COMP0084: IRDM CW 2

Anonymous Submission

ABSTRACT

The project implements metrics to evaluate information retrieval quality, that is, mean average precision(map) and Normalized Discounted Cumulative Gain (map and NDCG), to first assess the retrieval quality from BM25 implemented in the previous assignment and then look at Information Retrieval through machine learning. First, logistic regression is implemented using average word embeddings followed by a Lambda Mart model and finally, a simple neural network is tuned. All machine learning tasks are evaluated using the metrics (mAP and NDCG) reported at various ranks, that is, 3, 10, and 100.

KEYWORDS

information retrieval, NDCG, mAP, Learning to Rank

ACM Reference Format:
Anonymous Submission. 2018. COMP0084: IRDM CW 2. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

1.1 Objective

How do we measure the quality of our information retrieval? To what extent are the queries being returned relevant? Are some of the questions to follow logically from our previous pursuit of implementing retrieval through Cosine-Similarity using TF-IDF vectors, BM25, and other probabilistic retrieval methods using various smoothing techniques. Furthermore, we turn our attention toward machine learning for advances in the Information Retrieval domain and implement solutions to inquisitively understand as we meander through the different architectures in succession, what machine-learning architectures work well for information retrieval, how does the training loss vary with a higher or lower learning rate in case of logistic regression with the specified representation of data, how does a Lambda-Mart model perform to re-rank the passages? And if a chosen neural network is able to learn the ranking as well.

An exhaustive list of objectives is as follows.

(1) Implementing methods to compute metrics to asses retrieval quality, that is, mean average precision (map) and Normalized Discounted Cumulative Gain
(2) Implementing Logistic Regression on word-embedding representations of the passages and queries using a word-embedding method (Word2Vec) and studying how training loss varies with the learning rate. The implementation is then evaluated using map and NDCG.
(3) Implementing a Lambda-Mart model using the XGBoost library and features for both the query and passages and evaluating using map and NDCG.
(4) Implementing a neural network to rank the passages using the same representations from the Task 3 and evaluating it using the metrics map and NDCG.

1.2 The Dataset

The following tab-separated files were provided to us

(1) *test-queries.tsv* a file with 200 rows of queries same as the previous coursework, with qid tab-seperated with the query text.
(2) *train _data.tsv* a file with 4,364,338 rows containing <qid> <pid> <queries> <passage> <relevancy>, all tab-separated. In all the file contains 4,590 unique queries and 2,933,768 unique passages along. Total 4,797 relevant passages were present. A total count of relevant passages for queries is given below.
   (a) 4,403 queries with 1 relevant passage
   (b) 168 queries with 2 relevant passages
   (c) 18 queries with 3 relevant passages
   (d) 1 query with 4 relevant passages
(3) *validation_data.tsv* a file with 1,103,038 rows with tab separated <id> <pid> <queries> <passage> <relevancy> fields. The filed contains 1,148 unique queries and 955,211 unique passages. In total 1,208 relevant passages were present. A total count of relevant passages for queries is given below.
   (a) 1,094 queries with 1 relevant passage
   (b) 49 queries with 2 relevant passages
   (c) 4 queries with 3 relevant passages
   (d) 1 query with 4 relevant passages
(4) *candidate _passages _top1000.tsv* a file with 189,876 rows <qid> <pid> <query> <passage>. The file essentially has the same data as the previous CW with 1,000 passages per query but with initial rankings.

2 TASK 1 : EVALUATING RETRIEVAL QUALITY

2.1 Task Overview

Primordially, retrieval evaluation is of utmost importance for it essentially answers the question of how effective a retrieval mechanism has been in matching the information with the user requirements or queries. In Coursework 1, we implemented BM 25 but didn't implement a method to evaluate what was the quality of our retrieval.

For the scope of our task, we are only going to look at Offline retrieval in a classical sense where the relevancy of documents has been split only into relevant and non-relevant, that is 1 and 0. We first proceed by training our implementation of BM25 from task 1 on the dataset, that is, *validation _data.tsv* and then move on to evaluate our model using the metrics that we have been asked to implement, Mean Average Precision(mAP) and Normalized Discounted Cumulative Gain (NDCG).

