IRDM Course Project Part I Report

Ziwei Lin
ID:19035341

ABSTRACT

Information retrieval model is crucial for passage ranking. Besides the basic retrieval models - Vector Space and BM25, more retrieval models for passage ranking are continuously emerging. This project report will first gives a literature review about the research in passage ranking model, then report the implementation of test pre-processing for Zipf's Law comparison, the inverted index generation, the Vector Space Modeling and BM25 Modeling to prepare for the building of a complete passage re-ranking system.

- Note: The implementation code in this Project Part I assignment is written in Python 3.7. Please make sure you have install or can import the following libraries when you run the code: "re", "math", "matplotlib.pyplot", "codecs", "pandas", "os", "datetime", "operator.itemgetter".

1 LITERATURE REVIEW

1.1 Introduction

Passage ranking is an essential topic in the field of information retrieval. With the enormous need of searching information nowadays, especially on the web search engine, the quality (includes the relevance extent and the reliability of the retrieved passages) and efficiency of passage ranking according to the user-input query are highly mattered. Some basic retrieval models such as the Vector Space Model and the BM25 Model, whose implementations are described in the later report, have proved their effects in passage ranking to some degree. Nevertheless, their ranking results still lack of satisfaction. Retrieval models are continuously researched and developed these years with some progress.

To investigate the emerging information retrieval models for passage ranking and enhance the knowledge about current retrieval model development, this literature review is conducted base on the topic of "passage ranking model". Twenty pieces of material are selected from Conference Proceedings like SIGIR and articles from journals like DBLP, after the year 2000 for reviewing.

This Literature Review section will summarise the main ideas and findings of each paper, categorise the papers by the characteristics of the models they introduced, identify the merits and drawbacks of the models with critical analysis, and may introduce some ideas for model improvement.

1.2 Categorisation and Analysis

From the summary in Table 1, the twenty papers can be classified as below:

- Implement existing models
  Paper 4 provides the implementation of the BERT model (Bidirectional Encoder Representations from Transformers) - which is a bidirectional language model - by calculating the probability of passage-query relevance in a single layer neural network ($BERT_{LARGE}$).

Table 1: Summary of Papers I

Paper ID	Main Ideas and Findings
1[8]	Different from models like BM25, deep neural IR models usually compare whole query with documents and used for late stage re-ranking, however, this paper applies a separate query term independence model to three deep neural IR models: BERT, Duet and CKNRM-. It finds that these deep models can be applied to offline ranking for large collection.
2[5]	This paper introduce CEDR (Contextualized Embeddings for Document Ranking), which joins the BERT classification vectors to existing neural model (as context) and outperform the baseline ranking (BERT ranking without current neural model).
3[13]	This paper trains BERT in a weak supervision framework and develops a weak supervised BERT-PR model which outperform BM25 remarkably and even beats the full supervised model on two test datasets.
4[9]	This paper describes a re-implementation of BERT for query-based passage re-ranking on TREC-CAR and MS MARCO.
5[7]	This paper describes some modifications to the original Duet model that can improve its performance in the MS MARCO task significantly.
6[2]	This paper illustrates an upgraded version of BM25 called BM25P, which computes a linear combination of term statistics in the different portions of news articles, and outperform BM25.
7[12]	This paper introduces a new model that elaborates information from inter-passage,inter-document, and query-based similarities from the initial ranking, and re-rank the collection to optimise the top ranking items.
8[11]	This paper introduces AQuPR, Attention based Query Passage Retrieval system for Web query. It trains a deep recurrent network with an attention mechanism using a database of human query with answer passages. The system improves the base search results remarkably.
9[4]	This paper looks into the representation of the relevance between a document and the query. It introduces semantic-graph that links notions extracted from documents with the relatedness between notions (edge), which can help enhance the performance of other retrieval models.
10[1]	This paper proposes a dynamic ranking algorithm that can interact with the user to dynamically build up a decision tree to satisfy the needs. It also provides an evaluation system and proves the approving performance of this dynamic ranking model.

