Information Retrieval and Data Mining (COMP0084)
Coursework 1

General Notes

The code for this coursework is divided into four Python scripts, one for each task: "task1.py", "task2.py", "task3.py", and "task4.py". The four scripts are designed to be run in order, for better understanding, since later scripts build on code presented in earlier ones. Moreover, "task2.py" generates an inverted index that is saved and then imported and used in subsequent scripts "task3.py" and "task4.py". Specific programming choices are commented in the actual scripts, but the marker should refer to this report for any other clarifications such as general methodology, data frames or text-processing techniques used for each task.

For the scripts to run accordingly, the three data set files "test-queries.tsv", "passage-collection.txt", and "candidate-passages-top1000.tsv", should be in the same folder as the scripts.

The first script task1.py also generates and exports in .jpg format the two figures included in this report.

Task 1 - Text Statistics

Deliverable 1

The passages collection text file has been imported into the Python code as a list of passages (strings), where each element corresponds to a row in the passages collection file. Each passage has then been pre-processed and tokenised in the following manner:

1. Convert all the characters of the passage to lower case to avoid mismatches of the same word due to capitalisation after punctuation or stylistic choices from the writer. This results in our tokenisation not accounting for terms whose difference in meaning is represented by capitalization, such as "Mark" (male name) and "mark" (spot, visible sign). However, these cases are less common, and they would not have much impact on subsequent tasks.

Term	the	a	of	and
Rank [k]	1	2	3	4
Freq. [f]	626,313	351,000	334,307	255,039

Table 1: *Top 4 most frequent words in the corpus vocabulary.*

2. Expand contractions such as "should've" into "should have", to end up with proper words after removing punctuation.

3. Remove punctuation and replace it by a white space. This step gets rid of all punctuation, which is unnecessary for tokenisation, but it also replaces it with a space, so that words that were joined by dashes, bars, or full stops, will be able to be tokenised separately. This is very useful when tokenising website addresses: "ruclip.com/videogame" into "ruclip", "com", and "videogame", or variants: "ar-15" into "ar" and "15", and therefore, if the query just has some part of the URL or the variant name, then the passages containing them will still be considered.

4. At this point, the only thing separating tokens are white spaces, so we tokenise by splitting the passages at the whitespaces. Every passage is converted to a list of individual tokens (strings).

5. Finally, we lemmatise the nouns returning them to their base (dictionary) form. This step groups words that represent the same thing but are written in different forms of the term, such as "cacti" and "cactus", or "cars" and "car". Only nouns are lemmatised because lemmatising other word types such as verbs or adjectives would be much slower to implement, and it would not have significant benefits regardless, since the most important terms in queries are usually the nouns.

The tokens extracted from every passage are put together in a list that contains a total of 10,658,796 tokens, including repetitions.

The passages vocabulary has been generated by counting the number of unique tokens and their

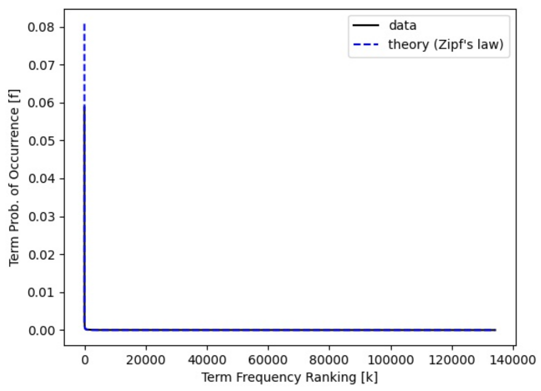

Figure 1: *Comparison of Zipf's law and corpus term frequency distribution.*

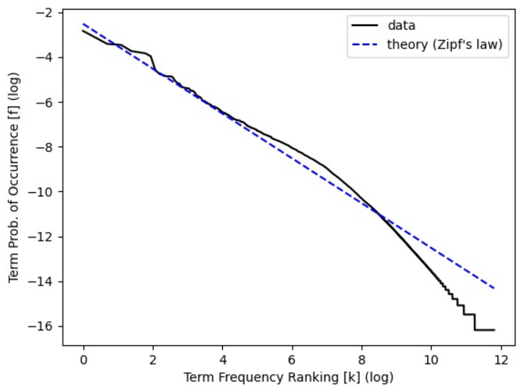

Figure 2: *Comparison of Zipf's law and corpus term frequency distribution in log-log space.*

respective occurrences. The vocabulary has N = 134,158 terms, and the four most frequent terms and their occurrences are shown in Table 1.

