Search Engine

Saaket Agashe

May 2022

1 Software

The software consists of three sub-modules. All software is writen in Python and standard sub-modules like Re(Regex), Collections(Dictionaries), JSON and Numpy have been used. The software can be found at:
https://github.com/saa1605/search_engine
For using the code, all data needs to be put in a separate data folder.

- Data Parsing (Query + Corpus)
- Creating the inverted index
- Scoring and Ranking
- Creating the log file

The Parsing, Creation of inverted index and the log-file generation is independent of the algorithm used and can be modularly used with any new retrieval algorithm. The specifics of the custom algorithm and how the data parsing might differ is covered in the section Custom Algorithm.

1.1 Data Parsing

1.1.1 Corpus Parsing

Separating Documents: The OHSUMED corpus is a single body of text, where each document is separated using a .I header and has a unique .U Document ID. The first step of parsing the entire corpus consists of separating the single text body into an array of individual documents. I separately collect the document IDs given under the .U header after splitting the corpus.

Cleaning Entries: Each document, before being passed to the inverted index creation module, is first cleaned using a regular expressions filter. This process removes the headers (Of the form .X), removes punctuation, any extra tabs, white-spaces and newlines, removes any digits and finally clears any leading or trailing white spaces. This cleaning process gives a list of tokens for each document. Stopwords are then removed from this list of tokens using a predefined list of stopwords.

1.1.2 Query Parsing

The query has a different format from the corpus and uses headers in the form of ⟨ HEADER ⟩. I first extract the query IDs for each query and then remove all the angular bracket based headers. I remove reserved words like Description and Title from the query. The query is then sent to the same cleaning pipeline used to clean the corpus. Finally, I remove stopwords from the query.

1.2 Creating the inverted index

The inverted index is a reverse mapping from individual terms in the corpus to the document that they appear in. For this purpose, I use a dictionary in Python. They keys of the index are the tokenized terms in the document and their values include a relative ID (Unique for each document and different from the document ID extracted from the .U header) as well as the frequency of the term in that document. The process of creating the inverted index take about 1 and a half minute in Python.

While generating the inverted index, I also create a separate array which stores the Lenghts of the individual documents, to be used while generating relevance scores for the documents.

1.3 Scoring and Ranking

I tried four different ranking algorithms and then implemented my own which is a modification of the relevance feedback algorithm.

All algorithms follow a pipeline where a score is computed using the term frequencies and other auxiliary information from the inverted index for each query. Then the Top k (50) documents are retrieved. A log string is generated for these returned documents which is then written to a log file. Finally the trec_eval script is run with this log file.

1.3.1 Boolean Retrieval

The scoring process for boolean retrieval involves assigning a 0 or 1 score for each document. The score is calculated by adding (1 / length(query)) value to for each term that is present in the document. If the final value for the score of a document is greater than t ($0 < t < 1$) it is made 1, else 0. Thus the boolean retrieved documents get a final score of either 1 or 0 with a tolerance of t.

1.3.2 Term Frequency

The term frequencies are stored in the inverted index along with the document in which the term appears. During the scoring process, a euclidian normalized term frequency is used to assign the final scores. The scoring algorithm is given by

$$w_{(t,q)} = tf\_query(t)$$

$$scores[d] = \sum tf_{(t,d)} * norm * w_{(t,q)}$$

Where $tf\_query(t)$ is the term frequency of the term t in the query, norm is the normalization factor, and $tf_{(t,d)}$ is the term frequency The scores are divided by the respective document lengths for normalization.

1.3.3 TF-IDF

The TF-IDF scoring uses a similar method used for the Term Frequency scoring. The difference lies in the following term:

$$w_{(t,q)} = tf\_query(t) * (idf(t)^2)$$

Here $idf(t)$ is the Inverse Document Frequency of the term t which is computed beforehand. The frequency is square to consider it's multiplication in both the query vectorization and document vectorization.

1.3.4 Relevance Feedback

I have implemented a pseudo-relevance feedback algorithm. The algorithm first retrieves the 5 most relevant documents using a TF-IDF ranking algorithm. The processed tokens from these documents are added into the tokens of the original query. This expanded query is then passed to a TF-IDF based ranking algorithm and the documents are retrieved.

1.3.5 * Custom Algorithm

The custom algorithms makes use of two facts about the intermediate document retrieved during pseudo relevance feedback:

1. The tokens of the intermediate documents retrieved by the pseudo-relevance feedback are less important than the tokens of the original query.
2. The lower the retrieved documents are ranked, the less important their tokens are to the information need.

Thus, I use a discounted, weighted pseudo-relevance feedback. The scoring process uses two loops. In the first loop, only the original query is used to score the documents. The scores obtained from this loop are multiplied by a factor α

$$w_{(t,q)} = (tf\_query[t]) * (idf[t]^2)$$

$$scores[d] = \sum tf_{(t,d)} * norm * w_{(t,q)} * \alpha$$

The second loop goes over the intermediately retrieved documents in a ascending order of their ranks. The scores obtained from the most highly ranked document is multiplied by a factor $(1 - \alpha)$, the second ranked document by $(1 - \alpha)^2$ and so on.

$$w_{(t,q)} = (tf\_retrieved\_document[t]) * (idf[t]^2)$$

Method	Mean Average Precision	R Precision
Pseudo Relevance Feedback	0.044	0.1011
Pseudo Relevance Feedback with Discounted Weighting (alpha = 0.7)	0.047	0.1023
Pseudo Relevance Feedback with Discounted Weighting (alpha = 0.85)	0.049	0.1100

Table 1: Comparison of Pseudo Relevance Feedback with and without discounted weighting

Method	Running Time
Pseudo Relevance Feedback	3min 35seconds
Pseudo Relevance Feedback with Discounted Weighting	3min 40seconds
TF-IDF	2min 30seconds

Table 2: Comparison of retrieval run time of the different algorithms

$$scores[d] = \sum_{k} \sum tf_{(t,d)} * norm * w_{(t,q)} * (1-\alpha)^{k}$$

2 Experimental Results

Table 1 shows the Mean Average Precision and R Precision for 50 documents obtained by Pseudo Relevance Feedback, with and without discounted weighting. In this case Discounted weighting improves performance over normal Pseudo Relevance Feedback. It also shows how varying alpha changes the results. Alpha decides how much weight is to be given to the main query versus the retrieved documents. It can be seen that keeping alpha high is beneficial as the original query is significantly more important than the documents retrieved during feedback. However, adding these documents does improve performance over plain pseudo relevance feedback.

Table 2 describes the run-time comparisons for the three approaches. It is clear that pseudo relevance feedback requires more time to process due to the added process of intermediate retrieval and processing. However, the Discounted Weighting process does not significantly increase the running time of the Pseudo Relevance Feedback method. The creation of the inverted index takes **1 Minute 41 Seconds**.

During the experiments, I observed that normalizing the term frequencies actually decreases precision on TF-IDF ranking. However, using a common normalizing factor, which is a median of the individual normalizations for all the documents helped the overall performance.

3 Learnings

This assignment helped me understand the inner workings of the ranking algorithms covered in class. It helped me understand the complexity of parser which can significantly improve performance based on how the document is cleaned and stop words processed.

I was able to dive deeper into the concept of Relevance feedback and was able to come up with a small change that improved performance over ordinary relevance feedback.