ADB-Project-1

COMS 6111 Advanced Database Systems Project 1: Relevance Feedback

By:

Xuejun Wang, UNI: xw2355

Akshaan Kakar, UNI: ak3808

Essential Information

List of Files in Submission

```
dyn-160-39-205-135:ADB-Project-1 AmyWang$ tree
.
├── LICENSE
├── MainScript.py
├── README.md
├── ScoringSystem.py
├── ScoringSystem.pyc
├── TermParamsClass.py
├── TermParamsClass.pyc
├── VectorSpaceClass.py
├── VectorSpaceClass.pyc
├── key.json
├── resources
|   └── english
└── transcript.txt
```

Additional Remarks:

1. Bing Search API accountKey is in file `key.json` and it is in the submission directory
2. The transcript contains our results for 3 required test cases, and additional test case 'columbia' for Columbia University, 'milky way' for the candy bar, both at required precision@10 0.9

Compile/ Run Instructions

1. `key.json` is the file that stores Bing Search API accountKey. Make sure it is under the same file path as `MainScript.py`. The file is in following format:

   ```
   {"accountKey": "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"}
   ```

2. From terminal, change file path to the project root folder
3. Run the project using following command format:

   ```
   python MainScript.py -q 'your query' -p 0.9
   ```

4. When asked for relevance feedback, input n/N/y/Y are all going to be accepted, any other characters will not be recognised and you will be prompted to input until the project receives acceptable character

Internal Design

The three main classes used and their descriptions are as follows:

1. **TermParams** : This class is responsible for holding all the relevant information for a term. This includes the inverse document frequency, as well as a disctionary, whose keys are the documents containing the term and values are the corresponding positions list of the terms in the doc. The structure of the class is shown in the example below.

   ```
   class TermParams:
       self.idf = 0.1
       self.doc_pos = {"docA" : [1,2,4,22,55] , "docB" : [1,2,3,5,8]}
   ```

2. **VectorSpace** : This class holds the vector space representation of the current query and the entire corpus of documents (relevant and non-relevant). The class holds an inverted file and a relevance list. The inverted file holds all the terms in the vocabularies with their corresponding TermParams objects. An example is shown below:

```
#Inverted File
{
"term1":
    {
    self.idf = 0.1
    self.doc_pos = {"docA" : [1,2,4,22,55] , "docB" : [1,2,3,5,8]}
    },
"term2":
    {
    self.idf = 0.2
    self.doc_pos = {"docP" : [1,3,5,7], "docQ" : [2,4,6,8]
    },
"term3": ...
"term4": ...
.
.
.
}
```

The relevance list is a simple dictionary, with documents as keys and their user idnicated relevance as the corresponding values.

```
#Relevance List
{
    "docA" : 'y' , "docB" : 'n', ..... , "docP" : 'y', "docQ" : 'y'
}
```

The class also contains a method to produce the weight vectors for the documents and query using the term frequency inverse document frequency (tf-idf) scoring system.

1. **ScoringSystem** : This class is reponsible for scoring the terms based on the relevance feedback from the user, and expanding the query by picking the highest scoring terms. The class contains helper methods to add, subtract and multiply sclars to lists. The main scoring method utilizes the Rocchio Algorithm as described in the next section, to reweight the terms based on the feedback. This class takes in an object of class VectorSpace, so as to obtain the document and query tf-idf weight vectors. This class also take in three additional parameters α, β and γ which are parameters used in the Rocchio Algorithm.

Query Modification Mechanism

In order to incoroporate the relevance feedback and expand the query with every successive iteration, we used the **Rocchio Algorithm** (Manning et al. 2009). This algorithm produces a new query term weight vector from the original query vector and the term weight vectors for the relevant and non-relevant documents according to the following equation:

$$Q_{new} = \alpha . Q_{orig} + \beta . \frac{1}{|D_{rel}|} \sum_{D_j \in D_{rel}} D_j - \gamma . \frac{1}{|D_{nrel}|} \sum_{D_k \in D_{nrel}} D_k$$

Here, we subtract the normalized weights of the non-relevant documents and add the normalized weights of the relevant documents to the original query weight vectors, therefore moving the query towards the desired set of terms. The three parameters α, β and γ are free parameters and can be tuned as desired. For out implementation, we found the following values to be ideal:

```
α = 1
β = 0.8
γ = 0.3
```

Once the new query vector is calculated, we then pick the new highest scoring terms that are not present in the original query. These terms are then appended to the query to produce the final expanded query.

One last decision to make before augment original query with new highest scoring tesms is to decide if the top two terms should be added or one. We decided to compare the weighted scores of the top two terms, if their absolute difference is within a small threshold (`<0.05`), they are both considered highly relevant and will be augmented into new query for next round. Otherwise, only the top one term will be augmented. In our test case for 'columbia', this mechanism can make sure both 'new' and 'york' are added for next round, instead of appending the single word 'new' that does not make sense for information retrieval, and would not overshoot by appending two words for every round in other cases.

References

1. Stop-word corpus was borrowed as-is from the NLTK toolkit.
   Bird, Steven, Edward Loper, and Ewan Klein. "Natural Language Processing with Python."
   Http://www.nltk.org/nltk*data/. O'Reilly Media, n.d. Web. <http://www.nltk.org/nltk*data/>.

2. Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze: An Introduction to Information Retrieval, page 181. Cambridge University Press, 2009