Assignment 2

Document Reranking Task

0. Introduction

There are *3213835* documents in the dataset *(named **MS-MARCO**)* which is based on a collection from *Microsoft Bing*. The collection is in a single tsv file in the format: `docid`, `url`, `title`, `body`. ***The body has already been pre-processed to remove the HTML tags.*** The queries are given in a separate tsv file in the format: `query_id` and `query`.

The top-100 documents for each query retrieved from a BM25 model has already been provided (in the same format supported by *trec_eval*) and the aim of this assignment is to reorder/rerank these documents so that more relevant documents appear first using two methods: **Probabilistic Retrieval Query expansion** and **Relevance Model based Language Modelling**.

1. Probabilistic Retrieval Query expansion

1.0. Introduction

The implementation is based on a technical report *"Simple, proven approaches to text retrieval"* by S.E. Robertson and K. Spärck Jones[1]. The terms taken from relevant documents are ranked according to their Offer Weight (Robertson 1990) defined as:

```
OW(i) = r * RW(i)
```

where RW(i), called the Relevance weights, is defined as:

```
RW(i) = log [((r+0.5)(N-n-R+r+0.5))/((n-r+0.5)(R-r+0.5))]
```

where

- `n` = the number of documents term t(i) occurs in
- `N` = the number of documents in the collection
- `r` = the number of known relevant documents term t(i) occurs in
- `R` = the number of known relevant document

In this assignment. `N` is taken to be the top-100 documents and a small percentage of these 100 as `R` (`R`=35). Such a decision was made after observing the performance in both the cases (taking N as the full collection vs N as 100 documents given). Moreover, the case of taking N documents completely has offline costs for indexing (which took about 6 hours) to compute the number of documents having each term (idf). For instance for expansion limit set to 1, the case of taking N as entire collection had a ndcg of about 0.2416 while the case of taking N as top-100 documents had a ndcg of about 0.2798 for 114 queries in dev-set.

[1] *"Simple, proven approaches to text retrieval"* by S.E. Robertson and K. Spärck Jones [https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-356.pdf]

1.1. Algorithm

1. Parse the collection and store the offset to each record (will help in efficient retrieval while querying)
2. Open the query file and top100 documents and get the 100 documents matching each query. For each query, get the list of document IDs from top 100 file and do:
    1. Treat the first 35 documents as relevant (R=35) and total number of documents is taken as 100 (N=100). Process the query (which involves tokenisation, stemming using *Krovetz Stemmer* and stop words removal from *nltk english corpora*).
    2. Initialise a dictionary to store number of documents having a term (df) and number of relevant documents having that particular term. Note that we treat the first 35 documents as relevant and the vocabulary corresponds to the terms in the top-100 documents (after pre-processing) only. Also, initialise a list to store the body of each document in the top100 list. We also calculate the documents length which is the sum of number of terms in each of the top100 documents.
    3. For each doc_id collected:
        1. Fetch the record from the collection using the offsets collected in Step 1
        2. Process the document body (which involves tokenisation, stemming using *Krovetz Stemmer* and stop words removal from *nltk english corpora*).
        3. Add the document length to total document length and add this document body to the list of all documents body of top 100 documents.
        4. For every distinct term in the body, add this term to the dictionary used to store df. Then, if this term is not in the query and if we are still working on the top-35 documents, add it to the dictionary that stores document frequencies of relevant (top-35) documents.
    4. Compute Offer Weight for each term present in the top-35 documents with the expression similar to the one in Section 1.1. Note that r=0 for terms not in top 35 documents and we can be ignored. However, we modify RW(i) as given below to avoid problems in domain of logarithm:
       `RW(i) = log [((r+0.5)(4N-n-R+r+0.5))/((n-r+0.5)(R-r+0.5))]`
    5. After computing OW(i) for each term, we rank the terms based on the scores. Note that the query terms will **not** be present (as we ignored them while adding it to document frequencies for relevant documents in Step 3.4.
    6. Pick the top-n terms (where n is the expansion limit) and add this to the query (query expansion)
    7. Pass the new query terms and document bodies to a BM25 model which does the following:
        1. Set k1=1.4 and b=0.75 (constants in Okapi BM25 model[2].
        2. The scores for each document will be stored in a list and will be updated query-wise (rather than document-wise).
        3. For each query term,
            1. Compute the IDF as per the formula in BM25
            2. For each document:

[2] Okapi BM25: https://en.wikipedia.org/wiki/Okapi_BM25

        1. Find the frequency of the term in the document
        2. Get the score for the document for the particular query term using BM25
        3. Add this score to the list that stores the scores of the document
    4. Having obtained the score of all the documents, rerank them such that higher scored documents appear first and return the (doc_id,score) for all the documents.
8. Output the ranked documents list in the format that trec_eval can understand.