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

2.2 Pre-processing

As juxtaposed to the previous task, we make slight changes to our approach. Rather than tokenizing the words first using a word tokenizer from the NLTK library followed by lemmatization and stop word removal, we use Gensim's in-built pre-processing using the following command **gensim.utils.simple_preprocess**.

According to the official documentation, the aforementioned utility converts the text into lower case tokens ignoring too small or too long tokens.

The approach was primarily changed because the previous approach increased the processing time considerably with a comparatively larger dataset here.

2.3 Mean Average Precision

If understood through its eponymous relation to the term precision, the term, mean average precision can be very misleading as it is far from the mean of precision. Rather, the term is actually the mean of average precision (AP) for all the queries we are looking at for the retrieval problem. Average precision basically condenses the precision-recall curve into a value.

This happens by going through every retrieved document for a particular query and at every rank, the precision is calculated considering if the document was retrieved or not divided by the rank of the retrieved document. For example if at rank 1, the document retrieved was relevant the score for it will be $\frac{1}{1}$. Supposedly if for the next rank, the document retrieved is non-relevant, the score for it will be $\frac{1}{2}$, where the denominator corresponds to the rank.

All the scores for each rank are then added up for up till the threshold till which we are reporting the AP. In our case, the thresholds were 3,10 and 100. And the total sum of the average precision(s) for each query then up till the threshold is divided by the total relevant documents available up till the threshold rank.

The AP score is then summed up for every query and then divided by the total number of queries.

It must be noticed the highest weight for the score in average precision is assigned to the topmost passage that is relevant, and the second passage is considered to be twice as less important as the first.

2.4 Normalized Discounted Cumulative Gain

Normalized Discounted Cumulative Gain is simply put the Discounted Cumulative Gain optimized by the possible performance, that is, perfect ranking scenario of the particular query in question.

The Discounted Cumulative Gain (DCG) lays emphasis on the retrieval of highly relevant documents and uses graded relevance as opposed to a binary of relevance/non-relevance. The DCG makes the following assumptions.

(1) Highly relevant documents are more relevant than say a document that is marginally relevant in context of the query at hand.
(2) Lower ranked documents are useful for the user and the strength of usefulness for the user generally lowers with a lowering rank.

The term gain finds its roots in how graded relevance is used as measure of usefulness in gain from examining a document. The gain accumulation is done from the starting and ranks and is discounted for the lower ranks.

For our implementation we use the alternative formulation from the week 3 slides [?], where DCG at a rank p is given by.

$$DCG_p = \sum_{i=1}^{p} \frac{2^{rel_i}}{log(1+i)}(1) \tag{1}$$

Where,

- p denotes the rank up till we are looking at the gain for
- rel_i is the relevance for the i$^{\text{th}}$ rank for which the query is being currently examined.

The NDCG is then calculated as

$$NDCG = \frac{DCG}{optDCG}(2) \tag{2}$$

2.5 Implementation

First we implement the BM25 on the validation data from our last task after the pre-processing specified in section 2.1 using gensim.

Then after the scores are obtained for every rank we choose to report the mAP and NDCG at, we basically create different data structures that use in both the functions we wrote for mAP and NDCG.

We first create a data-dictionary containing the scores per each unique query in our validation_data.tsv and then sort them as per the scores into top 3, top 10, top 100 ranks. We basically write a function to do so which can sort the scores up-to a specified rank and then returns the dictionary in the form of <qid> : [pid:score, pid, score].

Then we write a function to basically create a dataframe from this dictionary, containing qid, pid, score and relevancy of the fetched passage from the score supplied. Then we also create another dictionary that stores the number of relevant passages available per query and stores it in the format <qid>: number of relevant passages.

The functions for both mAP and NDCG can also take another argument which is up-to what rank should the metric be reported, and those in our case are 3,10, and 100.

The arguments for the mAP function can be defined as (sorted dataframe,dictionary with sorted scores, dictionary with total relevant passages, rank score requested upto). Both the functions take the same arguments.

2.5.1 mAP. The mAP function only counts the scores for queries which have the minimum required threshold by checking the length, that is, number of elements in the sorted dictionary per each query id. For example, if we are reporting up till ranks of 3, queries which do not have even 3 passages available to rank and score from are excluded.

The AP scores for each query is calculated and summed, in the end divided by the total number of relevant queries available for that query.

