March 17, 2019

Manas Gaur, Amanuel Alambo

Information Retrieval, CS 7800

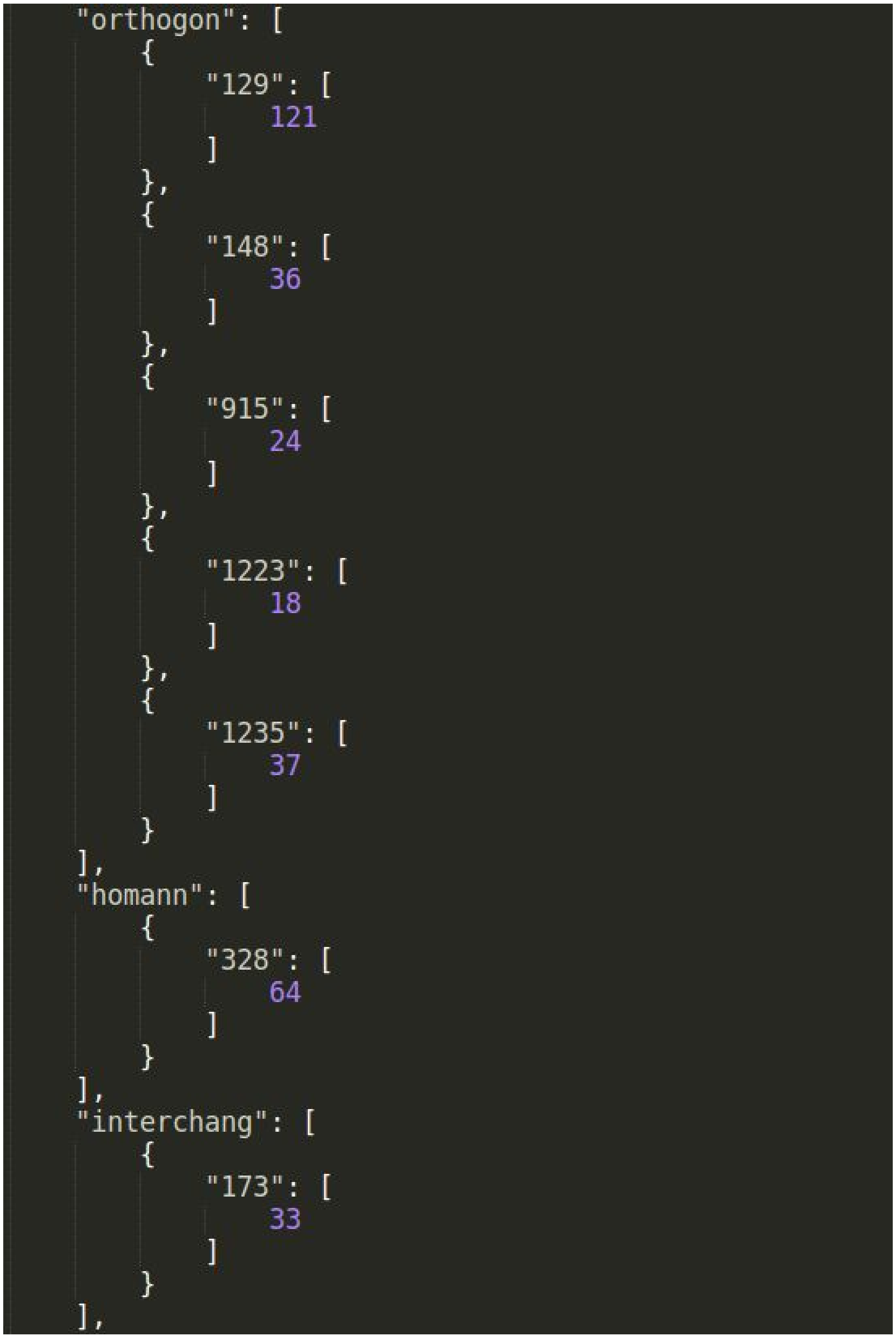
Project-1

A Simple Search Engine

This search engine is designed and implemented to conduct indexing and query processing on the Cranfield Dataset. The programs are designed and organized in such a way that indexing of the corpus is performed by the *index.py*.
Abbreviation if any, used in the documentation :
"doc" := document
"id" := identifier
"NLTK" := Natural Language Tool-Kit