1.2. Observations

a. It was observed that treating N as the top 100 documents and R as top 35 documents *[case i]* (rather than whole collection and top 100 *[case ii]*) performed slightly better. For a case where 114 queries were considered from dev-set, ndcg was observed to be about 0.2798 for *case i* and 0.2416 for *case ii*. On the other hand, the mrr was 0.1720 and 0.1302 respectively.

b. When tried to iterate over different values ([20,25,30,35,40,45]) of R, number of documents assumed to be relevant, there was no significant difference in metrics observed (differed by ~0.1). Hence, a random (favourite) value of 35 was chosen.

c. The reason for modifying RW(i) is to avoid negative values in domain of logarithms. For instance, consider >>> N, n, R, r = 100, 75, 35, 1. This does not affect the relative ranking as all the values are integers and are expected to be more than 1 most of the times.

1.3. Performance Report

(Note that the performance testing were done on dev-set while statistics report is based on 250 queries on dev-set)

	MRR	**nDCG**
original	0.2219	0.3295
expansion = 1	0.1500	0.2623
expansion = 2	0.1339	0.2486
expansion = 3	0.1278	0.2427
expansion = 4	0.1184	0.2345
expansion = 5	0.1115	0.2286
expansion = 6	0.1052	0.2230
expansion = 7	0.1021	0.2204
expansion = 8	0.0977	0.2163
expansion = 9	0.0954	0.2139
expansion = 10	0.0922	0.2109

Paired t-Test (2-tailed)

	MRR			nDCG		
	p	tstat	sd	p	tstat	sd
expansion = 1	0.0715	1.8064	0.2861	0.0648	1.8510	0.2781
expansion = 2	0.0272	2.2159	0.2828	0.0246	2.2538	0.2749
expansion = 3	0.0121	2.5190	0.2791	0.0118	2.5285	0.2718
expansion = 4	0.0022	3.0722	0.2728	0.0028	3.0056	0.2662
expansion = 5	1.8141E-04	3.7720	0.2600	8.789E-05	3.5105	0.2567
expansion = 6	1.4964E-04	3.8211	0.2599	4.1008E-04	3.5576	0.2564
expansion = 7	1.0082E-04	3.9203	0.2579	2.9637E-04	3.6442	0.2551
expansion = 8	4.8301E-05	4.0997	0.2566	1.5906E-04	3.8056	0.2537
expansion = 9	3.3667E-04	3.6104	0.2640	7.0282E-04	3.4098	0.2594
expansion = 10	2.1968E-05	4.2847	0.2559	7.4537E-05	3.9948	0.2526

Wilcoxon signed rank test

	MRR			nDCG		
	p	zval	signedrank	p	zval	signedrank
expansion = 1	2.3810E-06	4.7181	10752	1.4776E-06	4.8142	10816
expansion = 2	2.3696E-07	5.1677	11286	1.6570E-07	5.2342	11331
expansion = 3	1.5112E-08	5.6603	1.1984E+04	1.2296E-08	5.6956	12009
expansion = 4	2.2180E-09	5.9810	12083	1.2652E-09	6.0718	1.2146E+04
expansion = 5	5.0971E-10	6.2161	1.2750E+04	2.8118E-10	6.3088	1.2816E+04
expansion = 6	3.1232E-10	6.2925	12550	2.3132E-10	6.3390	1.2582E+04
expansion = 7	2.0163E-11	6.7048	1.3492E+04	1.6785E-11	6.7316	13512
expansion = 8	7.7866E-11	6.5047	1.3086E+04	3.7569E-11	6.6134	1.3164E+04
expansion = 9	7.2339E-09	5.7854	12195	3.2257E-09	5.9197	12289
expansion = 10	1.6761E-11	6.7318	13380	1.0349E-11	6.8016	1.3430E+04

1.4. Conclusion

The actual setup of the BM25 used for initial ranking of top-100 documents is not known and hence the performance based on the metric values appear poor even after query expansion.

