CSI 4107 Assignment 1

Student Name	Student Number
Arild Yonkeu	300123590
David Lumbu	300144520

Task Distribution

To implement the IR system, Arild was the one in charge for step 1 and for writing the README file. David was the one in charge of step 3 and running the system on the 50 queries. Both of us were in charge of step 2.

Functionality of the programs

Our IR system is written in Python 3.

First, we have the `preprocessing.py` file. It takes as input our collection of documents and our list of stopwords to give us our vocabulary. Next, we have the `inverted_index.py` which will take our generated vocabulary and make our inverted index in JSON format. Finally we have our `retrieval_and_ranking` which will take our generated inverted index and calculate the cosine similarity between the documents.

How to run the programs

To be able to run the program you must run these following commands:

```
pip install nltk
python preprocessing.py
python inverted_index.py
python retrieval_and_ranking.py
```

Discussion

Preprocessing

To preprocess our collection of documents, we created four functions, `read_file_and_tokenize()`, `stem()`, `load_stop_words()`, and `remove_stop_words()`. Throughout our Python file, we have tried to use sets in many places as possible, as they are faster than using lists. The `read_file_and_tokenize()` function will extract content within a specified file and return a list of tokens. The `stem()` function will use the PorterStemmer to stem a list of tokens as input and return a list of stemmed tokens. The `load_stop_words()` function will load our stopwords in the `stop_words.txt` into a set. Finally, our `remove_stop_words()` will take our loaded stop words and tokens as input and return a set of tokens without stop words. the `load_stop_words()` is executed only once while the rest of the functions are executed for each file in our collection, and therefore we generate our vocabulary. Our vocabulary has a total of 122340 words.

Indexing

With our generated vocabulary, we build our inverted index with a dictionary. It uses the tokens in the vocabulary as the keys and the values are dictionaries with the document number that the token appears in and the number of times it occurs as the value. We do this using the `build_inverted_index()` function. This function will output our index into a JSON file format.

Example of the structure of the index

```
...
    "zwinger": {
        "AP880815-0232": 3
    },
    "zwingl": {
        "AP880702-0153": 2,
        "AP881015-0068": 1
    },
...
```

The words `zwinger` and `zwingle` are tokens acting as keys. Their values contain dictionaries with the document number of the document that these tokens appear in, and the number of times it occurs.

Retrieval and Ranking

With our generated index we compute the cosine similarity between the chosen query and the documents and we rank the document by revelance, using the `retrieve_and_rank_queries()` function, along with a few helper functions to provide our function with the necessary data. The `get_idf_values()` function will calculate all the idf values of each token in the inverted index and return a dictionary of idf values where the keys are tokens and the values are the idf values associated to each key. The `create_doc_vectors()` will create the necessary document vectors, which are dictionaries with tokens as the keys and the normalized tf values as the values. The `calculate_docs_tf_idf_values()` will calculate the tf-idf values of all documents and return a dictionary of tf-idf values where each key is a document number and each value is a document vector, which is dictionary where each key is a token and each value is a tf-idf value. The `load_queries()` function will load our queries in `queries.txt` into a dictionary where each key is the query num and each value is a dictionary that contains the title and the description of the query. The function also has a `titles_only` flag if the loaded queries do not need the description loaded into a specified dictionary. The `calculate_queries_tf_idf_values()` function will normalize the tf values of our queries and calculate the tf-idf values of all of them and return a dictionary where each key is a query number and each value is query vector, which is a dictionary where each key is a token and each value is a tf-idf value. The `cos_sim()` will return the cosine similarity between a document vector and a query vector. The `retrieve_and_rank_queries()` will take our document tf-idf values and our query tf-idf values and and it will use the `cos_sim()` function calculate the similarity scores and then rank our documents for each query in the descending order of similarity scores and return them into a dictionary where each key is a query number and each value is a dictionary where each key is a document number and each value is the similarity score associated with the query and the document. Finally, the `save_results()` function will take our results and save them into our chosen output file name.