Later, all AP scores are added and a mean is taken out by divding from the total number of queries.

2.5.2 NDCG. A similar approach of excluding queries with not the required number of passages is followed from the previous section, and first the DCG is calculated per query using equation (1) and

then optimized DCG is calculated from the dictionary giving us the total number of relevant passages available per query. DCG is divided by optimized DCG and NDCG is caclulated using equation (2).

2.6 Scores

mAP

- The mAP at rank 3 was 0.17
- The mAP at rank 10 was 0.20
- The mAP at rank 100 was 0.22

NDCG

- The NDCG at rank 3 was 0.19
- The NDCG at rank 10 was 0.26
- The NDCG at rank 100 was 0.32

3 TASK 2: LOGISTIC REGRESSION

3.1 Task Overview

This task onward we begin to inquisitively look at retrieval using machine learning models. In this task we first pre-process the texts, both in the training and validation data available to us. Then we concatenate all the unique passages, queries available in both the training and validation data and then train our Word2Vec model imported from the Gensim library.

We choose to pre-train our own model, for we wanted that the models learns context based on the total corpus we had and builds the vocabulary around it. Furthermore, on using pre-trained embeddings trained on Google News, size of the arrays increased considerably with no gain in performance. Both the approaches were tried and training own model was found to be more memory efficient with no considerable gains. Later, a subsample was taken out from the dataset, by assigning a weight of 60 to the relevant passages and a weight of 5 to the non-relevant passages and random 600,000 rows were pulled. Later, since the class to be predicted was still in extreme minority, we oversampled the data using sampling techniques built-in in pandas. The average word embeddings were then computed for each query and passage, and then a logistic regression model was trained with a learning rate of 0.9 with 400 iterations.

The retrieval evaluation quality was assessed using mAP and NDCG.

3.2 Pre-processing and Embeddings

The basic pre-processing for the text remained the same as task 1 (section 1.2). Later, a Word2Vec model was chosen for the word embedding task with the default CBOW configuration. The model was tuned with the following parameters.

- window=15
- min count=2
- workers=20
- sg=1
- negative=4
- sample=0.001

3.3 Average Word Embeddings

For the average word embeddings in Gensim 4.0, we used the index_to_key function and our process was to pass on the every term in the tokenized text for both the queries and the passages.

For each query and passage, let's assume and call it text t, then the term embedding can be represented by[4]

$$\overrightarrow{v_t} = \frac{1}{|t|} \times \sum_{t_i \epsilon t} \frac{vector_{t_i}}{||vector_{t_i}||} \tag{3}$$

Where,

- v_t denotes the centroid of the text, which could be a passage or a query
- i denotes every term or token in the query or passage
- and the normalized notation for vector holds a usual meaning from linear algebra

After calculating the average word embedding for each query, all of them are appended in a list and a vertical stack is taken of them using numpy's np.vstack. Same is also done for the passage.

3.4 Final Representations for the model

For our final representations, we stacked centroid of both passage and query together, along with taking a cosine similarity score of the query and passage inspired from dual embedding space models representation[4] also covered in week 6.

The cosine score was given by.

$$Sim(q, d) = Cos(\overrightarrow{v_q}, \overrightarrow{v_d}) \tag{4}$$

where, $Cos(\overrightarrow{v_q}, \overrightarrow{v_d})$ can be further written as

$$Cos(\overrightarrow{v_q}, \overrightarrow{v_d}) = \frac{\overrightarrow{v_q}.\overrightarrow{v_d}}{||\overrightarrow{v_q}|| \times ||\overrightarrow{v_d}||} \tag{5}$$

For the final feature set, a horizontal stack of query array, passage array (derived using 3) was stacked with the cosine similarity score of the query and the passages, in all having 201 features and were passed into the implementation of logistic regression after normalization. Later, the predictions were evaluated for precision, recall, and the cross entropy losses were also saved as lists using json and were analyzed for learning rates 0.9, 0.01, 0.001 for 20 iterations of the model each. A plot of this can be found in the next section.

3.5 Analyzing Loss, Precision, and Recall and other experimentation

From Figure 1 a rather odd trend of cross-entropy loss can be seen for 20 iterations for all of the learning rates. For 0.01, 0.001 the loss hovers around 0.69 and for 0.9 the loss actually increases after converging and goes up-to 1.4 then starts dropping.