Indexing

1. The Cranfield corpus (***cran.all***) is read by ***index.py*** using a special program (***cran.py***). *cran.py* is used for proper parsing of the ***cran.all*** file.
2. Once the corpus (***cran.all***) is read and parsed, function ***indexDoc*** of class ***InvertedIndex*** is invoked. A doc's title and body are extracted in *indexDoc* function as we are interested indexing the title and body of a doc from the corpus. We used program *doc.py* to deal with parsing a document object.
3. Program ***util.py*'s splitDoc** function is used for preprocessing of the document. The preprocessing of a doc entails tokenization (using NLTK), punctuations marks removal, stopwords removal, stemming and lowercasing. In this implementation, we used ***SnowballStemmer***. The preprocessed doc gets returned to the indexing program.
4. Once all the terms of a doc are preprocessed and retrieved, each of the terms is stored in a data structure of the form; **dictionary of dictionaries,** with a term as the key for the outer dictionary and the document id as the key for the inner dictionary and the list of term positions of the term in the document as the value of the inner dictionary. The complete dictionary is sorted using docid and term positions as well (See Figure 1).
5. Finally, the all terms and their docIDs and the term positions which are stored in dictionary structure *items* are saved to an output index file. Writing of the sorted inverted index structure (i.e. *items*) to an output file is handled through json serialization.
6. To perform indexing of a corpus (collection), execute the following:
python index.py "path to collection" "path to index_file"
Example : python index.py ../Cranfield/cran.all ../indexfile/index_file.json
7. We tested our program by loading the inverted index structure from a file and printing a sample term's docid and corresponding term positions. Sample test conditions are included in function *test* in program ***index.py***.
8. Sample snippet of the index file:

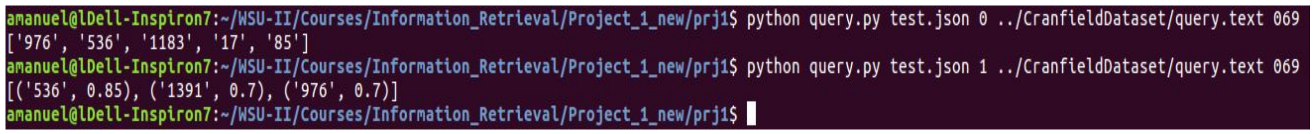


```
    "orthogon": [
        {
            "129": [
                121
            ]
        },
        {
            "148": [
                36
            ]
        },
        {
            "915": [
                24
            ]
        },
        {
            "1223": [
                18
            ]
        },
        {
            "1235": [
                37
            ]
        }
    ],
    "homann": [
        {
            "328": [
                64
            ]
        }
    ],
    "interchang": [
        {
            "173": [
                33
            ]
        }
    ],
```

