results. It is not the same as minimizing number of errors in classification.
2) Listwise transforms the problem from $O(n^2)$ complexity (pairs) to $O(n)$ for lists. This causes significant reduction in CPU time for training.
3) Listwise approaches do not make the assumption that the pairs are generated i.i.d.
4) The number of results in the retrieved set could vary a lot. This makes the feature space biased towards those queries with more results (more results $\implies$ more pairs).

To illustrate the issue with pairwise approach, consider the situation where a particular query gives 10 results. Assume results 1 and 10 where clicked. Pairwise approach would generate the following pairs to be optimized *(2, 10), (3, 10), ...(9, 10)* where each pair refers to a pair in equation (3). The objective here to find a function that minimizes the number of violations of equation (3). Consider the solution that moves first result to third and 10^{th} result to 5^{th}. This does indeed gives significant improvement from the original ranking function. But, it also brought down the first result - which will in turn bring down both CTR@1 and NDCG. Listwise approaches directly optimize the entire list using a probabilistic loss function.

We will describe in brief the loss function used for listwise approach. The proof of properties that lead to this loss function are beyond scope of current work. Interested reader may refer to [2]. For every (q, d, u), let the feature vector be x_j^i. Let $n^{(i)}$ represent number of results at query q_i. The list of features $x^i = (x_1^i, \ldots x_{n^i}^i)$ and $y^i = (y_1^i, \ldots y_{n^i}^i)$ form an instance. Let z^i represent the set of relevance scores. The loss function

we intent to compute is per instance as defined above. So, the objective would be to minimize:

$$\sum_{i=1}^{m} L(y^i, z^i) \tag{6}$$

Let π define a permutation of objects from $1, 2, \ldots n$. Let s_j denote the relevance score of document j. Let Ω_n represent the set of all possible permutations in n. Let $\phi(.)$ denote an increasing and strictly positive function. The probability of permutation given the scores is given by:

$$P_s(\pi) = \prod_{j=1}^{n} \frac{\phi(s_{\pi(j)})}{\sum_{k=j}^{n} \phi(s_{\pi(k)})} \tag{7}$$

The top one probability of object j is defined as:

$$P_s(j) = \sum_{\pi(1)=j, \pi \in \Omega_n} P_s(\pi) \tag{8}$$

The top one probability of object j is the sum of permutation probabilities of permutations in which object j is the first ranked result. We will state the following two theorems without proof.

Theorem 1.

$$P_s(j) = \frac{\phi(s_j)}{\sum_{k=1}^{n} \phi(s_k)} \tag{9}$$

Theorem 2. *Given any two object j and k, if $s_i > s_j$, $jneqk$, $j, k = 1, 2, \ldots n$, then $P_s(j) > P_s(k)$.*

$$P_s(j) = \frac{\phi(s_j)}{\sum_{k=1}^{n} \phi(s_k)} \tag{10}$$

Given above two theorems, if we use cross entropy as metric, the loss function would be:

$$L(y^i, z^i) = -\sum_{j=1}^{n} P_{y^i}(j) log(P_{z^i}(j)) \tag{11}$$

Exponential function was chose to represent $\phi(.)$.

III. Data collection

Collecting data for search must explicitly take care of three issues:

1) Position bias
2) Model inversion
3) Sampling bias

We shall describe in brief each of the above issues and ways we mitigated them in data collection. Users rarely look at the results lower (in the result page) than the clicked result. Further, users are heavily biased towards clicking results that are shown higher [7]. If result set is presented in order $(d_1, d_2, ..., d_{10})$ and d_3 is clicked, we cannot conclude that d_4 must have lower ranking score than d_3, d_3 may have been clicked due to position bias. One way of handling position bias is to ignore the relationship between d_3 and all results d_i where $i > 3$ [5]. This has a serious issue of model inversion. Let the original model that produced the result be $\Phi(q, d, u, \vec{w})$. If we feed the clickthrough ordering in equation (5) into the model ignoring the results *after* the clicked one, the optimal model would simply be $\Phi(q, d, u, -\vec{w})$. This is neither right nor desirable. Hence we need to explicitly specify some results that are strictly worse than the clicked ones as well. The third issue is sampling bias. If we feed the results from clicked data into the model, most of the results fed are from the first page (typically top 10). This implies results from page 5 (say) and their corresponding features are not represented in the training set. The training set will end up being an unfair representation of the feature space. We employed the following techniques to avoid the issues stated above.

