Personalized Web Search using different ranking algorithms

Biman Gujral
biman@jhu.edu

Rujuta Deshpande
rdeshpa3@jhu.edu

Abstract

Our project aims at personalizing web search rankings of a user by analyzing user search history from Yandex' search engine logs. We have built a total of 73 features on the log data, which we turn on and off for different runs of the ranking algorithms. We analyze the contribution of different sets of features to the resultant NDCG score. We compare the performance in terms of the NDCG score using various Learning To Rank Algorithms - Random Forests, LambdaMART, Ranknet and AdaRank for a given set of features. We also tune the parameters of the algorithms - number of trees, learning rate, hidden nodes per layer and number of hidden layers and then, observe the change in the NDCG scores.

1 Introduction

Learning-to-rank is a technique that constructs ranking models for Information Retrieval Systems. The training data consists of a set of query-document[1] pairs. Each query-document pair forms one data point and is associated with a feature vector. This pair is given a relevance score and in the training data, the ranking is associated with an ordering over these relevance scores. The task of a learning-to-rank model, is to assign a relevance score to each query-document pair in the test data and thereby, create a permutation over the different documents returned for a particular query. Personalized rankings, re-rank the returned results in an order that would be preferable to a given user. The preference is learnt from a user's past search history, by means of user specific and generic features.

[1]Document and URL is used interchangeably

1.0.1 Background

Our project was essentially a Kaggle competition sponsored by the Russian search engine Yandex. We borrow heavily in terms of approach from the winning team of the competition.(Dataiku's solution). However, the features they used are not the same in entirety with our features. Our aim differs from theirs, in that, we intend to measure how different learning to rank algorithms can perform, in the task of personalized web search.

2 Related Work

Personalized web search has been approached in various ways. Some approaches, aim at using only the user's current session history at learning ranking, taking into consideration, the current context(Ref). There are others, which look at user's past history, disregarding the current context. User clicks have been observed to be most informative in predicting rankings(Ref). The satisfied-clicks(SAT-clicks)() have become a standard in evaluating personalized search systems. Some literature() has shown that not all queries can be personalized. Navigational queries, that are straightforward queries, aimed at accessing a unique URL need not be personalized.
Our approach, uses user's historical data but does not explicitly separate the current session of user from the previous sessions.

2.1 Algorithms

Personalized web search typically uses some learning-to-rank algorithms. They usually fall into three broad categories(Wikipedia). They are pointwise, pairwise and listwise.

2.1.1 Pointwise Algorithms

In pointwise algorithms, each query-document pair has a numerical score associated with it. Then, a learning-to-rank algorithm can transform into a regression problem of predicting a score given a query-document pair. If the scores take values from a finite set, this can even be a classification problem. eg) Random Forests

2.1.2 Pairwise Algorithms

In the pairwise approach, each pair of documents returned for a query is selected and between each member of a pair, one tries to determine the more relevant document. The goal is to minimize the inversions in a ranking.eg) Ranknet

2.1.3 Listwise Algorithms

Listwise algorithms try to either find the optimum score or minimize inversions, however, they average over all queries in the training data. eg) AdaRank, LambdaMART

2.2 Evaluation Metric

There are several evaluation metrics used in learning-to-rank algorithms. We are using one called NDCG (Normalized Discounted Cumulative Gain).
The evaluation metric used is NDCG (Normalized Discounted Cumulative Gain). It is a metric between 0 and 1 that evaluates the ranking order. It is given by:

$$NDCG_k = \frac{DCG_k}{IDCG_k}$$

where k denotes documents uptil rank k and DCG is Discounted Cumulative Gain, given by:

$$DCG_k = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{log_2(i+1)}$$

where rel is the actual relevance of the document provided by labels and i is the rank given by the algorithm. Thus, a highly relevant document ranked later in the list results in penalization and a low DCG. The Ideal DCG or IDCG gives the best ranking in accordance with the relevance values. Therefore, NDCG = 1 when the ideal ranking is obtained.

3 Our Approach

We attempt at personalizing a user's ranking data based on the past history of his searches. This is a problem that Yandex - a Russian search engine had proposed on Kaggle. The key idea that the winning team used is creating a large number of features and then, they ran the Random Forests and LambdaMART algorithms to generate an NDCG score. We are creating a subset of features that they have used. However, we have provided a mechanism for creating a configuration file that turns features on and off, hence, allowing a user to test out how various feature configurations can affect the correctness of the ranking order. We also do a comparitive study of 4 algorithms - LambaMART, Random Forests, AdaRank and Ranknet using our different feature configurations. This allows us to determine how a pointwise algorithm would perform as compared to a list wise or pair wise algorithm on the same data set.
We experiment by varying different parameters used by each of these algorithms.

3.1 Background

3.2 Data Preparation

Kaggle had provided about 16 GB of training data that covered 27 days of search log history. The format of the data is as follows:
Session metadata (TypeOfRecord = M):
SessionID TypeOfRecord Day USERID
Query action (TypeOfRecord = Q or T):
SessionID TimePassed TypeOfRecord SERPID QueryID ListOfTerms ListOfURLsAndDomains
Click action (TypeOfRecord = C):
SessionID TimePassed TypeOfRecord SERPID URLID

3.2.1 Data Sampling

We had to sample the data as our system is limited to 8 GB of RAM with dual cores. We split their train data into history, train, development and test sets for a specific subset of 28 users. This led to a significant reduction in the file size with history becoming about 120K, train 60K, development and test each with 30K file size.

