Information Retrieval and Data Mining Coursework 1

Anonymous

1 Introduction

The end-to-end running time of all four py files is about 35-45 minutes using a Ryzen 9 5900HS laptop. To create the conda environment for running the codes, use the line:

conda create -n mining_cw python matplotlib numpy nltk

Put the three given data and the four py files in the same folder and there should be no problem running all four py files. The py files might produce additional data files for the next task.

2 Deliverable 1

2.1 Pre-processing choice

For this section, we are told not to remove any stop words. Thus, the pre-processing follows the steps below,

1. Read passage-collection.txt into a single string. This is to speed up the reading. Since the goal is to analyze the whole text, process the file row by row will be unnecessary.

2. Convert all characters to lower cases.

3. Replace all \n with spaces. These \n are initially used to separate the document into lines.

4. Replace all punctuation with spaces. In the original file, some punctuation are misused and spaces after punctuation are missing. Removing the punctuation directly will cause two individual words to stick together and create an odd vocabulary.

5. Retain lowercase alphabets only. There are a lot of non-alphabet characters. Some could be typos and some could be errors. Removing numbers and those unreadable characters could provide a cleaner index of terms.

6. Tokenisation to turn the long string into terms.

7. Lemmatize the terms. Get the base form of the terms.

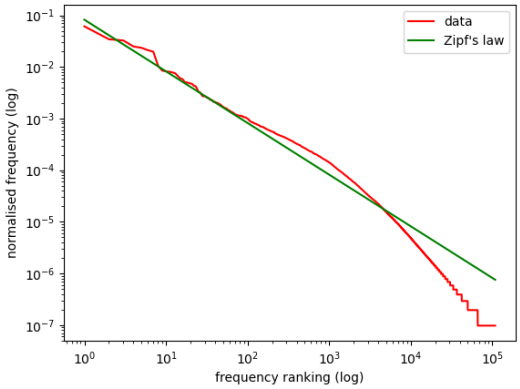

Figure 1: Term normalised frequency vs term ranking compared with Zipf's law

2.2 Size of the identified index of terms

The code identified 108680 different terms.

2.3 Zipf's law

Zipf's law is given as,

$$f(k;s,N) = \frac{k^{-s}}{\sum_{i=1}^{N} i^{-s}} \tag{1}$$

Where

$f(\cdot)$ = the normalised frequency

k = term's frequency rank

N = total number of terms

s = distribution parameter (equals 1 in our case)

The comparison of our data and Zipf's law is shown in Fig 1. Our empirical distribution follows the theory fairly well except low frequency ranking terms.

From Eq.1 we can see that the denominator is fixed, so the only factor that affects the distribution is k^{-1}. The main issue in our data is that after all the pre-processing, there will be a lot of individual terms that only appear several times (some could be typos in the origin file). The rare terms result in the main difference shown in Fig.1.

3 Deliverable 2

A task1.py file is submitted. After running the code, there should be one key print out (identified terms: 108680) and one saved file (ZipLaw.png).

4 Deliverable 3

4.1 Pre-processing

Similar pre-processing choices are explained before, so only unseen methods are explained thoroughly.

1. Convert all characters to lower cases.
2. Replace all \n with spaces.
3. Replace all punctuation with spaces.
4. Retain lowercase alphabets only.
5. Tokenisation.
6. Lemmatisation.
7. Removal of stop-words and single alphabet terms. Stop-words are mostly meaningless. Single alphabet terms are usually typos which are also meaningless.

4.2 Generating inverted index

To build the inverted index, I created a nested dictionary. The outer dictionary saves all the different terms as the keys. Each term is also a dictionary, where the keys are the pid of the passages that include the term, and the values are the occurrence of the term. Following is the work flow of generating the inverted index.

1. Read candidate-passages-top1000.tsv by rows (different rows represent different qid-pid pairs, which might include repeating passages).

2. For each row, examine whether the corresponding passage appeared before. If not, preprocess the passage. The code keep track of the number of processed passages, which should match the rows in passage-collection.txt (182469 passages)

3. Move on to the inverted index dictionary. For each term in the processed passage, either find the key term or create a new key term in the inverted index dictionary.

4. The key terms are also dictionaries. In these dictionaries, store the pid as the key and the occurrence of the term in the passage as the value.

