Lillian Zha (lz2527) and Emily Hao (esh2160)
Advanced Database Systems
25 February 2020
Project 1

Files Submitted

- IR_model.java
- json-20140107.jar
- proj1-stop.txt
- README.pdf
- project1 script

API Key:

AIzaSyCsrIFbLRYxueXNFRLPCkHd9cwdtKYiYrk

Google Engine ID:

017054685595593044797:5wdco3mni7w

Program Instructions

1. Navigate to proj1 folder.
2. Allow permissions on the project1 script (chmod 754 project1)
3. Run with ./project <apikey> <engineID>

(if the script isn't working, compile and run with the following)

```
### To Compile

`javac -classpath json-20140107.jar:. IR_model.java `

### To Run

`java -classpath json-20140107.jar:. IR_model <apiKey> <googleEngineID`
```

Internal Structure & Query Modification Method

We ask the users for the query and target precision score first. Then we get the search results from google in json format and parse it for the title, url and snippet. We also read in the file of stop words as a set of strings. We then tokenize (alphanumeric) the snippet and filter out stopwords if the word wasn't in the original query. Next, we generate hashmaps (term_frequencies, count_matrix) to keep track of the term frequencies and document frequencies for each unique term from the snippets and each of the 10 documents. Then we run the loop that asks users for the relevance of each of the 10 documents and compute a precision score out of 10. If the score is less than the specified score, then we alter the query.

Our query modification method followed the approach described by the referenced paper below. First, after retrieving the search results from Google, we create vectors of size |V|, where V is the number of words in the vocabulary, out of the query and the documents. The entries in the query vector are weights associated with each word in the vocabulary, defined by the following formula:

$$w_{ik} = \frac{(\log(f_{ik}) + 1.0) \cdot \log(N/n_k)}{\sqrt{\sum_{k=1}^{t} [(\log(f_{ik}) + 1.0) \cdot \log(N/n_k)]^2}},$$

where f_{ik} is the number of times word T_k occurs in query Q_i, N is 10 (the number of results returned by Google at each iteration), and n_k is the number of documents T_k appears in. This weighting system follows the popular tf-idf weighting system. The entries in the document vectors are calculated similarly, with the same formula removing the $log(N/n_k)$ terms in the numerator and denominator.

After getting user relevance feedback, we use the set of relevant documents and irrelevant documents to calculate a new query vector with Rocchio's algorithm. The new query vector moves the query's representation towards where the relevant documents vectors are in the representation space and away from where the irrelevant ones are.

$$Q_{\text{new}} = A \cdot Q_{\text{old}} + B \cdot \text{average\_wt\_in\_rel\_docs} - C \cdot \text{average\_wt\_nonrel\_docs}.$$

To decide which two words to add to the query, we find the words that are not in the previous query with the two highest weights and append them in descending order according to their weights in the query vector.

Additional Project Implementation Details

- We only analyze snippets, not the entire html webpage. So we also have not encountered trouble with non-html files.
- We tried stemming (OpenNLP), however the results were not better so we removed that.
- When considering word order in the expanded query, we tried ordering them by their query vector weights (as defined in the paper below) in descending order, but saw no improvement in precision. We decided to, in each iteration, keep the previous query's words and add new words in descending order.
- The only external library we used is the org.json library to parse the results of the google search.

References

Automatic routing and retrieval using smart: TREC-2

Transcripts

[brin]

[per se]

[jaguar]