Report for assignment 1 of COMP0084

Anonymous ACL submission

Abstract

Information retrieval models are an important part of many applications such as search, question answering and recommendation. In this assignment, an information retrieval model that solves the paragraph retrieval problem was developed. Specifically, we implement that for a given number of queries, a sorted list containing a number of relevant paragraphs is returned, based on different theories.

In deliverables 1 to 2, we perform text preprocessing and a simple analysis of the built vocabulary for a given passages. In deliverables 3 to 4, we construct an inverted index based on the glossary. In deliverables 5 to 7, we built two information retrieval models: i.e. for the given 200 queries, candidate passages were re-ranked by means of TF-IDF-based cosine similarity and BM25 similarity, respectively. Finally, in deliverables 8 to 12, we have used Laplace estimation, Lindstone correction and Dirichlet smoothing, respectively, to model the query likelihood language and to analyse the parameter taking. The source code for the above deliverables, as well as the output, are included in other files in the folder.

1 Deliverable 1

Based on the data in passage-collection.txt, we performed text pre-processing and briefly analysed the vocabularies created.

First, we extracted terms (1-grams) from the original text. Specifically, we designed the TextPreprocessing function in Task1.py, which for the original input paragraph, lowercased it and removed the non-text symbols from it, before splitting it into tokens. in addition, with the help of nltk.stem, we also provided a list of terms using the WordNetLemmatizer and SnowballStemmer to preprocess the text for Lemmatizing and Stemming, which we have not done in this job for the sake of speed (the code has been commented out). In addition, we did not remove the stop words from it.

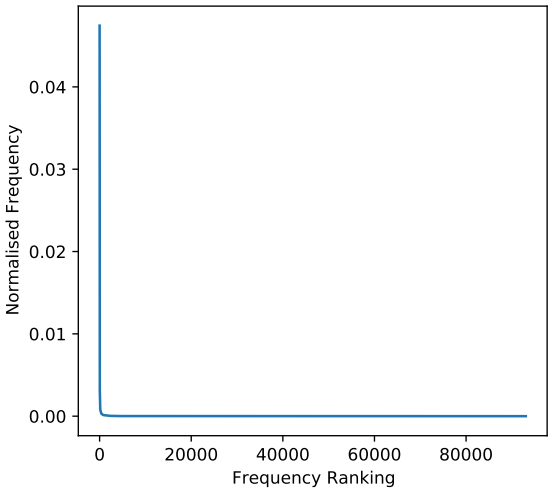

Figure 1: Schematic of normalised frequencies for terms

Based on the above text preprocessing steps, we determined the total number of terms to be 93,041. We then implemented the StatisticalFrequency function, which was used to calculate the number of occurrences of the terms in the provided dataset and plotted their probability of occurrence (normalised frequency) against the frequency ranking, as shown in Figure 1.

As can be seen, for the vast majority of terms in the vocabulary (over 99.9%), the probability of occurrence is less than 0.01%, while the probability of occurrence for the top 0.1% of terms is higher than 1%, which proves that the probability distribution obeys Zipf's law.

To make it more obvious that the terms in this dataset follow Zipf's law, we plotted a log-log comparison schematic of the actual data and Zipf's law, as shown in Figure 2.

Looking at Figure 2, we can see that the paragraph data in passage-collection.txt generally conform to the distribution pattern of Zipf's law, but there are slight variations. For example, for terms

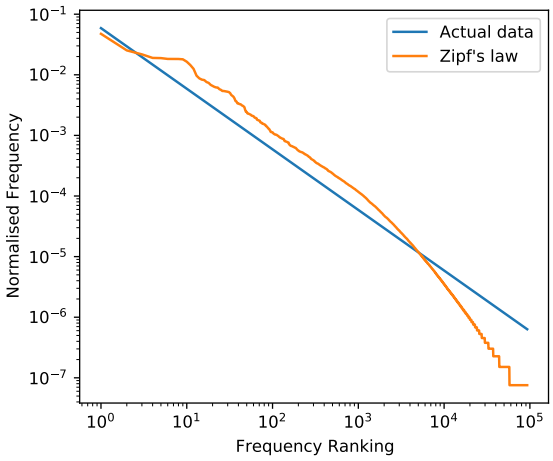

Figure 2: Normalised frequencies of Zipfś law compared to actual data

with frequencies between 10 and 5000, the actual data is normalised more frequently than expected, while for most of the lower frequency data after 10000, the actual data is much lower than expected by Zipf's law. The reason for this may come from the fact that during text pre-processing we removed most of the non-text symbols but did not perform Lemmatizing and Stemming operations related to them.

2 Deliverable 2

The source code in Deliverable 1 has been committed with the folder named task1.py.

3 Deliverable 3

In order to retrieve the passages in an efficient manner, we created an inverted index based on the passages provided in candidate-passages-top1000.tsv and the vocabulary identified in Deliverable 1.

Specifically, we first read the data in candidate-passages-top1000.tsv and removed any duplicate passages from it. After that, we designed the GenerateInvertedIndex function to generate the inverted index. In this function, all the passages in the input variable np_passage_data are first iterated through, and for each of them, the text is first pre-processed to split it into tokens, and then, for each token, it is checked if it exists in the inverted index of the dictionary type, and the pid of the token, the token, the number of occurrences of that token in that passage and the total number of words in that passage are stored in the dictionary. Finally, the dictionary storing the pid corresponding to all terms and the passage in which they occur, the number of occurrences of the word, and the total number of words in that passage is returned.

