Information Retrieval and Data Mining (COMP0084) Coursework 1

Abstract

In this project, **Task 1** uses 'passage- collection.txt' to justify Zipf's law. **Task 2** uses 'candidate-passages-top1000.tsv' to construct an inverted index so that a single word can be used to collect all the passages containing it and the number of occurrences of this word. **Task 3** extracts TF-IDF for passages and queries using the inverted index in task 2 and sorts the similarity between passages and queries by cosine similarity. Then BM25, another method, is used to sort the similarity between passages and queries. **Task 4** implements the query likelihood language models respectively with Laplace smoothing, Lidstone correction and Dirichlet smoothing. It aims to retrieve passages for each query by calculating the natural logarithm of the probability.

1 Introduction

1.1 Aim

Information retrieval models extracted practical knowledge by searching through data. This project will develop information retrieval models to retrieve passages and achieve a re-ranking system based on specific indices. Some relevant libraries are used in this coursework, i.e. NumPy, matplotlib, re, pandas, nltk and collections.

1.2 Data

This project uses three datasets: 'test-queries.tsv', 'candidate-passages-top1000.tsv' and 'passage-collection.txt'. 'test-queries.tsv' has 200 rows with a format of <qid[1] query[2]> containing the identifiers of the query and the query texts. 'candidate-passages-top1000.tsv' with $189,877$ rows includes an initial selection of at most $1,000$ passages for each of the queries in 'test-queries.tsv' with a format of <qid pid[3] query passage[4]>.

2 Text processing

Text processing helps simplify the input text and remove useless information. It reduces the number of features and gets a more generalized model, improving the retrieval model's efficiency and accuracy.

2.1 Normalisation

First, all the letters in the input file are converted to lowercase letters when reading the file by lines. Simultaneously, long blanks at the start and end of each line are deleted. Then all non-English letters are removed, such as numbers, greek letters, punctuation, and hyphens.

2.2 Tokenisation

The input text is cut into tokens using the 'Regexp-Tokenizer' function.

2.3 Stop word removal

In this step, the extremely common and meanless words in English text are removed, such as pronoun, preposition, and article words. Stop words will not be removed for Task 1.

2.4 Lemmatisation and stemming

To extract word roots of vocabulary, the lemmatisation step helps get the base form of a word and the stemming algorithm used helps convert the derived word to a word stem. 'SnowballStemmer' algorithm is used in this project. This step will not be used for Task 1.

3 Tasks

The code for these tasks are stored in four py files, and there are five csv files storing task outcomes.

3.1 Task 1 – Text statistics

Use the data in 'passage-collection.txt' for this task. Extract terms (1-grams) from the raw text. In doing so, you can also perform basic text preprocessing steps. However, you can also choose not to. In any case, you should not remove stop words for Task 1, but you can do so in later tasks of the coursework.

D1: Describe and justify your preprocessing choice(s), if any, and report the size of the identified index of terms (vocabulary). Then, implement a function that counts the number of occurrences of terms in the provided data set, plot their probability of occurrence (normalised frequency) against their frequency ranking, and qualitatively justify

[1] the identifier of the query
[2] the query text
[3] the identifier of the passage
[4] the passage text

that these terms follow Zipf's law[5]. Some preprocessing steps mentioned in the previous section are performed on the text in 'passage-collection.txt'. 'passage-collection.txt' is read bylines, and letters in each line are converted to lowercase letters. At the same time, the long blank at the start and end of each passage will be removed if it exists. Then, all the non-English words are deleted. These steps will not remove any English words and will not affect the vocabulary identified. After tokenisation, this file is split into $10,061,726$ tokens. As required, stop word removal will not be performed in Task 1. Lemmatisation and stemming steps will not be used for this task since this task focuses on the repeated occurrences of terms, not on the meaning of the terms. In Brown Corpus, the terms with high frequencies are mostly pronoun, preposition, and article words, which usually have little information. Without lemmatisation and stemming, Zipf's law can still be verified.

The function 'word_occurrence' in 'task1.py' returns the vocabulary list and the number of occurrences of each term in the list. After counting, 'passage-collection.txt' after preprocessing contains $174,911$ identified terms. Top 8 terms are shown in Table 1. Figure 1 shows the probability of occurrence against their frequency ranking.

Term	Times	Term	Times
`the`	622860	`to`	239509
`of`	333460	`is`	216621
`a`	281312	`in`	199205
`and`	253567	`for`	107612

Table 1: Top 8 vocabulary(Term) and the number of occurrences of terms(Times) in 'passage-collection'.