A. Fair Pairs

Fair pairs avoids presentation bias by explicitly flipping the original relevance in pairs [7]. In the original result set, a pair (d_i, d_{i+1}) is flipped in presentation. For example, d_3 and d_4 could be flipped. If the user clicks on d_4 (now the higher result), it tells us nothing. However, if a click is made on d_3, then it points to the fact that d_3 is indeed a better result than d_4. We collected data by flipping pairs as described above to a small percentage of live production results. To minimize the impact on user experience, we never flipped the top most result. We collected features for the corresponding result set and noted the clicks.

B. Adding easy positives and easy negatives

As defined above, assume that d_3 is the clicked result. To avoid model inversion, we will pick a result which we are confident is worse than d_3 and add this to our training set. We shall make the assumption that the original model (that was used to collect the training data) is good enough that a tail result is indeed much worse than result in the top page. So, we pick result d_n where $n >> 10$ to be a strictly bad result and we shall call these easy negatives. We also hand curate some positive results. We construct a set of high confidence queries/results and add them to easy positives. Example, assume "John Smith" works for Microsoft and lives in Seattle. Let's say "John Smith" is connected to "John Doe" and among Doe's connections, there is only one "John Smith" who works for Microsoft and lives in Seattle. So, the query "John Smith Microsoft Seattle" issued by John Doe constitutes entry into training set for easy positives. Note that the introduction of easy negatives also gives a fair representation of the feature space.

C. Relevance labels for Listwise approaches

Listwise approaches detailed in [1] directly optimize NDCG. NDCG requires relevance labels to be applied to each (query, document, user). One of the best ways to do relevance labels is using human judgment. This is both prohibitively expensive and unscaleable. We will therefore employ the following workaround for relevance labels. Higher label indicates better relevance.

1) Clicked result has label 5.
2) Unclicked result which has original ranking lower than clicked result will have label 0.

3) Unclicked result which has original ranking higher than clicked result will have label 1.

The rationale for above labeling is as follows. LinkedIn result shows enough information in the result snippet that a clicked result is assumed to be good and hence label 5. Users typically go over queries in the order presented. So, if 5th result is clicked, then we can make a reasonable assumption that results 1 to 4 were *evaluated* by the user and found to be not useful and hence the label 0. Results ranked lower may or may not be looked. If we end up not giving any label to these results, it will result in model inversion as described earlier. So, we assign some reasonable non-zero label.

IV. Experimentation and Insights

A. Evaluation criteria

We will use the following evaluation metrics:

1) Training error. Example, for classification, this is measure (for classification) as number of mislabels samples.
2) Generalization error. We will use hold out cross validation where 70% of samples are used for training and 30% are used for validation.
3) The entire search result set (typically top 10) is re-ranked using the model. We will then compute CTR@1 and MRR. Using CTR@1 and MRR provides a uniform platform to compare results across different models. It also makes it much more relevant to the problem in hand.

B. Logistic Regression on pairwise data

As described in section III, we collect pairwise data using the fair pairs technique. The collected pairs can be marked as '0'/'1' and run through a classification algorithm like logistic regression. The loss function was the standard log likelihood. We used [6] library to implement logistic regression. Regularization was set to $C = 1.0$.

C. SVM on pairwise data

Pairwise problem can also be set as SVM (section III). Again, we used the scikit library to implement SVM. Here we tried a couple of options: regular linear SVM and SVM with kernel degree 3.

D. Listwise models using LamdaMART

We will use the listwise using gradient boosted trees as detailed in [9]. The actual implementation is done using the RankLib library [4].