Zipf's law states that a term's (normalised) frequency is inversely proportional to its frequency rank:

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

Therefore, the most common term in the corpus should appear double the number of times the second most common term, and triple the number of times of the third, and so on. This is exemplified in Table 1, where the most common term "the" appears 1.8 times more than the second most common term "a". The general idea is that a small quantity of items appears very often in the corpus, whereas a large amount rarely appears. This is illustrated in Figure 1, where the data probability occurrence distribution has an asymptotic behaviour that matches the Zipf distribution. This same relationship can be better visualised in log-log space in Figure 2, where Zipf's law proposes that the relationship between normalised frequency and frequency ranking is constant (straight dashed blue line), as seen in eq. (1). This relationship is followed quite well by our data representation, but not entirely. The discrepancy is more obvious at the tails of the distribution, especially on the least frequent terms. Rare terms seem to be less likely than what Zipf's law predicts, which could be attributed to the fact that the text processing techniques followed did not manage to properly group tokens in extreme cases. Moreover, at the very end of the distribution a stair-like shape can be observed. This represents all the very rare terms that have the same number of occurrences but are given different frequency rank, because the algorithm gives a different rank to every unique term without any policy for ties.

Deliverable 2

See submitted script "task1.py".

Task 2 - Inverted Index

Deliverable 3

The inverted index we create is a dictionary that contains all the unique terms in the corpus as keys, and the value for each term is a sub-dictionary containing the *pid* of the passages that term appears in, as keys, and the number of occurrences of that term in passage *pid* , as the values.

First, we load the candidate passages as a data frame using the pandas module in Python. This data frame contains all the candidate passages for each query, each row containing the *qid* of the query, the *pid* of the candidate passage for that query, the actual query, and the actual passage. For this task we do not need the queries yet, we simply need the passages available in our dataset. Therefore, we create a new data frame that contains the unique passages found in the original data frame and their respective *pid*.

The text pre-processing and tokenisation approach we apply to the passages when creating the inverted index, is the same we followed in Task 1, with an extra step of removing stop words (i.e. common words that do not carry useful information, such as “the”, “is”, “and”) after tokenising but before lemmatising, since lemmatisation is the most computationally expensive step, so we apply it last when we have reduced the amount of tokens as much as possible.

For the creation of the inverted index, the most straight forward approach would be to import the vocabulary created in Task 1, loop over the terms in the vocabulary, and for every term, loop over all the passages and store the ones that contain the term with the respective number of occurrences. However, this would require running an outer loop of 134,158 iterations for the vocabulary, an inner loop of 182,469 iterations for the passages, and a further inner loop for every unique term in each passage, which would have a high computational cost. Instead, we generate the inverted index by directly looking at the unique words in the passages. If the words are not in the inverted index, we create a new key for them, and if they already are in the inverted index, we simply add the passage *pid* and the occurrence of the term in the passage in the term's sub-dictionary. Applying this approach to all the passages completes the inverted index with higher computational efficiency.

Note that the inverted index includes the number of occurrences each term has in each passage because this is needed to calculate similarity scores between queries and passages in subsequent Tasks 3 and 4. Having this information readily available in the inverted index significantly speeds up computation.

Deliverable 4

See submitted script "task2.py".

Task 3 - Retrieval Models

For this task we import the inverted index created in Task 2. The text pre-processing and tokenisation approach is the same as the one followed in Task 2.

We use the test queries and candidate passages datasets to create three data frames with the pandas module in Python. The "cp" data frame contains all the candidate passages for each query, each row containing the *qid* of the query, the *pid* of the candidate passage for that query, the actual query, and the actual passage. The "tq" data frame contains 200 queries with their respective *qid*. Finally, the "passages" data frame is created from the "cp" data frame, and it contains all the unique passages in our dataset with their corresponding *pid*.

When calculating scores between queries and passages, we do not check the similarity between a query and all the passages in the dataset, we just check the candidate passages for the respective query, as specified in the "cp" data frame.