The merit of BERT is remarkable, it outperforms other models in MS MARCO MRR@10 task and TREC-CAR test. It also retrieves passages with more novel words. However, BERT can also fails when the query is long and relatively poor with respect to query numerical and entity type questions.[10]

- Improve other models
  Paper 1 proposes query term independent model for training deep neural IR models and proves the possibility for BERT, Duet and CKNRM- to perform passage ranking task for large collections offline.
  In Paper 2, the introduced method CEDR is an approach join contextualised models like BERT with some neural ranking architectures to benefit the rankings.
  Paper 5 derives five variants of Duet v2 (which updated the architecture of Duet), perform testing on MS MARCO MRR@10 dataset and investigates that it can reduce the number of learnable parameters to decrease the training cost.
- Introduce completely new models
  Paper 7 introduces a language-model based model for re-ranking the candidate documents to elaborate the correctness of the top ranking item. The advantage for this model is that it fully utilises the information across passages and document along with passage/document-query similarity, so that it outperform other models that focus on passage and item similarity.
  In Paper 8, the advantage of AQuPR is that it can improve the search results by learning the query and target answer on the Web search. But the creation and use of the query-answer dataset is considerable. Paper 10 presents dynamic ranking algorithm by continuously expanding the policy tree by interacting with user to satisfy their query needs. This is similar to a simple learning algorithm. Although it outperform the static ranking, the focus of document item which can be a disadvantage that it is computationally hard for calculating the optimal policy tree.
- Introduce new models derived from existing ones
  In Paper 3, by training the BERT model with weak supervised framework, it obtains the BERT-PR model that outperform BERT. Paper 6 upgrades the traditional BM25 model to BM25P by introduce a linear relations across portions of news articles. The merit lies in the efficiency and quality of the result, and can be developed for other domain-specific query. Paper 9 introduces the Semantics-Enabled Language Model (SELM) for relative probability estimation between document and query by creating conception graph.

1.3 Conclusion

In conclusion, the sate-of-the-art retrieval models utilise the learning idea that leads to continuous improvement of the ranking results. The application and variation of BERT model is the most popular. The research of semantic analysis also takes a large proportion in the development of these models.

2 TEXT STATISTICS

The task for the Text Statistics step is to analyse the relationship between term frequency and term rank with the Zipf's law, using data obtained by performing text pre-processing on the source `"../dataset/passage_collection_new.txt"` (which contains a passage collection of 182469 lines, where each line stands for one passage). This section will give descriptions about the implementation of text pre-processing, the reasons for text pre-processing as well as the comparison between the result data and the Zipf's law.

2.1 Text Pre-processing

2.1.1 Implementation.

The object of this step is counting the frequency of each term of the passage collection, ranking them in descending order and then observe the actual rank-frequency relationship to see whether it consists with the Zipf's law. As no further process like passage ranking is required for this step, and only term and term frequency matter, the implementation of text pre-processing only conducts number and punctuation removing and alphabet lowercasing, without stopwords applying. It generates a dictionary to store the term - term frequency pairs and sort it for rank-frequency image plotting. The implementation details are described below. The implementation of this step is inside the "Text-Statistics" folder, which contains "preProcess.py", "proportion.txt", "rank-freq-raw.png", "rank-freq.png" and "rank-prob.png". The idea is: 1) Open the passage collection from the dataset folder, read all lines and join them as a single string. 2) With the help of regular expression support in "re", we replace all characters other than alphabet a-z A-Z with space, lower the case then split the string as a list of terms - **tList**. This is equal to the tokenisation step. 3) By iterating through the **tList** once, we store each term with its count in a dictionary - **tCountDict**, sort it by the term counts and store it as a list of tuples (count, term) in **sortedList**.

