COMP0084 : Information Retrieval and Data Mining - Coursework 1

Anonymous ACL submission

Abstract

This document contains answers to multiple tasks given as an individual coursework for IRDM. It contains detailed description to the questions and ideas behind certain approaches.

1 Introduction

Without so much data everywhere, Information retrieval systems are very common nowadays. Efforts are being made to develop more accurate retrieval systems that give relevant results when a query is given. In this report, I'll mention how I approached the problem of making a information retrieval system that gives ranked results when a query is given. For this, I've used cosine similarity for vector space models and BM25 (probabilistic model). At last, I've made query-likelihood model using laplace smoothing, Lindstone correction, dirichlet smoothing and compared them.

2 Tasks

2.1 Text Statistics

For pre-processing, I've made a function to remove punctuation, tokenized passages, convert them to lower case and removed any token that is a number. For this coursework, we will be making embedding for word similarity, so having punctuation won't make much sense, that is why it has been removed. Words like "RNA" and "rna" are same, but a computer would treat them different and we'll have two embeddings for it instead of one. To avoid such scenario, I'm converting all words to lowercase. Then, after tokenization, any token that is a number makes no sense in uni-gram, it's just noise. It would make sense if there was context (eg. in bi-gram, tri-gram etc.). For this task, stop words are not removed. After pre-processing, all the tokens are total words. The unique words in this will for the vocabulary. The length of vocabulary, thus formed, is 117124. Then, a function is created that gives

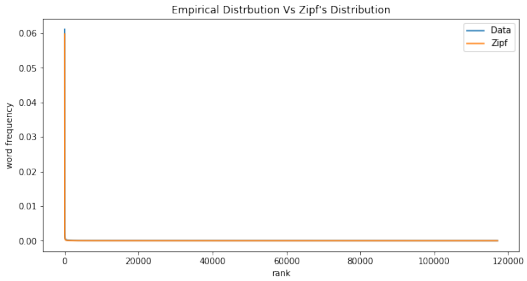

Figure 1: Probability of occurrence (normalised frequency) of a term vs the term's ranking

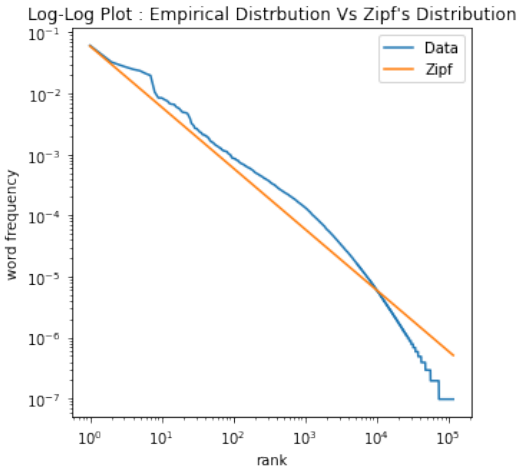

Figure 2: log of the Probability of occurrence (normalised frequency) of a term vs log of term's ranking

frequency distribution of words in the vocabulary. These results are then ranked in descending order. Zipf's law suggests that the rank of a term times its frequency (k * f) is constant. Zipf's distribution and empirical distribution are then compared.

As it can be seen in Figure 1, it seems like empirical distribution does follow the zipf's law. But, it's really large data and thus small differences might not be visible. Thus, it would be more beneficial to see a log-log plot of this.

In figure 2, it can be seen that Zipf's law is constant while empirical distribution deviates a bit.

This can be explained by the fact that in real world, ranks of words are unknown, they are calculated from extracted frequencies. Thus, the assumption, the nth most frequent word is the nth most likely word, might not be true. Thus, we see that empirical distribution is deviated.

2.2 Inverted Index

Inverted Index is really powerful concept. In this task, I tried to make a dictionary. The key of the dictionary is the word. The value of a key is a list of tuples. One tuple contains pid, frequency of key in pid and total tokens in pid. Using loops to go through each and every passage to calculate frequencies for every tuple made the code run for too long. Because having three loops takes a long time to compute. So, I used a function in nltk called FreqDist which gives frequency distribution of words that have been given as an argument. This made the code significantly fast as I just needed one loop now instead of three. So, now I loop over the unique passages, get passage id, tokens in the passage and frequency distribution of these tokens. Then, I loop over this frequency distribution and add the information in the appropriate key. I chose this particular indexing because in the next task for tf-idf, I can directly access it from here and would thus save space and also time in calculating these. Stop words have been removed from this task onwards. This is because they don't add much information but have high frequencies. By removing them, we give more importance to other words which are actually useful. The below dictionary shows how the data will be represented. Here 'definition' and 'rna' are words in vocabulary. 7130104 is passage id that has 16 tokens in total and out of those 3 are 'definition' and 5 are 'rna'.

```
{'definition': [(7130104, 3, 16),
  (8002085, 1, 40),
  (7133141, 1, 25),
  (8004107, 5, 31),
  (7135097, 3, 20),
  (8407773, 1, 17),....]

  'rna': [(7130104, 5, 16),
  (8466138, 1, 31),
  (8466146, 1, 18),
  (5259281, 1, 29),
  (5260519, 1, 25),
  (7297881, 4, 26),...]}
```

2.3 Retrieval models

Retrieval models specifies a computational process. Eg. how the documents are ranked.