4 Deliverable 4

The source code in Deliverable 3 has been committed with the folder named task2.py.

5 Deliverable 5

For this task, we built a TF-IDF based retrieval model.

First, for each paragraph in candidate-passages-top1000.tsv, we extracted the TF-IDF vector representation of the paragraph using inverted indexing. Specifically, we designed the GenerateTfIdfVectors function, which calculates the corresponding IDF value (i.e. log(total number of passages/number of passages containing the term)) for each term in the inverted index, and the TF value (number of occurrences of the term/total number of passages containing the term) for each record in its inverted index. of the term/total number of words in the passage). Finally, the product of TF and IDF is added to the TF-IDF vector of passages.

Afterwards, we created a TF-IDF vector for each query based on the TF-IDF representation of the passages. Similarly, we designed the GenerateTfIdfQueryVectors function to calculate the TF-IDF value of the query in the passage collection and added this value to the TF-IDF vector of the query.

After that, we retrieved up to 100 passages from the 1000 candidate passages of each query of test-queries.tsv using the basic vector space model of TF-IDF and cosine similarity, and designed the CalculateCosSimilarity function to calculate the cosine similarity between query and passages. and sorted them in reverse order.

Finally, we stored the results in a file called tfidf.csv. tfidf.csv has the following format.

qid,pid,score

where qid is the query identifier, pid is the paragraph identifier, and score is their cosine similarity score. qid,pid corresponds to the highest to lowest similarity score, i.e. starting with the highest similarity score.

6 Deliverable 6

Similarly, for this task, we build a retrieval model based on BM25.

First, we implemented BM254 using an inverted index. specifically, we designed the Calcu-

lateBM25Similarity function to calculate the BM25 similarity corresponding to passages and queries, setting k1=1.2, k2=100 and b=0.75.

Up to 100 passages were retrieved from the 1000 candidate passages for each test query, and the BM25 similarity between query and passage was calculated and sorted in reverse order. The results were stored in a file called bm25.csv.

7 Deliverable 7

The source code in Deliverable 5 and 6 has been submitted with the folder named task3.py.

8 Deliverable 8

In this task, we implemented a query likelihood language model using Laplace smoothing based on test-queries.tsv and candidate-passages-top1000.tsv. Specifically, we used the CalculateLaplaceSimilarity function to calculate parameters such as the number of passage words, the length of the vocabulary, and the number of occurrences of term in the passages, and calculated their likelihood based on the Laplace smoothing principle.

After that, we retrieved 100 passages from the 1000 candidate passages for each test query and calculated the likelihood corresponding to their query and passages respectively. The final results are stored in the laplace.csv file.

9 Deliverable 9

In this task, we implemented a query likelihood language model based on test-queries.tsv and candidate-passages-top1000.tsv, using Lidstone correction. Specifically, we used the CalculateLidstoneSimilarity function, setting = 0.1, to calculate parameters such as the number of passages words, the length of the vocabulary, the number of occurrences of term in the passages, and the TF, and calculated their likelihood based on the Lidstone correction principle.

Afterwards, we retrieved 100 passages from the 1000 candidate passages for each test query and calculated the likelihood corresponding to their query and passage respectively. The results are then stored in the lidstone.csv file.

10 Deliverable 10

In this task, we implemented the query likelihood language model using Dirichlet smoothing. Specifically, we used the CalculateDirichletSimilarity function to calculate parameters such as the number of passage words, the length of the vocabulary, and the number of occurrences of term in the passage, and set = 50 to calculate its likelihood based on the Dirichlet smoothing principle.

After that, we retrieved 100 passages from the 1000 candidate passages for each test query and calculated the dirichlet likelihoods corresponding to their queries and passages respectively. In the end, the results were stored in the dirichlet.csv file.

11 Deliverable 11

1. Which language model do you expect to work better?

Answer: I expect the language model based on Dirichlet smoothing to have better performance.

2. Which would be expected to be more similar and why?

Answer: I would expect Laplace smoothing and Lidstone correction to give similar results. This is because Lidstone only modifies the weight of invisible terms in similarity compared to Laplace smoothing.

3. Comment on the value of = 0.1 in the Lidstone correction - is this a good choice and why?

Answer: This is not a good choice. In the Laplace smoothing method we give too much weight to the invisible items and therefore need to reduce their weights to achieve regularisation. In this text base, the average total number of words in the passages is 57 and the length of the vocabulary is 93,041, so setting = 0.1 is not enough to satisfy the regularisation requirement.

4. If we set =5000 in the Dirichlet smoothing method, would this be a more appropriate value and why?

Answer: This is not a good choice. Because the average number of words in the passage in this text base is about 57, setting =5000 at this point undoubtedly gives huge weight to the parameter of the importance of the word in the set, while ignoring the importance of the word in the passage, which would introduce an imbalance and even stop smoothing.

12 Deliverable 12

The source code in Deliverable 8 to 11 has been submitted with the folder named task4.py.