2.1.2 Reason for Text Pre-processing.

Text pre-processing operations include: tokenisation that transform a long text string into terms of words which can be treated as elements of recogniser of a text; number removal and characters lowercase can reduce the entries of terms to save memory space and speed up the query and ranking process; removing stopwords has the same effect; stemming can also reduce the entries of terms as it extracts the root of the words, however, stemming will cause the lose of too much information, so this implementation does not apply any type of stemming. The reason for performing text pre-processing in this Text Statistic step is to recognise the terms and calculate their frequency to evaluate the correctness of Zipf's law, so only tokenisation, number removal and characters lowercase is applied. While for the creation of the text re-ranking system, when generating the inverted index, stop words like articles and pronouns cannot help differentiate a passage, so removing is also applied which can further spare some memory and speed up the system.

2.2 Zipf's Law Implementation

After the text pre-processing above, a sorted term - frequency list (**sortedList**) is generated. Next step is to implement the Zipf's Law which reveal the relationship between the words' frequency rank and their frequency. One version of Zipf's law states that the i_{th} most common term and its collection frequency cf_i satisfy the following proportion:[3]

$$cf_i \propto 1/i \tag{1}$$

Equivalently, adding logarithm to both sides results in this equation:

$$log(cf_i) = log(c) - log(i) \quad (2)$$

The code "preProcess.py" prepares three lists: list of log(rank) - **logR**, log(frequency) of actual result - **f_real** and Zipf's law result - **f_zipf** to plot the Zipf's Law and Actual Values Comparison Graph (as Figure 1 presents) which shows the relationship between log(rank) and log(frequency). Using the **sortedList** from the preprocessing step, $log(cf_i)$ and log(i) are obtained, and a red line for Zipf's Law and green points of actual values can be drawn on the log(Rank) - log(Frequency) graph. By observing the first draft Figure 1 which plot the line for $log(cf_i)$ = - log(i).

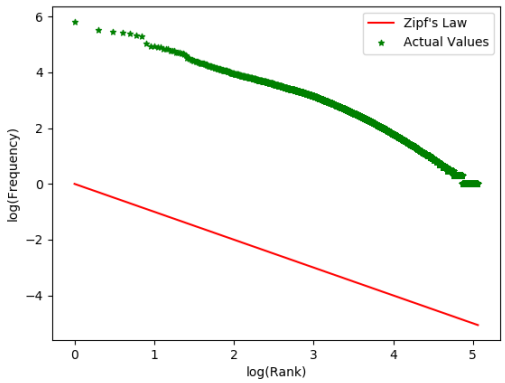

Figure 1: Zipf's Law and Actual values Comparison Represent by log(Rank)-log(Frequency) without constant parameter

"Six" can be a suitable value for log(c). Therefore, it finally plots the graph for the comparison of Zipf's Law and Actual Values, as Figure 2 shows. The trend of the green spots is consistent with the theoretical result as the red line represents, which proves the correctness of Zipf's Law. Another version of Zipf's Law focus on

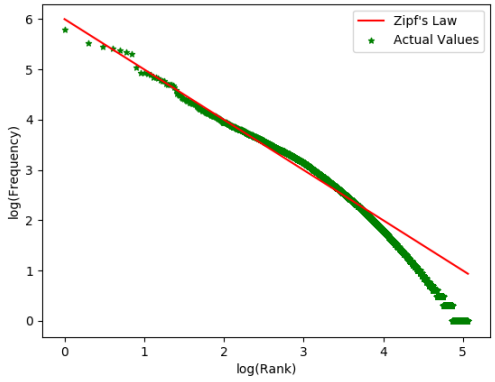

Figure 2: Zipf's Law and Actual values Comparison Represent by log(Rank)-log(Frequency)

the frequency rank and the probability of terms, and states that the probability of word occurrence P_r times its frequency rank **r** is a constant c, said:

$$r * P_r \approx c \quad (3)$$

The parameters reported in "proportion.txt" are each term's frequency(Freq), frequency rank(r), Proportion(Pr%) - Frequency/total number of words(length of **tList**), rank * Proportion(r*Pr). The snapshots are shown by figure 3. It can be observed that the values in the last column r*Pr are around 0.1 generally. Driven by this fact,