2. Relevance Model based Language Modelling

2.0. Introduction

In this part, *Lavrenko and Croft's* relevance model is implemented using *Unigram Model with Dirichlet Smoothing* and *Bigram Model with Dirichlet Smoothing with Unigram Backoff*.

Firstly, relevance model probabilities P(w|R) is estimated using *Lavrenko and Croft's* relevance model as:

$$P(w|R) = \sum_{M} P(M)P(w|M)\prod_{q\in Q} P(q|M)$$

It has been assumed that M follows uniform distribution (can be dropped) and all query terms (q) and w are independent (in fact, iid).

Then the documents are reranked using KL-divergence as

$$\sum_{w} P(w|R)\log(P(w|D))$$

Here, we assume w belongs to set of top 100 documents only (rather than the collection) to reduce the time spent for computation. Additionally, Dirichlet smoothing is used while implementing the relevance model as:

$P(t|M) = \dfrac{f_{t,d} + \mu P_C(t)}{|D| + \mu}$ where $P_C(t) = \dfrac{f_{t,C}}{|C|}$ and μ is set as 2.

In the above expression, $f_{t,D}$ denotes the frequency of term t in document D, $|D|$ is the total number of terms in the document, $f_{t,C}$ is the frequency of term t in the collection (*note that this refers to the entire collection which can be computed offline*) and $|C|$ is the number of words in the collection.

A similar approach is used in the bigram model, where we take pair of words (w[i-1],w[i]) and apply KL-divergence across all such pairs of terms. We replace $P(q)$ as $P(q_0)P(q_1|q_0)P(q_2|q_1)\ldots$ and get the following analogous expressions:

$$P(w_1|w_0, R) = \sum_{M} P(M)P(w_1|w_0, M)P(q_0|M)P(q_0)P(q_1|q_0, M)P(q_2|q_1, M)\ldots$$

and the score of the documents as:

$\sum_{w'} P(w'|R)\log(P(w'|D))$ where w' are the set of all pairs that occur together in document (like {(w0,w1),(w1,w2),…}

The modified Dirichlet smoothing expression for a bigram setup which ultimately has a unigram backoff is given as:

$$P(w_i \mid w_{i-1}, D) = (1-\lambda_1)\left[(1-\lambda_2)\frac{f_{w_i,w_{i-1},D}}{f_{w_{i-1},D}} + \lambda_2\frac{f_{w_i,D}}{|D|}\right] + \lambda_1\left[(1-\lambda_3)\frac{f_{w_i,w_{i-1},C}}{f_{w_{i-1},C}} + \lambda_3\frac{f_{w_i,C}}{|C|}\right]$$

where $\lambda_i = \dfrac{\mu_i}{|D| + \mu_i}$ is a hyper-parameter and whose terms are similar to the unigram setting. They are set as $\mu_1 = 1$, $\mu_2 = 2$ and $\mu_3 = 2$ (there is no specific reason behind choosing such values).

In some places, values are added with a small value *(like 1e-3)* ***to avoid 'Divide By Zero' error.***

2.1. Algorithm

For unigram setting:

1. Parse the collection and store the offset to each record (will help in efficient retrieval while querying). While doing this, also process (which involves tokenisation, stemming using *Krovetz Stemmer* and stop words removal from *nltk english corpora*) the document body and store the frequency of each term and number of words in the collection.
2. Open the query file and top100 documents and get the 100 documents matching each query. For each query, get the list of document IDs from top 100 file and do:
    1. Initialise a dictionary to store the words that appear in the top 100 documents.
    2. Process the query (which involves tokenisation, stemming using *Krovetz Stemmer* and stop words removal from *nltk english corpora*).
    3. For each document in top 100, do:
        1. Get the offset of the record and read the record from the collections file corresponding to the doc_id given.
        2. Get the document body part of the record and process it.
        3. Add the processed document body to a list of all document body in top 100 and update the length of the document.
        4. Make a note of words that are present in these documents.
    4. Now calculate $P(w \mid R)$ for every word in the top 100 documents as:
        1. *The steps are modified (rearranged) to improve efficiency.* Maintain a dictionary to store the P(w|R) value until the i-th document that is been processed.
        2. For each document, do:
            1. It can be observed that $\prod_{q \in Q} P(q \mid M)$ remains constant for a given document. Use the Dirichlet smoothing and compute this product and store them (say, *prod*)
            2. For each word (w) in the top 100 documents:
                1. $P(w \mid M)$ can be computed

        2. Update the contribution (the products P(w|M) and *prod*) of the particular document to the word (w) through the dictionary mentioned in previous steps.