Deliverable 5

The aim of this deliverable is to get the TF-IDF vector representations of the passages and queries, in order to calculate the cosine similarity scores between queries and passages, and retrieve at most 100 passages for each query, ranked from highest score to lowest.

$$TF - IDF_t = \frac{f_{t,D}}{|D|} \times \log \frac{N}{df_t} \tag{2}$$

Where $TF\text{-}IDF_t$ is the TF-IDF representation of term t in document D, D is the document term t is in (either a query or a passage), $f_{t,D}$ is the number of occurrences of term t in document D, $|D|$ is the size of document D with repetitions, N is the total number of passages in the corpus, and df_t is the number of passages in the corpus that contain term t.

For each passage, we store the TF-IDF value of each unique word it contains after text pre-processing and tokenisation. The TF value is computed using the occurrences stored in the inverted index and normalising with the length of the passage in question. The IDF value is specific to every term in the vocabulary, and it can also be obtained from the inverted index.

The TF-IDF representation of the queries is calculated in a very similar manner to the passages, where the TF value corresponds to the normalised frequency of the query's unique terms within the query, which is computed in place, since this information is not available in the inverted index. This does not represent any concerning computational cost given that there are only 200 queries. The IDF value is the same as for the passages.

Note that only the words in the queries present in the inverted index (i.e. present in the passages) are included in the vector representation of the queries, because they would not provide any additional information when retrieving passages and their IDF value cannot be calculated. However, their presence in the query is considered when computing the normalised frequency of the other words in the query that are in the vocabulary.

Cosine similarity scores are computed for each query and their respective candidate passages using the following equation:

$$similarity(q,p) = \frac{dot(q,p)}{norm(q) \times norm(p)} \tag{3}$$

Where *p* corresponds to the passage TF-IDF vector representation and *q* corresponds to the query TF-IDF vector representation. Note that the dot product in the numerator only considers the terms in common between the query and the passage, whereas the norms in the denominator take into account all the terms in the query and the passage.

File “tfidf.csv” contains the top 100 passages with the highest score in descending order for every one of the 200 queries, in order of appearance in the test queries data file. Note that some queries have less than 100 candidate passages, so all of them are considered “top passages” and are included in the .csv file.

Deliverable 6

BM25 is a ranking function used to estimate the relevance of passages (*p*) to a given search query (*q*). Its score is calculated applying the following summation equation to the common terms between a query and its candidate passage:

$$\begin{aligned} BM25(q,p) = \sum_{i=1}^{Q} \log \left(\frac{\frac{r_i+0.5}{R-r_i+0.5}}{\frac{n_i-r_i+0.5}{N-n_i-R+r_i+0.5}} \right) \\ \times \frac{(k_1+1)f_i}{K+f_i} \times \frac{(k_2+1)qf_i}{k_2+qf_i} \end{aligned} \tag{4}$$

$$K = k_1((1-b) + b \cdot \frac{dl}{avdl}) \tag{5}$$

Where *Q* is the number of unique terms in query *q*, n_i is the number of passages containing term *i* (it is term specific and can be retrieved from the inverted index), *N* is the total number of passages in the corpus, *dl* is the length of passage *p* (number of tokens with repetition), *avdl* is the average passage length of the entire corpus (constant for all passages), f_i is the number of occurrences the term *i* has in passage *p*, qf_i is the number of occurrences term *i* has in query *q*, k_1, k_2 and *b* are constant parameters set empirically, r_i is the number of relevant documents for query *q* containing the term *i*, and *R* is the total number of relevant documents for query *q*.

In this coursework we did not have any relevance feedback information, and therefore, the relevance feedback parameters r and R have been set to zero. This is a good approximation since the number of relevant documents will always be very small compared to the total number of documents in a corpus.

The BM25 score has been calculated for every candidate passage of every query. File “bm25.csv” contains the top 100 passages with the highest score in descending order for every one of the 200 queries, in order of appearance in the test queries data file.

Deliverable 7

See submitted script "task3.py".

Task 4 - Query Likelihood Language Models

This task uses the same inverted index, text pre-processing and tokenisation techniques, and data frames as Task 3.