Term	Freq	r	Pr(%)	r*Pr
the	626948	1	6.097	0.061
of	334311	2	3.251	0.065
a	284646	3	2.768	0.083
and	255232	4	2.482	0.099
to	240995	5	2.344	0.117
is	216885	6	2.109	0.127
in	202270	7	1.967	0.138
for	108171	8	1.052	0.084
or	86935	9	0.845	0.076
you	86662	10	0.843	0.084

Figure 3: Parameters of Top 10 Terms in Proportion.txt

the code also creates another three lists: **r** - the rank, **p_real** - the real proportion (Pr) and **p_zipf** - the Zipf's Law proportion (Pr = 0.1 / r) to plot a rank - probability graph as Figure 4 illustrates.

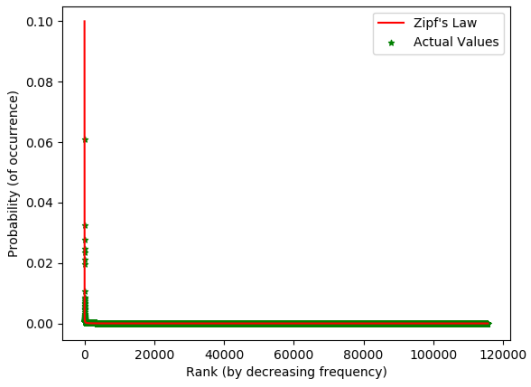

Figure 4: Rank-Probability Graph for Zipf's Law and Real Values Comparison

In conclusion, all the above real data proves that the Zipf's Law is correct and can be used for the estimation of term frequency, occurrence probability with the rank. The running time of "preProcess.py" is about 15 seconds on the developer's computer, which is approving.

3 INVERTED INDEX

This section describes the idea and implementation of inverted index based on "../dataset/candidate_passages_top1000.tsv", where

each query contains 1000 candidate passages generally. The code files are inside the "Inverted-Index" folder, the result files are placed in the "invertedResult" folder.

3.1 Extract Candidate Passages

The Inverted Index step asks for generating inverted indexes for the candidate passages of each of the 200 queries. Since the candidate_top1000.tsv contains (qid pid query passage) lines, groups them by (pid passage) without order, for the convenience of further data processing, passage extraction is implemented first. The code "ExtractPassages.py" first reads candidate _passages_top1000.tsv using pandas as DataFrame format. Then, it iterates through all unique qid, creates a dictionary **candidateDict** whose key is "qid" and value is a list of "pid passage" by selecting lines with the same qid out. Furthermore, it looks through the keys of **candidateDict** to create a separate directory for each query with "passages + qid.tsv" written and output inside it. The extraction and generation of qid folders and candidate passages files properly arrange the outputs by query, which assists further data accessing and processing for the re-ranking task as well as testing. The running time of "ExtractPassages.py" is satisfying which is around 3 seconds on the developer's computer.

3.2 Generate Inverted Index

The code "inverted_index.py" implements the generation of inverted indexes for 200 queries and stores them inside the corresponding qid folder. It defines the inverted index in a class. In the main body (outside the class), it first opens "test-queries.tsv" to obtain the 200 qids for iteration. Then, it changes the working directory to the current qid folder, instantiates an Inverted_index object. Finally, it creates the inverted index, stores the entries of "Term: [(pid, term frequency)]" and writes it in "invIndex + qid.tsv". **The exact term frequency in each passage is stored as additional information in the inverted index, because such information is highly useful when creating the ranking models (for both vector space and BM25).**

The "createInvertedIndex()" function opens the "passages + qid.tsv" and tokenises each line of passage in a pre-processing step. This step includes removing characters other than a-z A-Z space and hyphen"-" (hyphen is kept because it can be part of a single word, for instance, "re-ranking" is different from "re" + "ranking" and is more distinguishable); lowering the case to reduce the entries number and save memory, and split the string by space; removing 46 stopwords include articles, be and some pronouns since these words are too common and not helpful for differentiating passages when ranking; removing the terms that only contains hyphens.