From Figure 2, this trend can be seen over 400 epochs and it can be seen the loss actually drops around 1 but converges there. While it might seem the cross entropy loss can directly translate to a gain in NDCG it's quite the opposite. The loss did signal the model got stuck at saddle points but in case of 0.9 the model did really learn to rank better.

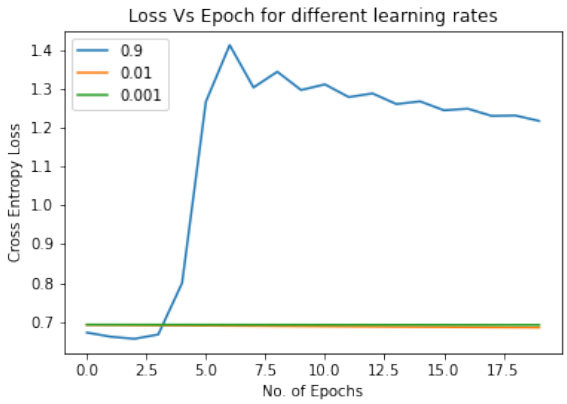

Figure 1: Learning Rate Vs Cross Entropy Loss

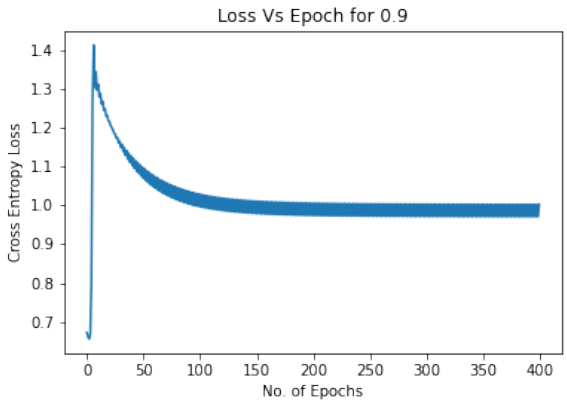

Figure 2: Cross Entropy Loss for Learning Rate 0.9 for 400 Epochs

Learning Rate	mAP 100	NDCG 100	Precision	Recall
0.9	0.018	0.061	0.50	0.56
0.01	0.014	0.047	0.63	0.57
0.001	0.011	0.043	0.62	0.57

Table 1: Learning Rate and Evaluation metrics without Lemmatization

Learning Rate	mAP 100	NDCG 100	Precision	Recall
0.9	0.011	0.042	0.56	0.53
0.01	0.011	0.040	0.61	0.57
0.001	0.011	0.038	0.60	0.57

Table 2: Learning Rate and Evaluation metrics with Lemmatization

Another experiment that was tried was to justify the choice of not using lemmatization and we found that using lemmatization actually dropped the model performance.

As can be seen from a comparison of performances from table 1 and table 2, the models without lemmatization including training with lemmatization on Word2Vec performed better.

Consequently, our choice further on involved not using lemmatization for training the word2vec model.

3.6 Final Results

Inspired from our learning around the fact that learning rate of 0.9 worked the best, we tuned our model for 400 iterations and got the following scores

mAP

- The mAP at rank 3 was 0.024
- The mAP at rank 100 was 0.041
- The mAP at rank 100 was 0.041

NDCG

- The NDCG at rank 3 was 0.026
- The NDCG at rank 10 was 0.052
- The NDGC at rank 100 was 0.11

As requested, a file was compiled and saved in the .txt format using pandas, with <qid> <A1> <pid> <score> <algorithm name>.

4 TASK 3: LAMBDA MART

4.1 Task Overview

In our last section, we looked average word embeddings and tuned a custom-implementation of Logistic Regression albeit we saw it get stuck into a rather textbook problem associated with non-convexity of getting stuck into a local minimum. We did see slightly better results for learning rate of 0.9.

In our current task we use the XGBoost Library to implement pairwise ranking using the Lambda-Mart model[3]. For this we come up first with features/representations to train the model, and then do the hyper-parameter tuning of the model by taking out a holdout set from the the subs-sampled dataset of the training data we discussed in section 1.2.