After the given step, we will have a dictionary full of terms. By accessing these terms, we can find which passages contain these terms and the occurrence.

5 Deliverable 4

A task2.py file is submitted. After running the code, there should be one key print out (Total processed passages: 182469) one saved file (inverted_index.pkl) and print outs that confirm the saving.

6 Deliverable 5

6.1 Pre-processing

The pre-processing is identical to previous section.

6.2 TF-IDF vector and cosine similarity

To get a TF-IDF vector of a query/passage, we need to get the term frequency (TF) and the Inverse Document Frequency (IDF). TF is the number that the term appears in the query/passage, and IDF can be calculated by,

$$IDF_t = \log_{10}\left(\frac{N}{n_t}\right) \tag{2}$$

Where

N = number of total passages

n_t = number of passages in which term t appears

Note that the inverted index could speed up the search of n_t. Also, there might be cases that the term in the query do not exist in the inverted index. We will skip this term in these cases.

To store the TF-IDF vector, initialize a zero-vector with the size of total terms, where each index of the zero-vector represents a term. Store the TF-IDF value of the term in the corresponding index. After calculating the TF-IDF vector of the query and the passage, we can proceed to calculate the cosine similarity.

Cosine similarity is the dot product divided by the cross product of two vectors. Which could easily be calculated.

6.3 Calculating TF-IDF scores

The following steps are carried out to calculate TF-IDF scores of all queries.

1. For each query, find the matching passages by checking the qid (at most 1000).

2. For each matching passage, calculate the TF-IDF scores with the query. Finally, store the top 100 scores. The stored information includes the score and the pid.

3. Save the stored data to a csv file. Each row of the saved file includes qid,pid,score with no spaces. The qid order is identical to the order

in test-queries.tsv. The pid order starts from the highest scores to the lowest scores.

7 Deliverable 6

7.1 Pre-processing

The pre-processing is identical to previous section.

7.2 BM25

For our situation, relevance feedback information is not given, therefore the equation for our BM25 model could be written as,

$$\sum_{i \in Q} \left(\log \frac{1}{(n_i + 0.5)\,(N - n_i + 0.5)} \right) \left(\frac{(k_1 + 1)\, f_i}{K + f_i} \right) \left(\frac{(k_2 + 1)\, qf_i}{k_2 + qf_i} \right) \tag{3}$$

$$K = k_1 \left((1 - b) + b \cdot \frac{dl}{avdl} \right) \tag{4}$$

Where

Q = number of unique terms in query
N = number of total passages
n_t = number of passages in which term t appears
f_i = term occurrence in passage
qf_i = term occurrence in query
k_1 = 1.2, k_2 = 100, b = 0.75

7.3 Calculating BM25 scores

The following steps are carried out to calculate BM25 scores of all queries.

1. For each query, find the matching passages by checking the qid (at most 1000).

2. For each matching passage, calculate the BM25 scores with the query. Finally, store the top 100 scores. The stored information includes the score and the pid.

3. Save the stored data to a csv file. Each row of the saved file includes qid,pid,score with no spaces. The qid order is identical to the order in test-queries.tsv. The pid order starts from the highest scores to the lowest scores.

8 Deliverable 7

A task3.py file is submitted. After running the code, there should be two saved files (tfidf.csv, bm25.csv) and print outs that confirm the saving.

9 Deliverable 8

9.1 Pre-processing

The pre-processing is identical to previous section.

9.2 Laplace smoothing

Laplace smoothing gives weights to unseen terms. For each unique terms in the query, find the probability of the term appearing in the passage. Apply Laplace smoothing to avoid getting 0 probability for unseen terms. The Laplace estimates of each query term appearing in the passage is,

$$\frac{m_1 + 1}{D + V}, \frac{m_2 + 1}{D + V}, \cdots, \frac{m_V + 1}{D + V} \tag{5}$$

Where

m = the occurrence of terms in passage
D = the number of terms in the passage
V = the number of unique terms

9.3 Calculating Laplace scores (in log)

The following steps are carried out to calculate Laplace probability scores of all queries.

1. For each query, find the matching passages by checking the qid (at most 1000).

2. For each matching passage, calculate the Laplace probability scores with the query. Finally, store the top 100 scores. The stored information includes the score (in log) and the pid.