After the pre-processing, it obtains a dictionary **tokenPDict** that stores (pid: list of passage words) pair. Then, it initialise a inverted index dictionary term: list of (pid, term count), for every pid in **tokenPDict**, it creates a dictionary **freqDict** for storing (term: term count) pair. Next, for each term in **freqDict**, it appends the current pid and the term count to the corresponding term entry of the inverted index. Finally, it sorts the inverted index by the alphabetical order of the terms and writes to "invIndex + qid.tsv" file in the correcponding qid folder. Here is a snapshot of one inverted index:

```
ability:    [(311061, 1)]
able:   [(1868425, 2), (8214867, 1), (3077645, 1),
abnormal:   [(5736595, 1)]
about:  [(2689153, 3), (8336020, 3), (5855684, 3),
above:  [(4981041, 2), (7489873, 1), (2848653, 1),
above-average:  [(2277155, 1), (2277154, 1)]
abrasion:   [(4096598, 1)]
abrasive:   [(1691035, 1)]
absolutely: [(1391573, 1), (4266263, 1), (8214864,
absurdly:   [(6345913, 1)]
abuse:  [(8336018, 2), (8336017, 1), (3431663, 1),
```

Figure 5: Snapshot of part of an Inverted Index File

To speed up the counting process, a countFreq() function is defined whose mechanism is iterate the list of passage words only once, if the term does not exist in **freqDict**, set the **freqDict**[term] to 1, otherwise, increase the value of key term by 1. The time complexity is O(n) which is much faster than using "list.count(i) for i in list" whose time complexity is $O(n^2)$. The total running time of "inverted_index.py" is about 1.5 minutes on the developer's computer, which is acceptable.

4 BASIC RETRIEVAL MODELS

(How to implement, what parameter is used for BM25) In the Basic Retrieval Models step, two passage ranking models are implemented: the vector space model and the BM25 model. This section describes the implementation details about modeling using the candidate passages and inverted indexes generated in the previous step to re-rank the candidate passages for each query.

4.1 Vector Space

The notion of vector space model is by treating each term as an axis, the term frequency as the value on this axis, both passages and queries can be represented as vectors in a high-dimensional space. The similarity between a passage vector and a query vector determines their relevance degree. This similarity score is defined as: for query q and document d,

$$sim(q, d) = \Sigma_{t \in q \cap d} tf_t \times idf_t \tag{4}$$

In this equation, **t** is the term contains in both the query and the passage, tf_t is the term frequency (term count) of the term t in the current passage, idf_t is the inverse document frequency of term **t**, which is defined as:

$$IDF_t = log_{10}(N/n_t) \tag{5}$$

where N is the total number of passages in the collection and n_t is the number of passages that contains term **t**. The implementation for Vector Space Model:

(1) It load queries data from "../dataset/test-queries.tsv", creating file "VS.txt" (VSFile) for storing the VS score and ranking results, creating file "VSRepre.txt" (VSRepre) for storing the vector representation of each candidate passage under the query.