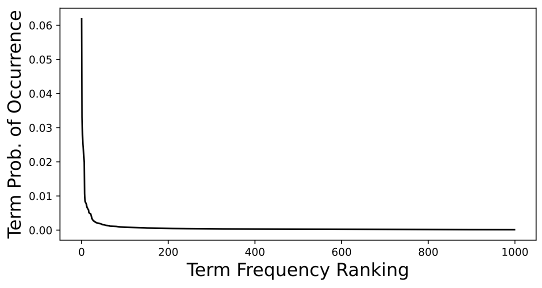

Figure 1: The probability of occurrence against the frequency ranking of terms.

Zipf's law shows that the product of its frequency ranking and frequency is constant for any term, given a large corpus. Figure 1 roughly shows an inverse proportion for top ranking 1000 terms.

Zipf's law for text sets $s = 1$ in the Zipfian distribution is defined by

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

Hence, $f(k; N) = \frac{1}{k \sum_{i=1}^{N} i^{-1}} = \frac{C}{k}$. Where k is the term's frequency rank, N is the number of terms, $C = \frac{1}{\sum_{i=1}^{N} i^{-1}}$. To show Zipf's law more intuitively, a log-log tansformation did on this equation and then $log(freq.) = log(C) - log(k)$. The plot after tansformation is showed in Figure 2.

The log-log plot gives a straight line with the fitted equation $log(freq.) = -1.29 - 0.86 * log(k)$, following Zipf's law. The slope is -0.86, slightly larger than -1 (the slope should be). The terms ranking from 3 to 7 deviate from the fitted straight line. The difference comes from the number of occurrences counted in 'passage-collection.txt'. The equation of Zipf's law suggests the frequency of the term ranking1 should be i times larger than the frequency of the term ranking i. Table 1 shows that the top 8 vocabularies do not strictly follow this rule, which might be the reason for the difference.

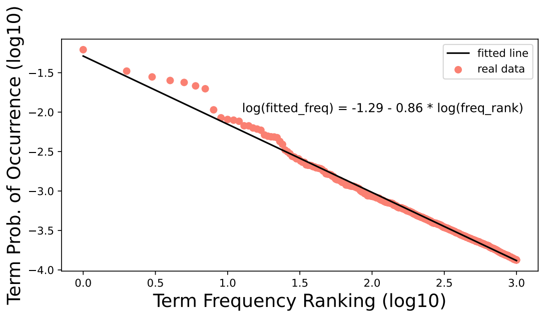

Figure 2: The log of probability of occurrence against the log of frequency ranking of terms.

3.2 Task 2 – Inverted index

Use 'candidate-passages-top1000.tsv' for this task (unique instances of column pairs pid and passage). Using the vocabulary of terms identified in Task 1 (you might also consider removing stop words), build an inverted index for the collection.

D3: Provide a description of your approach to generating an inverted index, report the information that you decided to store in it, and justify this

[5]Coursework Task1

choice[6].

Ans: First, the file 'passage-collection.txt' will be reprocessed in this task. In addition to the preprocessing methods used in Task 1, stop word removal, lemmatisation and stemming will also be used in this task, helping remove some meanless words and reduce the amount of vocabulary. After pre-processing, 'passage-collection.txt' has $144,060$ words in the vocabulary list. Then, the 'passage' column in 'candidate-passages-top1000.tsv' will be processed following methods used for 'passage-collection.txt' in this task. Finally, all pids and corresponding processed passages will be stored in a dictionary for convenient data extraction.
The construction method of generating an inverted index is as follows:

1. for each pid, find its coresponding passage;
2. for each token in the passage, count the number of its occurrence in this passage;
3. loop over all pids, store each token, all pids that the token appears and the number of occurrence of the token in each passage in a dictionary named inverted_index.

The inverted index can be used to quickly search for the pid that a term appears at and the number of times in a passage. For example, if the input token is 'ment', then the command **inverted_index['ment']** gives all the positions of 'ment' (pid) and the count of 'ment' in each passage. The total count of 'ment' in all passages could also be calculated by the command **sum(inverted_index['ment'].values())**.

3.3 Task 3 – Retrieval models

D5: Extract the TF-IDF vector representations of the passages using the inverted index you have constructed. Using the passages' IDF representation, extract the TF-IDF representation of the queries. Use a basic vector space model with TF-IDF and cosine similarity to retrieve at most 100 passages from within the 1000 candidate passages for each query. Store the outcomes in a file named tfidf.csv. Each row of 'tfidf.csv' must have the following format: <qid,pid,score>[7].