After our optimal parameteres are returned, we move on-to training the final model on the entire sub-sample taken out minus the hold-out and then get the predictions from our validation dataset using the same representations. We then evaluated the results obtained using the implementations of mAP and NDCG from section 2.

4.2 Features Used

For our task we look at the Learning To Rank for Information Retrieval paper [5] and come up with the features to train our datasets with. We also used the cosine similarity derived from the centroid of embeddings, using our trained Word2vec model from section 3 (task 2). In all our total features can be seen in the table below.

4.3 Approach

To save time on pre-processing, we took only the first 2,000 unique queries from the dataset and did our pre-processing and calculation

Table 3: Features and their description

Features	Description
score	BM25 Scores
cosine_similarity	Cosine Similarity
tf_idf_passage_sum	Sum of TF-IDF for every term in passage
tf_idf_query_sum	Sum of TF-IDF Query for every term in query
passage_length	Passage Length
query_length	Query Length
tf_idf_diff	tf_idf_passage_sum-tf_idf_query_sum
pid/qid_feature	tf-idf_diff/tf_idf_passage_sum

of representations on that dataset. First, we calculated the cosine similarity, followed by BM25 scores, tf-idf scores, and other features. We alsov saved the final representation file locally so it could be used in the next section. Later, from the training dataset, we took first 1200 queries and the related dataset for training and then took the rest 800 queries and their related dataset as a holdout to tune our hyperparameteres. For the training and grouping, we used the xgb DMatrix to hold our data and used the internal set_group function, by first using the groupby in pandas to pass on the size of each group to the DMatrix.

It must be noted that for our training, both during hyper-parameter tuning and final training, we tune our models only the first <300 ranks according to BM Scores per query. We do this to solve for imbalanced data and saw minute gains using this approach.

4.4 Hyperparameter-Tuning

For the hyper-parameter tuning, we wrote a custom function and trained our model on a test-set and tested it on the holdout set, the function returned the best parameters after the training was complete.

Hyper-parameters we tuned on were

- learning rates: 0.99 and 0.01
- gamma values of 0.3 and 0.7
- maximum depth of 3 and 5
- minimum child weight of 0.3 and 0.6
- eta shrinkage parameter of 0.1 and 0.5

The best hyper-parameters found after the tuning were learning rate of 0.99, maximum depth of 3, gamma of 0.3, and minimum child weight of 0.3. Additionally we used number of estimators as 90, objective as "rank:pairwise" and evaluation metric as "ndcg@10".

4.5 Feature Importance

The feature importance was plotted with bm25 score being the most important followed by cosine similarity(Figure 3). We tried removing the unimportant features but it didn't really improve the performance and saw minute losses so we decided to train the final model on all the features and then did the pre-processing on the validation dataset to create the representations to get predictions and evaluate the implementation.

4.6 Final Results

The final results obtained were evaluated using mAP and NDCG.

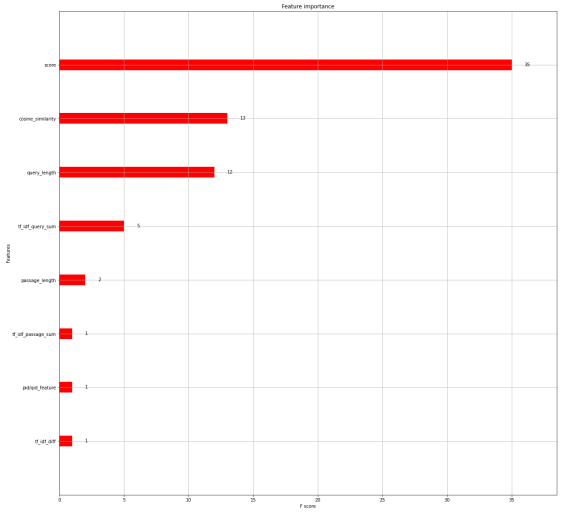

Figure 3: Feature Importance using xgb's internal implementation based on FScore

mAP

- The mAP at rank 3 was 0.145
- The mAP at rank 100 was 0.174
- The mAP at rank 100 was 0.187

NDCG

- The NDCG at rank 3 was 0.161
- The NDCG at rank 10 was 0.223
- The NDGC at rank 100 was 0.294

The final file was saved as LM.txt in the requested format of <qid> <A1> <pid> <rank> <score> <algorithm>.