V. Results

Figures (1) and (2) plot the training and test errors for three models we tried. The data is normalized, that is lowest sample size with logistic regression is treated as 1.0 and rest are computed based on this value. 70% samples were used to train and remaining 30% used for testing. SVM with degree 3 kernel showed the highest promise on training, but did not match this up on test. It indicates that SVM overfitted the data. We also have strong reason to believe that the data exhibits non-linear behavior. This is mainly due to the intuition behind the kind of features used.

Fig. 3. Normalized CTR@1

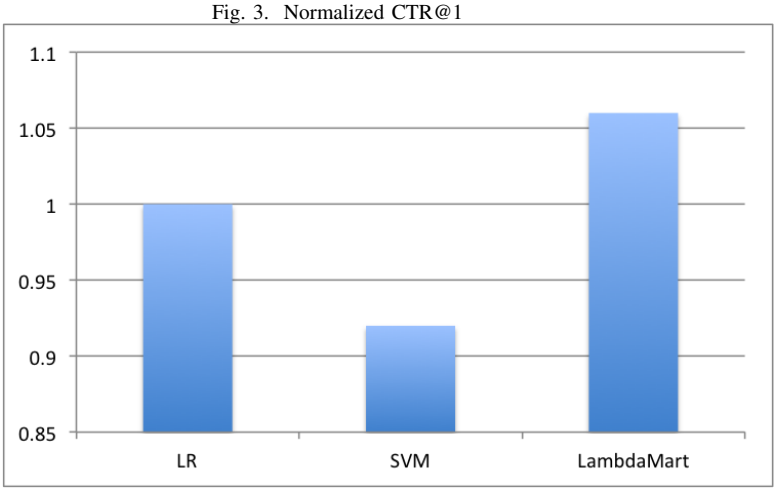
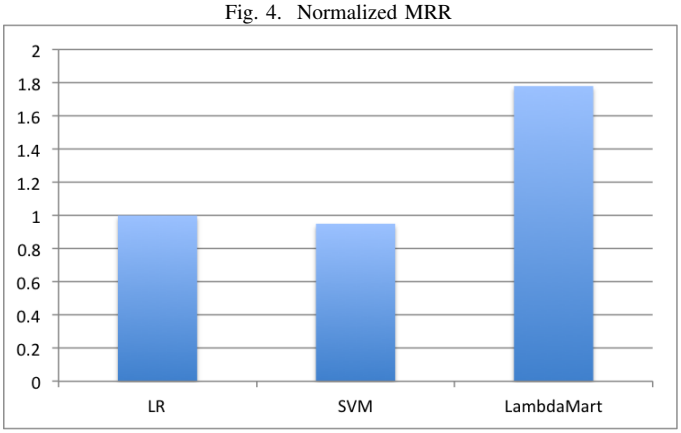


Fig. 4. Normalized MRR



We tried LamdaMART which directly optimized NDCG using listwise data. We used the RankLib package [4] for implementing LamdaMART. Note that LamdaMART and logistic regression/SVM's use completely different objective functions and hence their corresponding test/training errors cannot be directly compared like the ones plotted in Figures (1) and (2). We can however compare all the models across common metrics like CTR@1 and MRR. These are also the real search relevance metrics which ultimately matter. CTR@1 and MRR result comparisons are given in Figure (3) and (4) respectively. The results provide the following observations:

1) Logistic regression outperforms SVM in search metrics as well. This is in-line with our earlier observation on training/test errors.
2) Listwise approaches clearly outperform both logistic regression and SVM. This is due to advantages described earlier about listwise approaches (they directly optimize NDCG) as well as the fact that tree based models capture the underlying non-linearity in the data.