(2) It iterates through 200 queries, perform the same pre-processing operations (split strings, remove number, punctuation, stopwords and terms with hyphens only) on the query, store the query terms in **qTerms** and **qTs** (without duplication).
(3) It retrieves the inverted index and candidate passages from current qid folder inside "../invertedResult" and keeps **vsScore** list to store (pid, score) tuples.
(4) By iterating through each candidate passage, perform the same pre-process again, remove duplication, it results in a list **pTs** of unique terms in the current passage.
(5) For every term in **qTs**, if the term is not contained by **pTs**, it write value 0 to VSRepre.txt under the corresponding term as its axis value. Otherwise, it obtains the tf_t value from the inverted index's "term: (pid, tf)" entry. The df_t (n_t in the equation (5)) value is the length of (pid, tf) list, the **N** value is the length of candidate passages file, the idf_t is obtained with equation (5). Then, the **vsScore** for the current passage can be calculated with equation (4) by adding tf * idf of all terms in **qTs** and **pTs**. The tf * idf values are also written in VSRepre.txt simultaneously.
(6) After sorting the **vsScore** list by scores in descending order, it finally writes the **VSFile** in the required format through the **vsScore** list. Figure 6 shows a snapshot of VSRepre.txt, start with "qid, term1, term2 ..." the following lines shows "pid, term1 score, term2 score ...". Figure 7 shows a snapshot of VS.txt with "qid, A1, pid, rank, score, VS".

```
**********************************
20455   ar      glasses definition
7130335 0.0     0.0     0.0
7130336 0.0     0.0     0.0
8411362 0.0     0.624   0.0
8428575 0.0     0.624   0.0
8027220 0.481   0.0     2.458
7160425 0.0     0.0     0.0
7166509 0.0     0.416   0.0
804813  0.962   0.0     0.0
804818  0.962   0.0     0.0
7214862 0.0     0.0     0.0
```

Figure 6: Snapshot of Vector Representation

```
11096   A1      5638217 1       8.998   VS
11096   A1      1898573 2       6.763   VS
11096   A1      472573  3       6.472   VS
11096   A1      403335  4       6.254   VS
11096   A1      8724971 5       6.181   VS
11096   A1      8724969 6       6.072   VS
11096   A1      472569  7       6.072   VS
11096   A1      6195490 8       6.036   VS
11096   A1      7537783 9       5.854   VS
```

Figure 7: Snapshot of VS.txt

4.2 BM25

As for BM25, which stands for "Best Match", is a probabilistic weighing scheme whose algorithm is based on binary independence model but focuses on term frequency and document length[]. The formula for calculating the score is:

$$\Sigma_{i\in Q} log \frac{\frac{r_i+0.5}{R-r_i+0.5}}{\frac{n_i-r_i+0.5}{N-n_i-R+r_i+0.5}} \cdot \frac{(k_1+1)f_i}{K+f_i} \cdot \frac{(k_2+1)qf_i}{k_2+qf_i} \quad (6)$$

Where the K above has an expression:

$$K = k_1((1-b)+b \cdot \frac{dl}{avdl}) \quad (7)$$

Here are the descriptions of necessary parameters: qf_i - the number of term i in the query; r_i **and R** - the relevance information, as no such information provided currently, r_i = R = 0; **N** - the document total number, length of candidate passages file of a query; n_i - the document frequency (df) of term i in the query, number of passages in the inverted-index entry "term i"; f_i - the term frequency of term i in the current passage; **dl/avdl** - document length / average document length, words count of the current passage / average words count of current passage collection. k_1,k_2 - the tuning parameters, from current experiments on some test set, the optimised value for k_1 lies in [1.2, 2] and k_2 lies in [0, 1000]. This implementation sets k_1 = 1.2, k_2 = 100. **b** - another parameter to determine the extent of document scaling, whose value lies in [0, 1], the optimised value is 0.75.[6]

The implementation steps for BM25 begin with step (1) to (4) of Vector Space implementation, instead of VS.txt and VSRepre.txt, **BM25.txt** is created and **bmScore** is used to store the (pid, score) tuples. Just before step (5), it set k_1, k_2, **b, r, R, N and also the avdl** (words count of the whole passage collection / N) first. **dl** is obtained as the length of split string of passage content. In step (5), for term in both **qTs** and **pTs**, f_i is the tf in inverted index, qf_i is count from **qTerms** is step (2), n_i is identical with df in Vector Space, the length of list of passages in the current term entry in the inverted index. Then, in step (6), it can calculate the **bmScore** with equation (6) and write the result to BM25.txt. Figure 8 is a snapshot of BM25.txt:

```
11096   A1      7700305 1       4.496   BM25
11096   A1      5638217 2       3.374   BM25
11096   A1      2719750 3       3.318   BM25
11096   A1      2942747 4       3.168   BM25
11096   A1      4291652 5       2.98    BM25
11096   A1      3201151 6       2.79    BM25
11096   A1      472573  7       2.23    BM25
11096   A1      5638282 8       2.168   BM25
11096   A1      8244357 9       2.147   BM25
11096   A1      8724971 10      2.136   BM25
```