Deliverable 8

Laplace Smoothing computes the similarity score between a query *q* and a passage *p* by calculating the joint probability the terms in the query have of appearing in the passage. The terms are assumed to be independent, so to obtain the total score we multiply the likelihood of the terms together. The formula for Laplace Smoothing is:

$$\begin{aligned} Laplace(q,p) &= \prod_{i=1}^{Q} P(w_i|p) \\ &= \prod_{i=1}^{Q} \frac{m_i+1}{|p|+|V|} \end{aligned} \tag{6}$$

Where *Q* is the number of unique terms in query *q*, $P(w_i|p)$ is the Laplace likelihood of term *i* being in passage *p*, m_i is the frequency of term *i* in passage *p* (taken from the inverted index), |*p*| is the length of passage *p*, and |*V*| is the total size of the corpus vocabulary (i.e. the number of terms in the inverted index).

Unlike the Unigram Query-Likelihood Model, which gives zero likelihood to terms in the query that are not present in the passage, and hence, gives zero score to all passages that do not include all the terms in the query, Laplace Smoothing gives every word some probability of occurring in every passage by discounting the probability estimates that are seen in the passage and assigning equally the left-over probability to the estimates for the words that are not seen in the passage.

The Laplace Smoothing scores are summarised in the file "laplace.csv", which ranks the top 100 passages for every query.

Deliverable 9

Laplace Smoothing gives too much weight to unseen terms. The Lidstone Correction re-calculates the Laplace Smoothing scores by adding a small number ε to every term count.

$$Lidstone(q,p) = \prod_{i=1}^{Q} \frac{m_i + \epsilon}{|p| + \epsilon|V|} \tag{7}$$

Note how Laplace Smoothing has the same equation as the Lidstone Correction when ε = 1.

The Lidstone Correction scores are summarised in the file "lidstone.csv", which ranks the top 100 passages for every query.

Deliverable 10

Discounting methods such as Laplace Smoothing and Lidstone Correction treat all unseen words equally by adding or subtracting ε. However, some words are more frequent than others, which interpolation methods such as Dirichlet Smoothing take into account by using background probabilities with estimates from the entire corpus. Interpolation methods calculate the estimates of terms in a query with the following expression:

$$P(w) = \lambda P(w|p) + (1-\lambda)P(w|C) \tag{8}$$

Where *w* is the term under consideration, λ is a parameter, *P(w|p)* is the probability of occurrence of term *w* in passage *p*, and *P(w|C)* is the probability of occurrence of term *w* in the entire corpus.

Dirichlet Smoothing takes eq. (8) and makes the smoothing depend on the length of the passages, so the final score between a query *q* and a passage *p* can be expressed as:

$$Dirichlet(q,p) = \prod_{i=1}^{Q} \left(\frac{|p|}{(|p|+\mu)} \frac{f_{i,p}}{|p|} + \frac{\mu}{(|p|+\mu)} \frac{f_{i,C}}{|C|} \right) \tag{9}$$

Where *Q* is the number of unique terms in query *q*, |*p*| is the length of passage *p* (number of terms with repetitions), μ is a constant, $f_{i,p}$ is the frequency of term *i* in passage *p* (taken from the inverted index), $f_{i,C}$ is the frequency of term *i* in the entire corpus *C* (also taken from the inverted index), and |*C*| is the total number of terms in the corpus (with repetitions).

The Dirichlet Smoothing scores are summarised in the file "dirichlet.csv", which ranks the top 100 passages for every query.

Deliverable 11

Dirichlet smoothing is expected to be the best language model since it gives all the terms in the query a probability based on their frequency in the passage and their frequency in the entire corpus. In doing so, it does not assign equal probability to all the terms of the query that are not present in the passage, which is what laplace Smoothing and Lidstone Correction do.

As stated in Deliverable 9, Laplace smoothing and Lidstone correction are very similar with the single change that Laplace uses ε = 1 for discounting and Lidstone uses ε = 0.1. The value of ε represents how much importance we give to terms in the query that are not present in the passage. The larger the value of ε, the more importance we give to those terms. In our analysis, Laplace Smoothing assumed that the terms in the query that were not found in the passage were more likely to be in the passage, than the Lidstone Correction did.

The parameter μ represents the weight we give to the likelihood of the term in the entire corpus when calculating the similarity scores. Setting μ to a very large number would mean that we base the Dirichlet similarity scores between a query an a passage based mostly on the probability of the words in the query being in the entire corpus, which does not make sense, since it would ignore the specifics of the passage, and ultimately we are trying to estimate the similarity between the query and the particular passage. A good estimate for μ is usually the average document length. Given that our average document length is 35 terms, 50 is a more appropriate value for μ than 5000.

Deliverable 12

See submitted script "task4.py".