2.3.1 Vector Space Model

Vector space models means that the data would be represented as vectors and distance/angle between them would tells us how similar they are. The basic idea in this task is to generate tf-idf vectors for passages and queries. Then, we have to find cosine similarity between passages and queries. Based on this cosine similarity, we need to rank the retrieved results and return top 100 results for each query. When trying to store the tf-idf of passages in list/arrays, it was not possible due to MemoryError. Thus, I made functions for creating passage vectors and query tf-idf vectors. So, instead of saving everything, I'll just be passing it to the function and get the results. When making passage vector of length equal to size of vocab, many words won't come. Similarly, in query vector might contain new words (not in dictionary). So, to deal with them try-except is used as it improves the speed of code as compared to if-else. Next, I added cosine similarity code which takes passage and query vectors and returns cosine similarity. These similarities are again saved in dictionaries. This is because first dictionary is able to save such a large data and secondly, lookups in dictionaries are of O(1) while for list it is O(n). Thus, my cosine similarity dictionary will have qid as key and values would be a list which contains cosine similarities of that qid with different pid. It looks like this:

```
{1108939: [0.04271754862417325,
  0.07997984843621488,
  0.07775703541299413,
  0.08526632487905707,...]
  ...}
```

Now, once I have all values, I need to sort them (rank them) and get top 100 for each query. For this, I make a small function that gets the index of sorted array in ascending order and then extract last 100 elements. This is how this looks:

```
{1108939: array([ 30, 382, 526,
155, 855, 519,...])...}
```

tf idf dict is then made to store the values of these 100 indices as we need to report them. At last, I write qid,pid and scores to tfidf.csv which gives the desired output.

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

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

Figure 3: Formula for BM25

$$\text{Laplace estimates:} \left(\frac{m_1+1}{|D|+|V|}, \frac{m_2+1}{|D|+|V|}, \ldots, \frac{m_{|V|}+1}{|D|+|V|} \right)$$

Figure 4: Laplace smoothing formula

2.3.2 BM25: A probabilistic model

BM25 is a probabilistic model which is based on binary independence model. We're are already given the hyper parameter values. I calculated the dl (document length) and avdl (average document length). Then, I reused the functions I made above to calculate BM25 scores. The only difference was in formula. So, instead of cosine similarity score, here we have BM25 probability scores. As above, I used the simialr method to rank the results and make a BM25.csv file.

2.4 Query likelihood language models

A language model is a probability distribution over strings of text. Query likelihood language model is constructed for each document within our collection. We then rank documents by the probability that the query could be generated by that document model. When we say this, it assumes that all query words must be in documents. But, actually it's not like that. So, we do smoothing. In this, we estimate probabilities for missing words. This is done by lowering the probabilities for seen words and assigning left over probability to unseen words, so all words have some probability.

2.4.1 Laplace Smoothing

In Laplace smoothing, it is similar to having uniform priors over words. It's basically like count events in observed data, add one to each and renormalize. The same thing is repeated in the code. Additionally, we have taken log of the probabilities thus obtained. After this the same method of getting ranks and extracting top 100 documents for each query is repeated.

2.4.2 Lidstone Correction

The Laplace smoothing gives too much weight to unknown words and thus Lidstone correction was introduced.It adds a small number epsilon to every count and then re-normalizes it. Epsilon used is 0.1 which was already given in question.

$$P(w|D) = \frac{tf_{w,D} + \varepsilon}{|D| + \varepsilon |V|}$$

Figure 5: Lidtone correction formula

$$\lambda\, P(w|D) + (1-\lambda)\, P(w|C)$$

Figure 6: Dirichlet Smoothing formula

2.4.3 Dirichlet Smoothing

The problems with above methods are that they just give probabilities to unseen words equally. It might be possible that some words are more likely than other. Thus, it would be better to smooth with some background probabilities. This is what Dirichlet Smoothing does. We need to extract probability of word in document and probability of word in collection. These are then multiplied by constants that depend on sample size. Once, I did make the function for this, similar steps are repeated to get the top 100 documents for each query along with log of probabilities.

2.4.4 Comparison of Query Likelihood Models

I expect Dirichlet smoothing to work better in general as it takes sample size into consideration when giving probabilities to unknown words rather than treating them equally. In lidstone, epsilon basically adds a small number to every word count and thus if a word is not present it is still there at least epsilon times. It is a hyper-parameter. Practically, values less than 1 are good choice. Smaller value means more close to MLE. Mu is an estimate of average document length in Dirichlet Smoothing. If we increase value of mu, we are decreasing P(w|D) and P(w|C).I guess that would be bad as it would be underestimating the probabilities. The actual average document length is around 32, so 50 would be a better estimate than 5000.

2.5 References

UCL lectures Slides