3. Save the stored data to a csv file. Each row of the saved file includes qid,pid,score with no spaces. The qid order is identical to the order in test-queries.tsv. The pid order starts from the highest scores to the lowest scores.

10 Deliverable 9

10.1 Pre-processing

The pre-processing is identical to previous section.

10.2 Lidstone correction

The problem with Laplace smoothing is that adding 1 to unseen terms is adding a lot of weights. Lidstone correction tries to fix this by adding a smaller term, ε. Thus, the Lidstone correction estimates could be written as,

$$\frac{m_1 + \varepsilon}{D + \varepsilon V}, \frac{m_2 + \varepsilon}{D + \varepsilon V}, \cdots, \frac{m_V + \varepsilon}{D + \varepsilon V} \tag{6}$$

10.3 Calculating Lidstone scores (in log)

The following steps are carried out to calculate Lidstone probability scores of all queries.

1. For each query, find the matching passages by checking the qid (at most 1000).

2. For each matching passage, calculate the Lidstone probability scores with the query. Finally,

store the top 100 scores. The stored information includes the score (in log) and the pid.

3. Save the stored data to a csv file. Each row of the saved file includes qid,pid,score with no spaces. The qid order is identical to the order in test-queries.tsv. The pid order starts from the highest scores to the lowest scores.

11 Deliverable 10

11.1 Pre-processing

The pre-processing is identical to previous section.

11.2 Dirichlet smoothing

The main problem with previous two methods is that adding 1 or adding ε treats all terms equally. However, the occurrence of different terms are different. Therefore, a better way of smoothing could be adjusting the smoothing according to the context.

$$\sum_{i \in Q} \log \left(\frac{D}{\mu + D} \cdot \frac{fq_i}{D} + \frac{\mu}{\mu + D} \cdot \frac{cq_i}{t_{total}} \right) \quad (7)$$

Where

Q = number of unique terms in query
D = the number of terms in the passage
μ = average passage terms
fq_i = term occurrence in query
cq_i = term occurrence in passage
t_{total} = total terms in all the passages

11.3 Calculating Dirichlet scores (in log)

The following steps are carried out to calculate Dirichlet probability scores of all queries.

1. For each query, find the matching passages by checking the qid (at most 1000).

2. For each matching passage, calculate the Dirichlet probability scores with the query. Finally, store the top 100 scores. The stored information includes the score (in log) and the pid.

3. Save the stored data to a csv file. Each row of the saved file includes qid,pid,score with no spaces. The qid order is identical to the order in test-queries.tsv. The pid order starts from the highest scores to the lowest scores.

12 Deliverable 11

12.1 The better model

Dirichlet smoothing is expected to perform better, as it suites short queries well. The three models are compared with BM25, the possibly best model we used, and the result in table 1 confirms that Dirichlet is the best of all.

Table 1: Number of same passages identified in the three models and BM25

Top n psg	bm25 vs lap	bm25 vs lid	bm25 vs dir
30	2	4	20
50	5	14	40
100	7	50	86

12.2 Similar models

Lidstone is the improved version of Laplace, and Dirichlet is the further improved method. Thus, either Lidstone and Laplace are more similar, or Lidstone and Dirichlet are more similar. The results in table 2 confirmed this.

Table 2: Number of same passages identified between three models

Top n psg	lap vs lid	lap vs dir	lid vs dir
30	11	4	11
50	21	6	21
100	42	10	63

12.3 Lidstone correction with $\varepsilon = 0.1$

The ε value in Lidstone correction is to avoid the large parameter in Laplace smoothing. As shown in table 1, assuming BM25 is a perfect model, the correctly identified passages increased from 7 to 50, which is a significant improvement. Therefore setting $\varepsilon = 0.1$ is a good choice but there might be spaces to improve if we fine tune the value.

12.4 Setting $\mu = 5000$ in Dirichlet smoothing

This adjustment is not appropriate. The μ value in Dirichlet smoothing is the average text length of the passages. The average passage length in the file passage-collection.txt before pre-processing is roughly 57 words, which is closer to 50 instead of 5000.

13 Deliverable 12

A task4.py file is submitted. After running the code, there should be three saved files (laplace.csv, lidstone.csv, dirichlet.csv) and print outs that confirm the saving.