Initially we wanted to split the train data provided by Kaggle, in order, however, we realized that the users are repeated serially but not across files. Hence, we randomly assigned a user's session to one of test/dev/train or history. This has led to one limitation - we have lost the information of whether a user's session is in immediate history or anterior history. The component of 'past behavior' gets reduced to a user's generic search behavior.

3.3 Feature Engineering

Our aim in this exercise is to engineer features that would lead to an improvement in the NDCG score. We generate a feature vector for query-document pair we find in the input data. Here document refers to a URL,domain combination. For each vector, we calculate a relevance label. The relevance label is based upon two factors - whether a user clicks on the URL and if he clicks, what is the dwell time or time spent on that URL and the next click. The relevance labels are in table below:

1	URL missed
2	URL skipped
3	URL clicked with satisfaction 0
4	URL clicked with satisfaction 1
5	URL clicked with satisfaction 2

For any query, a list of URLs is returned. Now, the log file contains a field called time passed, which indicates how much time has passed from the beginning of that session, uptill that query. If a URL is clicked on the previous line in the same session, before another action, we can find the dwell time for that URL as the difference of the time passed field on this action and the time passed field for the click action on that URL.

A dwell time between 50 and 300 units of time, indicates a relevance score of 4, dwell time of more than 300 units, indicates relevance score of 5 and a dwell time lesser than 50 units, indicates relevance score of 3. Moreover, the last click in every session, automatically is assigned a relevance score of 5, without considering the dwell time. We assign the missed and skipped scores, based on the *Cascade Hypothesis*. This states that, all URLs above a clicked URL have been examined (and skipped) and all URLs below the lowest clicked URL haven't been examined (and missed). Thus, all skipped URLs are given scores of 2 and all missed URLs are given the lowest score or score 1.

This way, we assign a relevance label to each query-document pair. We have broadly divided our features into three categories:

1. Generic features - these features are not specific to any user in particular. Features falling in this category are:
   -Initial rank of a document for a particular query. This rank is what would be returned by a non-personalized search engine. This rank is all the information we have about page-rank or document similarity, that Yandex might have used to compute the original ranking.
   - The frequency of a particular query. This gives us information as to how popular a particular query is
   - The number of terms in a query.

2. Aggregate features - these features can be user specific or generic, however, they are computed over all the history data, based on a certain predicate. When a particular query-document is encountered in the training or development set, we consider a set of predicates to filter logs and produce features on this subset of logs. The predicate is a conjunction of conditions on :
   - the url (same url or same domain)
   - the user submitting the query (same user or any user)
   - the query (same query or any query)
   Thus, if we encounter a query-document combination $< q_1, d_1 >$ for some user u we can choose the predicate *U*ser u, queried for q_1 and got a document d_1. Here document refers to a URL,Domain combination. So, the feature we will calculate is:

$$P(outcome = \ell|PR) = \frac{Count(\ell, PR) + p_\ell}{Count(PR) + \sum_{\ell'} p_{\ell'}}$$

There are total of 5 possible outcomes - the user either missed or skipped the URL returned, or he clicked it with a satisfaction of 0 or 1 or 2. Given a query-document, the subsequent feature for this pair, given the example predicate above and an outcome of 'skipped' would be probability that this URL was skipped given that the same user queried for the same URL previously. Similarly, for a particular predicate there can be 5 possible features, one for each outcome. In a total, we have 8 possible predicates, thus leading to a maximum of 40 possible features.

3. User specific features - These features try to learn a user's click habits. They are:
   - Number of times the user clicked on returned URLs that were in ranks $1, 2$ or in ranks $3, 4, 5$ or in ranks $6, 7, 8, 9, 10$. These lead to 3 features per user.

We generated all of these features per user and created a user's feature vector. The format of our file is:
('42', '247', '8544155', '48886358,4122937') rank:1 pos:1 terms:1 frequency:1 score:5 aggr:[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]] The initial tuple is the query-document pair information, along with user and session data. Following that, are a set of key-value pairs for all the features we consider.
However, we want to be able to turn features on and off, as well as transform this data into the format a LETOR data set follows as this is accepted by the RankLib library.
For this we define a configuration file, that has all the feature keys defined, with a value of 1 or 0 against it, that indicates if that feature needs to be considered or not. Since the aggregate features are many, we defined a comma separated string against it. A string of 0,0,1,0,0,0,1,0 indicates that we consider the predicate with value 3 or binary 011 that translates into ¡Same domain - Same Query-Same User¿ and the predicate with value 6 or 110 that translates into ¡Same URL - Same Query - Any User¿.
We then run a script that reads both a the configuration file and the user defined feature file to create a LETOR data set input file per user. Now, we can actually run the algorithms on our data.

3.4 Details of Tests

3.5 Results

Milestones achieved

Out of the milestones, that we had stated, we have successfully managed to get the most optimum feature combination by setting and unsetting different feature combinations. We have also been able to carry out a comparitive study over 4 different ranking algorithms as we had stated.

References

[1] .Masurel,K.Lefevre-Hasegawa, C.Bourguignat, M.Scordia *Dataikus Solution to Yandexs Personalized Web Search*

[2] .J.C. Burges. From RankNet to LambdaRank to LambdaMART: An Overview. Technical Report, Microsoft Research, 2010.

[3] anklib. `http://people.cs.umass.edu/~vdang/ranklib.html`

[4] .Sontag et al. Probabilistic Models for Personalizing Web Search. Microsoft Research