Contents

1 wikiQA: Search Engine

A basic search engine using jacard/cosine similarity and tf-idf score for Wikipedia Q/A. [Naman Gupta, 2013064]

2 References

1. Porter stemmer from https://tartarus.org/~martin/PorterStemmer/
2. http://www.cs.cmu.edu/~ark/QA-data/

3 Q1

Assume a collection of document where each document is also tagged with a location. Design an indexing mechanism which will allow Text only queries (this is traditional inverted index) and Text + Spatial queries? Each location is denoted by (x, y). For extra credit, argue if the scheme will work if each document is tagged with multiple locations. If not then extend it to support multiple locations per document.

The inverted index will be the similar to the ones we have studied so far in the class. But additionally the multiple locations with each `document_name` will be created. The original inverted index will have 2 changes. It will now contain a pointer to the doc_name(the second index created will be accessed by `doc_id` foreign key) instead and a new score distance-tf-idf-score (more on this later) will also be stored with each term.

```
term: {
        doc_name(doc_id)*, //doc_id is foreign key
        frequency,
        distance-tf-idf-score,
        tf-idf-score,
        final_weight = distance-tf-idf-score * tf-idf-score,
        positions[], //positions in each doc
}

doc_name*: {
```

```
        doc_id, //primary key
        locations[] //array of locations (suporting), each location is (x, y)
}
```

1. tf(term_frequency) the number of occurences of term t in the document d) `tf = frequency of t in d/ number of terms in d` and

2. `idf (inverse of document frequence)`idf = log10(number of documents in collection / document frequency (how many terms contain the term t))‘

Now, the tf-idf-score will be calculated as usual, where TF-IDF-SCORE = tf*idf and stored for each term and corresponding doc it is present in. The query q contains text + spatial information (location (x,y) from where the query is fired from). Also, the document d contains the word present in the query. There are m locations associated with the document d, namely (x_1, y_1),(x_2, y_2),(x_3, y_3), ….. (x_m-1, y_m-1), (x_m, y_m). Now, the second score, which is distance-tf-idf-score is calculated as follows:

```
distance_1 = distance of x_1,y_1, from x,y = sqrt( (x_1 - x)^2 +  (y_1 - y)^2)
distance_2 = distance of x_2,y_2, from x,y = sqrt( (x_2 - x)^2 +  (y_2 - y)^2)
distance_3 = distance of x_3,y_3, from x,y = sqrt( (x_3 - x)^2 +  (y_3 - y)^2)

.
.
.

distance_m = distance of x_m,y_m, from x,y = sqrt( (x_m - x)^2 +  (y_m - y)^2)

normalized average for all the documents X,
given by X = sqrt(distance1^2 + distance2^2 ..... + distancem^2)

distance-tf-idf-score = 1 / log10(X)

and finally, final_weight (d,t) = distance-tf-idf-score*tf*idf)
```

The distance-tf-idf-score is the RMS (root mean square of the m distances from the location x,y where the query is fired from). Now, the results are ranked using final weight, which is given by distance-tf-idf-score score multiplied by tf-idf-score (`final_weight` (d,t) = distance-tf-idf-score * tf * idf)). This scoring scheme ensures that the content/documents which are geographically closer to the user (who fires the query) are ranked higher than the users residing far away from the document locations. The distances are normalized and averaged because there are multiple locations associated with each document - this ensures that the document containing locations USA, Africa, Spain, India - still stays locally relevant/irrelevant on an average to the user which is nearest to all of these countries (when all of them are taken together). Note, that reciprocal of log is taken to ensure that the documents nearer to the query maker (user) have a higher weightage in the calculation of the final_weight.

4 Q2

4.1 Usage

python search_engine_1.py

python search_engine_2.py

python search_engine_1.py inverted_index.json stop_words.txt ./Documents/ file_sentences.json = part 1

python search_engine_2.py inverted_index.json stop_words.txt ./Documents/ file_sentences.json = part 2

4.2 Features Implemented

1. For part 1: Uses cosine and jacard similarity to fetch questions for the answers in answer corpus.
2. For part 2: It uses tf-idf to retrieve ranked results of possible documents which may contain the answer to the question. A faster method/alternative was also used which splits senteces (with the help of nltk) and matches the query with every sentence in every doc. The second approach is much more relevant to the the type of Search engine we are designing. EXAMPLE: We have a query, "Who did James Monroe marry?", Now, the tf-idf will output all the docs which contain these terms, the terms may be in a single sentence, or far apart. The only demerit of tf-idf scoring is that it doesn't consider how close the words are in the query. However, the sentence similarity scoring is much relevant in this case, because we get a finer list of results which contains these words closely.
3. For part 1 and 2: Precision/Recall is designed for binary relevance (relevant or not relevant). Therefore, NDCG is used to evaluate the search engine system where rank of the results is also considered. Since there is no way to determine the Ideal DCG (we don't know the ideal relevance of docs/answers), the ideal ranked answers/docs are assumed to be randomly shuffled and then ndcg is computed. Also, mean average precision, mean average recall and mean average ndcg is also calculated to evaluate the system (with n queries). Note that, precision, recall and NDCG for every results for every query is stored as a JSON which is used to compute the Mean Average precision, recall and NDCG after the end of results of each query.
4. For Part 3: Because we know the document (Wikipedia article), the question and the answer, you can therefore verify if for the same question the output of part 1 is same as output of part 2. Please see the section 4.4 in this doc below. You can run these queries to match the answers form the part 1 matches the documents and answers returned from part 1.
5. Cutoffs used for Jaccard and Cosine similarity is 0.4 +- 0.0001.
6. Stores the inverted index (both inverted word index, and inverted sentence index) as a json allowing offline caching (saves precious time 7 minutes required to build inverted index, power and those CPU cycles on building the index again) (see `build_index()`, `load_index_in_memory()` and `write_to_file()`). The The inverted index json stores as follows

   ```
   term: {
       doc_name,
       frequency,
       score,
       positions[]
   }
   ```

   While, the sentence index json stores the following

   ```
   [
       doc_name: {
   ```

```
            sentences[]
        }
    ]
```

7. For part 2: Logic for tf-idf ranked matching `MultiWordQ()` function: Search for the each query term in the inverted index. Take a union of results. Make a set. The output is the list of documents that contain any of the query terms. Sorted according to tf-idf scores/relevance. Prints the recall, precision and ndcg scores.
8. The query can be entered iteratively just like a normal shell. A prompt is visible where the query is entered and results are shown almost instantaneosly.

4.3 Sample Queries for part 1

1. What are the similarities between beetles and grasshoppers?
2. What did Alessandro Volta invent in 1800?
3. What are the similarities between beetles and grasshoppers?

4.4 Sample Queries for part 2

1. Which county was Lincoln born in?
2. What important electrical unit was named in honor of Volta?
3. When did Lincoln begin his political career?
4. Who did James Monroe marry?