Ans: By definition, $TF_{t,d}$ is the frequency of occurence of a term t in the document d, i.e. the total count of term t in the document d divided by total number of terms in the document d. Inverse document frequency IDF_t is a measure of term specificity, i.e. $log_{10}(\frac{N}{n_t})$, where N is the total number of documents and n_t is the number of documents containing term t. Then, TF-IDF is the product of $TF_{t,d}$ and IDF_t.

Using the inverted index got in task 2, TF-IDF vector representations of passages can be constructed by following steps:

1. find the total number N of passages in the file 'candidate-passages-top1000.tsv' ($N = 182,469$);
2. for each pid, find its corresponding passage;
3. for each token in the passage, find TF, n and IDF using the inverted index, then get TF-IDF;
4. loop over all tokens in the passage, store TF-IDF and IDF for each token;
5. loop over all pids and store TF-IDF and IDF in two dictionaries, respectively TF_IDF_dict and IDF_dict.

Table 2 shows the TF-IDF vector representation for the passage with pid 7130104.

Pid: 7130104

Token	**TF-IDF**
`definit`	0.24126928464459813
`rna`	0.8878854198334536
`along`	0.2356678319559567
`exampl`	0.18115151845248195
`type`	0.1508411568396757
`molecul`	0.2670448678178688

Table 2: TF-IDF vector representation for the passage with pid 7130104.

The TF-IDF representation of the queries can be constructed by following steps:

1. reprocess the queries in the file 'test-queries.tsv' using the same methods in task 2;
2. construct a function 'tf_query' finding TF for queries:
   (a) for each qid, find its corresponding query;

[6]Coursework Task2
[7]Coursework Task3

   (b) for each token in the query, find TF, i.e. count of the token in the query divided by the total number of tokens in the query;
   (c) loop over all qids, store qid, token, and TF in a dictionary named query_TF_dict.

3. for each qid, find all corresponding pids;

4. for each pid, find common tokens in its passage and the query. Then calculate TF-IDF for those common tokens using query_TF_dict and IDF_dict, since IDF for queries is the same as the passages' IDF;

5. loop over all pids, store pid, tokens, TF-IDF in a dictionary.

After finding the TF-IDF vectors for passages and queries, cosine similarity is introduced to measure the similarity between the passage and query. If there are two vectors X and Y, the cosine similarity is calulated by

$$\frac{\sum_{i=1}^{V} x_i \times y_i}{\sqrt{\sum_{i=1}^{V} x_i^2} \times \sqrt{\sum_{i=1}^{V} y_i^2}}.$$

Using the formular above, the cosine similarity between queries and passages can be evaluated using TF-IDF vectors for queries and passages. The scores ranking top 100 are stored in a file named tfidf.csv.

D6: Use the inverted index to also implement BM25 while setting $k_1 = 1.2$, $k_2 = 100$, and $b = 0.75$. Retrieve at most 100 passages from within the 1000 candidate passages for each test query. Store the outcomes in a file named bm25.csv[8].

Ans: BM25 is another classic algorithm used to calculate query-to-document similarity scores in information retrieval. The formula for BM25 is as follows

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

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

k_1, k_2 and b are parameters seting empirically. dl is the length of each passage. $avdl$ is average passage length, which is about 32.2764 in this task. f_i and qf_i are respectivaly the number of occurrences of a token in passage and query. N is the total number of passages and n_i is the total number of passages which the token appears in.

Since there is no relevance information given, R and r_i are zero. With given parameters $k_1 = 1.2$, $k_2 = 100$ and $b = 0.75$, the score is calculated by the following steps:

1. for each qid, find its corresponding query and passages

2. find scores for all corresponding passages using the formula above

The scores ranking top 100 are stored in a file named bm25.csv.

3.4 Task 4 – Query likelihood language models

D8-10: Use 'test-queries.tsv' and 'candidate-passages-top1000.tsv' for this task. Implement the query likelihood language model with (a) Laplace smoothing, (b) Lidstone correction with $\varepsilon = 0.1$, and (c) Dirichlet smoothing with $\mu = 50$, and retrieve 100 passages from within the 1000 candidate passages for each test query. Store the respective outcomes in the files 'laplace.csv', 'lidstone.csv', and 'dirichlet.csv'. In these files, the column score should report the natural logarithm of the probability scores[9].

Ans: Query likelihood model is a standard language modelling approach. The probability of generating the query is given by

$$P(Q|M_D) = P(q_1 \ldots q_k|M_D) = \prod_{i=1}^{k} P(q_i|M_D)$$

$$score = log(P(Q|M_D)) = \sum_{i=1}^{k} log(P(q_i|M_D))$$

Where Q is the query, q_i is the ith token in the query, M_D is the language model for document D in the collection.