5. Having obtained $P(w|R)$ for all words, use KL divergence to rank EACH document as:
    1. set curr_score = 0
    2. For each word w present in the top 100 documents, find $P(w|R)\log(P(w|D))$ (*the former can be obtained from the dictionary created in the previous step and just the latter has to be computed for each word*). Add these scores.
6. Having obtained the scores of each document, sort it to obtain the new ranks.
7. Output the ranked documents list in the format that trec_eval can understand.

Note: Dirichlet Smoothing is applied wherever necessary.

For bigram setting:

1. Parse the collection and store the offset to each record (will help in efficient retrieval while querying). While doing this, also process ***without stop word removal*** (which involves tokenisation, stemming using *Krovetz Stemmer*) the document body and store the frequency of each term and pair of consecutive terms (bigrams) & number of terms in the collection.
2. Open the query file and top100 documents and get the 100 documents matching each query. For each query, get the list of document IDs from top 100 file and do:
    1. Initialise a dictionary to store the consecutive pair of words that appear in the top 100 documents.
    2. Process the query ***without stop word removal*** (which involves tokenisation, stemming using *Krovetz Stemmer*).
    3. For each document in top 100, do:
        1. Get the offset of the record and read the record from the collections file corresponding to the doc_id given.
        2. Get the document body part of the record and process it ***without stop word removal***.
        3. Add the processed document body to a list of all document body in top 100 and update the length of the document.
        4. Prepare the document body to get pairs of consecutive words in the document and prepare a list of this example. *e.g. [(w0,w1),(w1,w2),(w2,w3),..] for a document having “w0 w1 w2 w3…”*. Append this as pair of document body.
        5. Make a note of pairs of consecutive words that are present in these documents.
    4. Now calculate $P(w'|R)$ for every pair of consecutive words in the top 100 documents as (*w’ is used to denote pair of words*):

1. *The steps are modified (rearranged) to improve efficiency.* Maintain a dictionary to store the P(w'|R) value until the i-th document that is been processed.
2. For each document, do:
    1. It can be observed that $P(q_0|M)P(q_1|q_0, M)P(q_2|q_1, M)\ldots$ remains constant for a given document. Use the Dirichlet smoothing and compute this product and store them (say, *prod*)
    2. For each word pair (w’) in the top 100 documents:
        1. $P(w'|M)$ can be computed
        2. Update the contribution (the products P(w’|M) and *prod*) of the particular document to the word pair (w’) through the dictionary mentioned in previous steps.

5. Having obtained $P(w'|R)$ for all pairs of words, use KL divergence to rank EACH document as:
    1. set curr_score = 0
    2. For each pair of word w’ present in the top 100 documents, find $P(w'|R)\log(P(w'|D))$ (*the former can be obtained from the dictionary created in the previous step and just the latter has to be computed for each pair*). Add these scores.
6. Having obtained the scores of each document, sort it to obtain the new ranks.
7. Output the ranked documents list in the format that trec_eval can understand.

Note: Dirichlet Smoothing is applied wherever necessary.

2.2. Observations

a. By taking w to belong to collection and then computing the rank, it took a couple for hours to process single query itself. Hence, it was replaced with w belongs to relevant documents only (without affecting the performance significantly).
b. Data like frequency of words and pair of words can be stored to the disk for reuse (*it was implemented partially for saving time*).

2.3. Performance Report

(Note that the performance testing were done on a subset of queries)

	original	unigram	bigram
nDCG	0.3307	0.3401	0.3480
MRR	0.2223	0.2379	0.2446

(a) Paired t-Test (2-tailed)

	nDCG			MRR		
	p	tstat	sd	p	tstat	sd
unigram	0.9648	-0.0442	0.3065	0.9041	-0.1205	0.3118
bigram	0.5690	-0.5748	0.2970	0.4249	-0.8071	0.3115

(b) Wilcoxon signed rank test

	nDCG			MRR		
	p	zval	signedrank	p	zval	signedrank
unigram	0.6843	0.4066	6.4335E+03	0.5924	0.5354	6507
bigram	0.4887	–	47	0.4887	–	47

2.4. Conclusion

Through this method, there was a small amount of improvement in performance which is visible from the metrics.

3. End Notes

- Statistical Significance Tests were conducted on MATLAB using `ttest2()` and `signrank()`.