Fig. 1. Plot of training error vs. # Samples

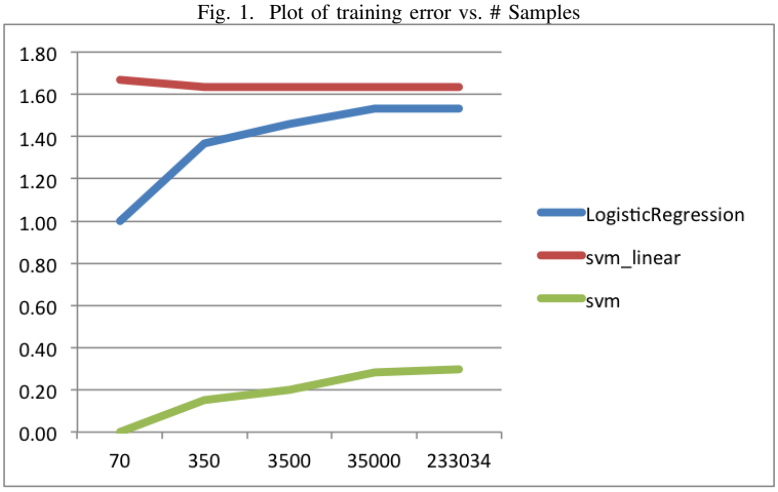
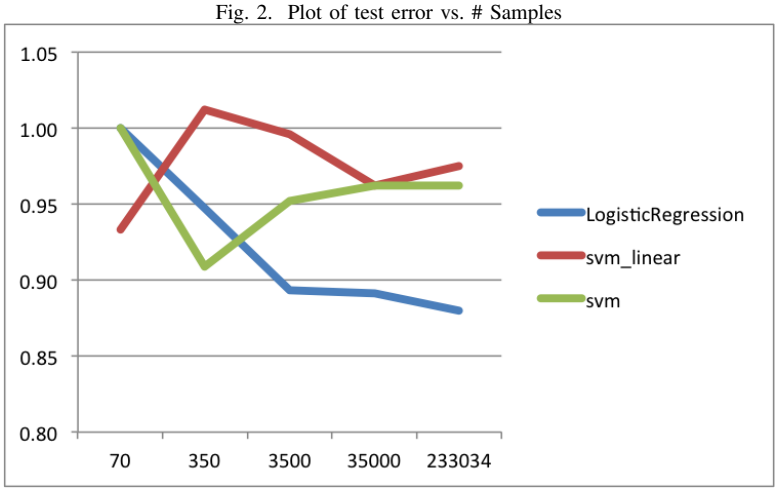


Fig. 2. Plot of test error vs. # Samples



VI. Conclusion

In this work, we took a class of real user queries (related to skills) from real LinkedIn query log and applied machine learning models. We identified the biases involved in training data collection and addressed them during the training phase. We applied few different models including logistic regression, svm (simple to start with) and moving on to more complex tree based listwise models. We observed that listwise models with tree give the best results so far.

VII. Future Work

Tree based models are harder to apply directly in production. We generated gradient boosting trees using 100 trees. Computing 100 trees for each retrieved document while maintaining production requirements for latency has it's own set of challenges. These will need to be addressed to make the listwise tree model run through live traffic.

VIII. Acknowledgments

The author would like to thank LinkedIn employees Mario Rodriguez and Viet Ha-Thuc for their valuable help and cooperation. If this work (or it's derivatives/extension) were to be published, Mario and Viet would be co-authors.

References

[1] Christopher J. C. Burges. From ranknet to lambdarank to lambdamart: An overview. *Microsoft Research Technical Report MSR-TR-2010-82*, 2012.

[2] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: from pairwise approach to listwise approach. pages 129–136, 2007.

[3] K. Hogen, H. Simon, and K. van Horn. Robust trainability of single neurons. *Journal of Computer and System Sciences, 50:114125*, 1995.

[4] http://people.cs.umass.edu/ vdang/ranklib.html. Ranklib library.

[5] T Jorchims. Optimizing search engines using clickthrough data. *Optimizing Search Engines Using Clickthrough Data*, 2002.

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[7] Filip Radlinski and Thorsten Joachims. Minimally invasive randomization for collecting unbiased preferences from clickthrough logs. 21(2):1406, 2006.

[8] S. Sinha and D. Tunkelang. Find and be found: Information retrieval at linkedin. *SIGIR 13 The 36th International ACM SIGIR conference on research and development in Information Retrieval*, 2013.

[9] Q. Wu, C. J. C. Burges, K. Svore, and J. Gao. Adapting boosting for information retrieval measures. journal of information retrieval. 2007.