First 10 answers to queries 1 and 25

```
1 Q0 AP881206-0124 1 0.4610 name_execution
1 Q0 AP881005-0001 2 0.3589 name_execution
1 Q0 AP880815-0061 3 0.3281 name_execution
1 Q0 AP880825-0054 4 0.3219 name_execution
1 Q0 AP880814-0089 5 0.3050 name_execution
1 Q0 AP881021-0218 6 0.3040 name_execution
1 Q0 AP881002-0014 7 0.3003 name_execution
1 Q0 AP881225-0044 8 0.2839 name_execution
1 Q0 AP880726-0173 9 0.2744 name_execution
1 Q0 AP881223-0053 10 0.2634 name_execution
```

```
25 Q0 AP880917-0094 1 0.6134 name_execution
25 Q0 AP880606-0019 2 0.5994 name_execution
25 Q0 AP880605-0026 3 0.5745 name_execution
25 Q0 AP880811-0163 4 0.5394 name_execution
25 Q0 AP880812-0017 5 0.5334 name_execution
25 Q0 AP881011-0091 6 0.5249 name_execution
25 Q0 AP880427-0240 7 0.5163 name_execution
25 Q0 AP880427-0150 8 0.5057 name_execution
25 Q0 AP881016-0013 9 0.4737 name_execution
25 Q0 AP880916-0009 10 0.4518 name_execution
```

The results are from running the queries with the titles and descriptions. Using only the titles gave poorer results. We can see that our results are not closely related to the ideal results.

TREC eval Results

```
runid                   all     name_execution
num_q                   all     50
num_ret                 all     50000
num_rel                 all     2099
num_rel_ret             all     1589
map                     all     0.2361
gm_map                  all     0.1253
Rprec                   all     0.2647
bpref                   all     0.2885
recip_rank              all     0.5345
iprec_at_recall_0.00    all     0.5884
iprec_at_recall_0.10    all     0.4421
iprec_at_recall_0.20    all     0.3781
iprec_at_recall_0.30    all     0.3256
iprec_at_recall_0.40    all     0.2853
iprec_at_recall_0.50    all     0.2458
iprec_at_recall_0.60    all     0.2063
iprec_at_recall_0.70    all     0.1546
iprec_at_recall_0.80    all     0.0959
iprec_at_recall_0.90    all     0.0591
iprec_at_recall_1.00    all     0.0287
P_5                     all     0.3680
P_10                    all     0.3320
P_15                    all     0.3093
P_20                    all     0.2940
P_30                    all     0.2653
P_100                   all     0.1564
P_200                   all     0.1003
P_500                   all     0.0538
P_1000                  all     0.0318
```

Our Mean Average Precision (MAP) score is 23.61%. This could be due to many things, including spelling mistakes within the corpus. This leaves room for improvement. One way to improve our system is to include the pseudo relevance feedback loop when retrieving and ranking documents. Another way is to use the BM25 formula for weighting.

Appendix: Vocabulary Sample

aa
aaa
aaaaaaaawk
aaaaaawk
aaaaargh
aaaah
aaaall
aaah
aaaron
aabb
aabi
aabl
aabout
aabpara
aabx
aaccord
aachen
aacn
aadministr
aadvantag
aaf
aah
aai
aaichiy
aaii
aainst
aajkal
aalborg
aalesund
aali
aalto
aaltonen
aam
aamal
aand
aanytim
aap
aaqbiyeh
aar
aarafat
aardema
aardvark
aarhu
aarn
aaro
aaron
aaronson
aarp

aart
aarvi
aarvik
aasa
aassoci
aata
aavoid
aawc
aazpa
ab
aba
ababa
aback
abaco
abacu
abacus
abad
abadab
abadan
abadi
abadia
abadilla
abadlah
abagail
abahani
abajo
abakr
abakua
abakumov
abalkin
aballa
abalon
abancay
abandah
abandon
abandond
abang
abaray
abasan
abasc
abash
abass
abassan
abassi
abat
abattoir
abaya
abayu
abaza

abb
abba
abbado