Figure 1: Snapshot of the Index

Query Processing

1. Program ***query.py*** is the main program for performing query processing. There have been defined two ways for query processing in program ***query.py***, boolean model and vector model. Just as done for indexing, a query is preprocessed using the functions defined in program ***util.py***. In addition to the functions defined in ***util.py***, **Norvig's spelling correction module** is used in query processing prior to preprocessing (stemming, stopword removal, lowercasing). A ***query id*** is used for selecting which query to pass to ***query.py*** from **query.text** in our implementation.
2. A boolean query implementation (function ***booleanQuery*** in class ***QueryProcessor***) takes in preprocessed query terms and returns list of document ids from an index file where all query terms co-appear. Thus, the boolean query is implemented as a conjunctive query across the index file where only document identifiers having mentions of all the query terms are returned.
3. Similar to the boolean model, a vector model query implementation (function ***vectorQuery*** in class ***QueryProcessor***) takes in preprocessed query terms. This function essentially represents each query term in a document using its TF-IDF value forming a matrix of query terms and document ids with the value in each cell corresponding to a TF-IDFvalue. The implementation of TF-IDF weighting in function *vectorQuery* uses a combination of term frequencies(TF) computed within the same function and idf computed in another function(*IDF*). Cosine similarity is used to compute the similarity between query term weights(normalized frequency of a term in a query) and TF-IDF scores for each document. Thus, cosine similarity is performed iteratively between a vector of query term weights and each document's TF-IDF scores. Finally, the **top-k** documents based on the cosine similarity scores are returned by the function (i.e., a ranked list of tuples of the form (doc ID, cosine similarity score)).
4. To execute query processing, execute the following: (see Figure 2)
python query.py "path to index file" processing_algorithm "path to query.text" query_id
Example: 0 for Boolean and 1 for Vector processing algorithm
Python query.py ../indexfile/index_file.json 0 ../queryfile/query.text 049
5. We tested our program by making separate calls to the boolean and vector models (which is specified while passing arguments in running the program) and seeing the returned list of docids for the boolean model and the list of top-k ranked pairs of docid-cosine similarity scores. The testing is done in function **query()** in program ***query.py***. We verified our results by manually computing the similarity scores.
6. Sample snippet of running the query processing script:



```
amanuel@lDell-Inspiron7:~/WSU-II/Courses/Information_Retrieval/Project_1_new/prj1$ python query.py test.json 0 ../CranfieldDataset/query.text 069
['976', '536', '1183', '17', '85']
amanuel@lDell-Inspiron7:~/WSU-II/Courses/Information_Retrieval/Project_1_new/prj1$ python query.py test.json 1 ../CranfieldDataset/query.text 069
[('536', 0.85), ('1391', 0.7), ('976', 0.7)]
amanuel@lDell-Inspiron7:~/WSU-II/Courses/Information_Retrieval/Project_1_new/prj1$
```

Figure 2: Snapshot for generating all relevant (boolean model) and top-k docIDs with respect to query id 069

Query Evaluation

1. For evaluation of query results by the **QueryProcessor** engine, we used the Normalized Discounted Cumulative Gain (NDCG). We defined function ***to_ndcg()*** in program ***query.py*** to compute aggregate ndcg scores. Program ***batch_eval.py*** is used to pass arguments which will be consumed by function ***to_ndcg()*** in program ***query.py***.
2. For easier manipulation, we represented the ***qrels.text*** file as a pandas dataframe in our ***to_ndcg()*** function. The ***qrels.text*** is organized into a dataframe in such a way that its columns have the following layout: **[‘query id’, ‘document id (docID)’, ‘boolean relevance (bool_rel)’, ‘vector relevance (vec_rel)’]**.
3. Our function (***to_ndcg()***) picks *n* random query ids from qrels, performs an ndcg score for one query id and places the ndcg score corresponding to the query id into a list structure.
4. To safely and consistently handle query id written in **nearly octal form** (for example, query id 010 when casted to integer becomes 8 and not 10) in file ***query.text*** and written in decimal form in file ***qrels.text***, we implemented a code snippet in our function (***to_ndcg()***) to cast a decimal query id into its octal equivalent. This casting is done for each randomly selected query id.
5. For representing the boolean query results corresponding to a query id, we start by initializing to zero-tuple a vector ***bool_list*** of size equaling parameter *k* passed to function *to_ndcg()*. We iterate through the list of all document ids returned by function ***boolQuery()*** and test if a document id is present in *qrels.txt*. If the docid is present in *qrels.txt*, we set the value of the corresponding tuple of *bool_list* to (1,1); otherwise, we would set (0,1). Once *bool_list* is populated with the list of all tuples, we extract the first element of each tuple and assign to a vector ***y_test*** and the second element to vector ***y_score***. Finally, *y_true* and *y_score* are passed to function *ndcg_score()*.
6. For representation of the vector query results, we initialize a vector ***y_score*** to the cosine similarity scores returned by function *vectorQuery()*. Since function *vectorQuery()* returns the same *k* number of (docid, similarity) tuples, we merely grabbed all the similarity scores. Similarly, we initialize a vector ***y_true*** equal to parameter *k* passed to the function. We grabbed the *docids* from the results returned by function *vectorQuery()* and tested the presence of each of these docids in *qrels.txt* for the specific query id. Since the length of the all-zeros initialized *y_true* and the length of the list containing the docids returned by the vector model are the same, we set *y_true* entries to ‘1’ whenever the corresponding index’s entry in the docids list is present in *qrels.txt*.
7. Once the ***bool_rel*** **and** ***vec_rel*** columns of the dataframe for a single randomly chosen query id are populated, we call function ***ndcg_score()*** defined in program *metrics.py*. Computation of the ndcg_score is handled by the functions defined in program ***metrics.py***.
8. Finally, we used the **wilcoxon signed rank test** to get the **p-value** between the boolean and vector model for randomly selected query ids.
9. For evaluation, execute the following:

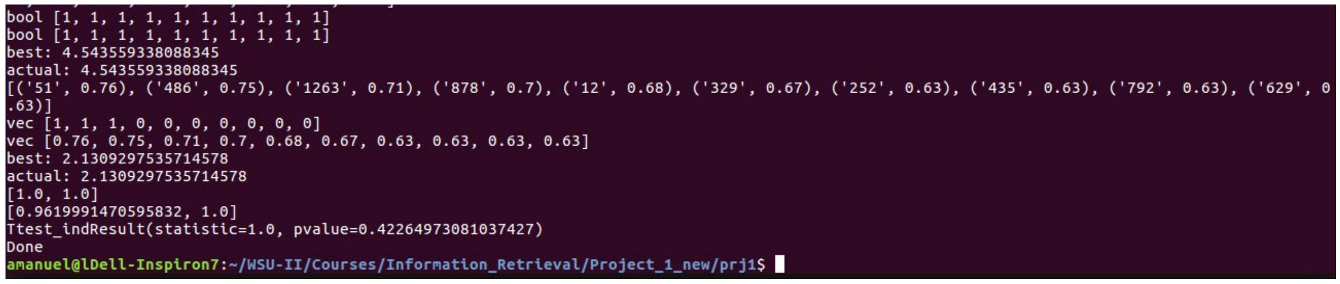
python batch_eval.py “path to index file” “path to query.text” “path to qrels.text” “number_of_randomly_selected_queries”
Example:

python batch_eval.py ../indexfile/index_file.json ../queryfile/query.text ../queryfile/qrels.txt 6

10. We tested our program by running the batch_eval.py file as in above and what we found was there were query ids present in file *qrels.text* but not in *query.text*. As a result, all boolean and vector relevance entries in the dataframe were all zeros making the ndcg scoring function return 'nan'. We use *ttest* for p-value in our program

11. Furthermore, due to spell checking implementation, it is possible that there be some words present in a query but not in the index file.

12. Sample snippet of the *batch_eval* program execution:



```
bool [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
bool [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
best: 4.543559338088345
actual: 4.543559338088345
[('51', 0.76), ('486', 0.75), ('1263', 0.71), ('878', 0.7), ('12', 0.68), ('329', 0.67), ('252', 0.63), ('435', 0.63), ('792', 0.63), ('629', 0
.63)]
vec [1, 1, 1, 0, 0, 0, 0, 0, 0, 0]
vec [0.76, 0.75, 0.71, 0.7, 0.68, 0.67, 0.63, 0.63, 0.63, 0.63]
best: 2.1309297535714578
actual: 2.1309297535714578
[1.0, 1.0]
[0.9619991470595832, 1.0]
Ttest_indResult(statistic=1.0, pvalue=0.42264973081037427)
Done
amanuel@lDell-Inspiron7:~/WSU-II/Courses/Information_Retrieval/Project_1_new/prj1$
```

Figure 3: Snapshot of batch_eval.py showing docIDs from booleanQuery and vectorQuery function, there presence in qrels.txt and generating best and actual arrays for calculation of ndcg_scores