The frequency distribution of words in a document is counted to estimate the unigram language model corresponding to the document. However, the probability of words not appearing in the document is set to zero, which is not reasonable. To prevent zero estimates of the probability of unseen

[8]Coursework Task3

[9]Coursework Task4

words, the model needs to be smoothed to make it closer to the true probability distribution. In this task, Laplace smoothing, Lidstone correction and Dirichlet smoothing are introduced to estimating probabilities for missing words:

1. Dirichlet smoothing:

   $$p(q_i|D) = \frac{f_{q_i,D} + \mu p(q_i|C)}{|D| + \mu}$$

   $f_{q_i,D}$ is the number of occurrences for q_i in document D, $|D|$ is the length of document D and μ is a constant.

2. Laplace smoothing: Laplace smoothing assumes that the distribution over words is uniform, equivalent to adding 1 for each kind of word in the document D. Hence, it is also called add-one smoothing.

   $$p(q_i|D) = \frac{tf_{q_i,D} + 1}{|D| + |V|}$$

   $tf_{q_i,D}$ is the number of times q_i occurs in the document D and $|V|$ is the size of vacabulary list.

3. Lidstone correction: Laplace smoothing gives too much weight to unseen terms, hence a small number ε is used instead of 1.

   $$p(q_i|D) = \frac{tf_{q_i,D} + \varepsilon}{|D| + \varepsilon|V|}$$

D11: Which language model do you expect to work better? Which ones are expected to be more similar and why? Comment on the value of $\varepsilon = 0.1$ in the Lidstone correction – is this a good choice, and why? If we set $\mu = 5000$ in Dirichlet smoothing, would this be a more appropriate value and why?

Ans: Dirichlet smoothing assumes that a multinomial distribution can represent a document. It adds μ new words to each document D, where μ is obtained by sampling from $p(q_i|C)$. $p(q_i|C)$ is the estimated probabilities of the token q_i obtained from the corpus. This method of estimating probabilities seems straightforward, but it makes more sense than Laplace smoothing and Lidstone correction as it takes the distribution in corpus into consideration.

Laplace smoothing is a particular case of Dirichlet smoothing[1], and Lidstone correction is adapted from Laplace smoothing. No matter adding 1 or ε, the nature of Laplace smoothing and Lidstone correction is simply adding a positive number. They are purely for not having zero probabilities and are unable to solve the problem of making valid predictions for unseen instances.

A study of smoothing methods mentioned that for short queries, Dirichlet works better than absolute discounting and Jelinek-Mercer. For long queries, Jelinek-Mercer works better than Dirichlet and absolute discounting[1]. Combining with the queries in this project, I expect Dirichlet works better than Laplace and Lidstone.

$\varepsilon = 0.1$ for Lidstone correction is a suitable choice. ε refers to the degree of certainty that adding ε is an accurate estimate of its probability. Increasing the Lidstone smoothing parameter compensates more for absent features. If choosing $\varepsilon = 1$, it is too confident about the assumption of uniform priors. Hence, $\varepsilon = 0.1$ carefully consider about the uncertainty about the uniform distributions.

$\mu = 5000$ is not a good choice. The smoothing parameter for Dirichlet is considered to be closely related to the document length as it would expect that a longer document requires less smoothing than a short one[1][3][4]. The average document length is often used as the parameter value in Dirichlet smoothing[2][3]. Without having any trails or prior information, I consider $\mu = 50$ is a suitable parameter since the average length of passages is around 32.2764. Jangwon Seo and W. Bruce Croft chose maximize mean average precision (MAP) and the average document length as the Dirichlet smoothing parameters. However, the experiments showed poor performance for the model with the average document length[2]. Hence, it is better to train and tune the models to get the optimal parameter for smoothing.

REFERENCES

[1] Zhai, C. & Lafferty, J. D. (2001), A Study of Smoothing Methods for Language Models Applied to Ad Hoc Information Retrieval, in 'Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2001' , pp. 334–342.

[2] Seo, J. & Croft, W. B. (2010), Unsupervised estimation of dirichlet smoothing parameters., in Fabio Crestani; Stéphane Marchand-Maillet; Hsin-Hsi Chen; Efthimis N. Efthimiadis & Jacques Savoy, ed., 'SIGIR' , ACM, pp. 759-760.

[3] Smucker, M. D. , Kulp, D. , & Allan, J. . (2005). Dirichlet Mixtures for Query Estimation in Information Retrieval.

[4] Fang, H. & Zhai, C. (2005), An exploration of axiomatic approaches to information retrieval., in Ricardo A. Baeza-Yates; Nivio Ziviani; Gary Marchionini; Alistair Moffat & John Tait, ed., 'SIGIR' , ACM, pp. 480-487 .