Figure 8: Snapshot of BM25.txt

Both of the two models are implemented in the code "models.py" as they share some parameters. The running time is about 3 minutes on the developer's computer, which is acceptable.

REFERENCES

[1] Christina Brandt, Thorsten Joachims, Yisong Yue, and Jacob Bank. 2011. Dynamic Ranked Retrieval. 247–256. https://doi.org/10.1145/1935826.1935872

[2] Matteo Catena, Ophir Frieder, Cristina Ioana Muntean, Franco Maria Nardini, Raffaele Perego, and Nicola Tonellotto. 2019. Enhanced News Retrieval: Passages Lead the Way!. In *SIGIR'19*.

[3] Combridge University Press. 2009. *Zipf's law: Modeling the distribution of terms*. https://nlp.stanford.edu/IR-book/html/htmledition/zipfs-law-modeling-the-distribution-of-terms-1.html Accessed: 2020-2-24.

[4] Faezeh Ensan and Ebrahim Bagheri. 2017. Document Retrieval Model Through Semantic Linking. 181–190. https://doi.org/10.1145/3018661.3018692

[5] Sean MacAvaney, Andrew Yates, Arman Cohan, and Nazli Goharian. 2019. CEDR: Contextualized Embeddings for Document Ranking. *CoRR* abs/1904.07094 (2019). arXiv:1904.07094 http://arxiv.org/abs/1904.07094

[6] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press, USA.

[7] Bhaskar Mitra and Nick Craswell. 2019. An Updated Duet Model for Passage Re-ranking. *CoRR* abs/1903.07666 (2019). arXiv:1903.07666 http://arxiv.org/abs/1903.07666

[8] Bhaskar Mitra, Corby Rosset, David Hawking, Nick Craswell, Fernando Diaz, and Emine Yilmaz. 2019. Incorporating Query Term Independence Assumption for Efficient Retrieval and Ranking using Deep Neural Networks. *CoRR* abs/1907.03693 (2019). arXiv:1907.03693 http://arxiv.org/abs/1907.03693

[9] Rodrigo Nogueira and Kyunghyun Cho. 2019. Passage Re-ranking with BERT. *CoRR* abs/1901.04085 (2019). arXiv:1901.04085 http://arxiv.org/abs/1901.04085

[10] Harshith Padigela, Hamed Zamani, and W. Bruce Croft. 2019. Investigating the Successes and Failures of BERT for Passage Re-Ranking. *CoRR* abs/1905.01758 (2019). arXiv:1905.01758 http://arxiv.org/abs/1905.01758

[11] Parth Pathak, Mithun Das Gupta, Niranjan Nayak, and Harsh Kohli. 2018. AQuPR: Attention based Query Passage Retrieval. 1495–1498. https://doi.org/10.1145/3269206.3269323

[12] Eilon Sheetrit. 2018. Utilizing Inter-Passage Similarities for Focused Retrieval. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval, SIGIR 2018, Ann Arbor, MI, USA, July 08-12, 2018*, Kevyn Collins-Thompson, Qiaozhu Mei, Brian D. Davison, Yiqun Liu, and Emine Yilmaz (Eds.). ACM, 1453. https://doi.org/10.1145/3209978.3210222

[13] Peng Xu, Xiaofei Ma, Ramesh Nallapati, and Bing Xiang. 2019. Passage Ranking with Weak Supervsion. *CoRR* abs/1905.05910 (2019). arXiv:1905.05910 http://arxiv.org/abs/1905.05910