5 TASK 4: NEURAL NETWORK

5.1 Task Overview

For this task, the approach was to look at how a neural network could optimize the results for ranking further than what was achieved by say Lambda-Mart using the same features we used in Task 3 and can be found in Table 3. For our task we use a simple neural-net, that is, a multi-layer perceptron imported from sklearn and then train it on the representations for the first 2,000 queries in the training dataset and later test it on the validation dataset, evaluating it using mAP and NDCG,

5.2 Approach and Features

We used our saved files from the pervious section, where save a file called bm25h.pkl containing the representations from task 3 on the training data and a filed called valdh.pkl containing the representations for the validation data from task 3.

We first read the pickled training data from task 3, normalize the X values using a custom implementation from task 2. Then we

pass on the inputs and labels to a multi-layer perceptron we created with the following parameters.

- activation= relu
- solver = adam (to aid convergence since we did see this trend of getting stuck at a local minimum in task 2)
- batch size = 200
- hidden layer size of (50,2)
- random state =1
- and maximum iterations of 400

An approach to over-sample the dataset was also tried like task 2 but it didn't improve any results.

Our features and sampling is essentially the same from the previous task since our over-sampling like task 2 didn't yield any good results.

5.3 Architecture

We choose a feed-forward network multi-layer perceptron for our task of supervised learning. A multi-layered perceptron, can learn a function a function $f : R^m \rightarrow R^o$ where m is the number of dimensions of the training data and o is the number of dimensions of the output given a set of features $X = x_1, x_2....x_m$ and a target y [**?**].

A multi-layer perceptron can learn non-linear approximations for both classification and regression through hidden layers which in our case have a dimension of 50,2. We also choose a rectified linear unit (reLU) as the activation function including adam as the solver.

A reLU activation function is extremely lucid in its function as a $max(x, 0)$ that is it gives the output as x when the x is positive else for a x = x<=0 it returns a zero value.

We also use adam, like we have already mentioned above to solve for the problem of getting into a local minimum. We tried multiple hidden states before settling for (50,2) which gave the best NDCG scores.

5.4 Final Results

The final results obtained were evaluated using mAP and NDCG.

mAP

- The mAP at rank 3 was 0.129
- The mAP at rank 100 was 0.148
- The mAP at rank 100 was 0.160

NDCG

- The NDCG at rank 3 was 0.143
- The NDCG at rank 10 was 0.183
- The NDGC at rank 100 was 0.255

The file was save as NN.txt in the requested format.

5.5 Conclusion

We inquisitively looked at various architectures throughout from tasks 1 to 4, and looked at various types of architectures from a vanilla logistic regression that gets stuck at a local minimum sans any optimization and yields rather abysmal NDCG, to a Lambda mart model that showed an improvement in re-ranking the passages, and finally a simple feed-forward network with a hidden layer that didn't really perform as well on the features we used for task 3.

While more state of the art solutions are available and did try an approach for task 4 that didn't lead any fruitful results based on an LSTM as it kept crashing, it must also be noted that the representations used in task 3 didn't convince to look for an architecture that could look at them as a sequence. Perhaps if we were to do it differently, and pass the embeddings into a sequence-to-sequence encoder-decoder network that could take advantage of a sequence kind of representation, we could have seen a better performance. But in all of our three tasks, the best performing model was our own BM25 that gave a not exceptional score which was better than all the other implementations except for say Lambda Mart that managed to inch closer to the BM25 albeit using the score as feature.

REFERENCES

[2]]yilmaz [n. d.].

[2]]scikit [n. d.]. 1.17. neural network models (supervised). https://scikit-learn.org/stable/modules/neural_networks_supervised.html

[3] Christopher JC Burges. 2010. From ranknet to lambdarank to lambdamart: An overview. *Learning* 11, 23-581 (2010), 81.

[4] Eric Nalisnick, Bhaskar Mitra, Nick Craswell, and Rich Caruana. 2016. Improving document ranking with dual word embeddings. In *Proceedings of the 25th International Conference Companion on World Wide Web*. 83–84.

[5] Tao Qin, Tie-Yan Liu, Jun Xu, and Hang Li. 2010. LETOR: A benchmark collection for research on learning to rank for information retrieval. *Information Retrieval* 13, 4 (2010), 346–374.