****

**COMP5324 Internet Information Retrieval Report**

**Assignment 1: A Simple Index and Query System**

|  |  |
| --- | --- |
| **Name** | **Student ID** |
| Chen Qi | 18083945G |
| Zhao Haiqi | 18087348G |

### 

Table of Contents

[Readme 3](#_Toc1743563)

[Functions of the program 3](#_Toc1743564)

[Introduction of the development 3](#_Toc1743565)

[Basic Requirement 3](#_Toc1743566)

[Advanced Requirements 3](#_Toc1743567)

[Superior Advanced Requirements 4](#_Toc1743568)

# Readme

For readme file, you can also check our Github’s Repo for **Installation Guide** and **User Manual** here.

# Functions of the program

## Main.py

Continutly detecting user’s input and handle error inputs. And call different functions for different input corresponded.

## FileReading.py

#### fileReading(file\_path):

#### Read files from the directory of basic search. It stores all the term of documents into an inverted index. Convert all word into lowercase.

#### A stop word list is used to delete the common words in all files. The file name and the term appear in each typical file with its times would be stored in a nested dictionary.

#### This function returns a dictionary with {term:file} pairs. The repetitive file name is recorded which allows the program to calculate the times that one term appears in a file.

#### This function also returns a dictionary contains the position information of each word. It returns a dictionary with {file:[]} pairs. The array in value store the words in the file in order. The program can, therefore, access to the position of each word.

#### termVectorFileReading(file\_path):

## Read files from the directory of vector search. It searches the directory which inputs by the user and opens all the non-directory files. The string lines inside the file will be stored and broke into an array by a blank. The array is composited with words appear in the file. A stop word list is also applied.

## The nested dictionary is composited with {term : times} pairs while the out layer is the {file : {term:times}} pairs.

## The times will then be transformed into normalization format by using the vector space model. The Euclidean weights are applied in this function.

## Function.py

#### IDF(document\_frequency,total\_document\_number):

Calculate the IDF using the formula of log10(N/df)

## Interface.py

#### \_\_init\_\_(self):

Display the command line interface of main menu to user.

#### do\_query\_simple(self,file\_collection):

Get the query of user input of command line.

#### do\_query\_boolean(self,file\_collection):

Get the query of user input of command line of using Boolean searching.

#### do\_query\_multiwords(self,file\_collection,position\_information):

Get the query of user input of command line of using Phase searching.

#### do\_query\_ranking(self,final\_word\_list,file\_collection,doc\_idf):

Get the query of user input of command line of using Vector Space Ranking searching.

#### do\_path\_basic(self):

Get the path of documents from user input.

#### do\_path\_vector(self):

Get the path of documents from user input when using vector space searching.

#### display\_results(self,data):

Display the searching’s results.

#### search\_mode(self):

Get the searching’s mode of user input. For user to select the searching’s mode.

#### do\_help(self):

Display the command line instruction to users.

## SearchModes.py

#### simpleSearch(query,inverted\_index):

#### Using an inverted index to get the results of the query using the index of the dictionary in python. Return the results together with the counted number to the command line interface.

#### booleanSearch(query,inverted\_index):

Check the user input of keyword ‘and’ or ‘or’ first. Separate the query term and store in a query’s list. For ‘AND’ search, check all the query’s term in the inverted index vector. Then delete the document’s result when it was not contained in the later query’s term. For ‘OR’ search, add all the document occur in the for loop searching.

#### multiwordsSearch(query,totalInfo):

#### Get the query list and find out the results of the first query using the inverted index of the dictionary in python. The inverted index and position information are packed in the object totalInfo. Then, check if the next word in the query list is the same as the next word in position information array. If not, delete the file in the final list.

#### rankingSearch(query,final\_word\_list,totalInfo):

# The query, final word list, normalized vector, and IDF information are used in this function. The function will turn the query into a vector then use the IDF information to transform the query vector into a normalized vector. Then do the cosine similarity between the normalized query vector and every normalized file term vectors. Store the file name and ranking points in a dictionary as a {name: value} pair. Use the sorted function in python to rank the file name then return the results.

# Introduction of the development

## Basic Requirement

We developed the basic one first. We wrote a program for a command line interface that allowed the user to type the command in the program. For the first requirement (Use the Inverted Index approach to implement a simple Index and Query system for the given document set.) We wrote the program to read all documents in the directory. Also, allow the user to indicate the path of the directory. All words in English are separated with a space character. So we found all the space character and store the word in the inverted index vector. Also, count the number of occurrence of the term in the document. The vector of the term indicated which document contained the word and the number of terms. In Python, we use the dictionary. Which is contain key and value pair. In this case, the key is the term. The value is the vector of the document. For example, {“apple”:[“doc1”, ”doc2”]}. After that, the searching is to put the user’s input into the vector of the inverted index. The dictionary could get the correspond key-value pair of the term. For example, query = “apple”, then inverted\_index[query] would return the value [“doc1”,”doc2”]. Up to now, all Basic Requirements were fulfilled.

## Advanced Requirements

For the Boolean search, we decided to implement a program that first detected the “AND” searching or “OR” searching, then stored all the query’s term in a vector. For the “AND” search. We use a for loop function to check all the query’s term in the inverted index vector. Then delete the document’s result when it was not contained in the later query’s term. For example, the query is “Alice AND Bob” then the Alice result is [“doc1”, ”doc2”], and Bob result is [“doc1”], at that time, doc2 of the final result would be deleted. For the “OR” search. We just add all the document occur in the for loop searching.

For the phrase query’s searching. We store the term of in document in a vector that it also contains the position’s index. When processing the phrase query, we check the second’s term of the query is the next term (index + 1) of the first term or not. If it didn't then remove the document from the result.

## Superior Advanced Requirements

We had removed the stop word from documents. And transform all the term to lower case. Also, apply that to query.

For the requirement of ranking search. We use the vector space model to implement the TF-IDF and the cosine similarity. First, we did not use the same way of indexing in the basic version. We read all the document to make a term collection for the calculation of Document Frequency later. And store the total number of documents too. Then, we used a nested dictionary to store the Term Frequency of different documents. Then we created a Document frequency. We create a dictionary to store the document frequency of each term. The program read all the document to count the number of each term’s appearance. Then we use TF and df to calculate IDF by implemented the formula in the program. Then use TF and IDF to calculate TF-IDF weight. And also calculate the length of each document and do the normalization of the weight. For the query function, we pass the IDF to the searching program for calculating the weight of the query. Then we construct a dictionary of the query which is the term frequency in the same way. And calculate the weight by multiplied TF with IDF. And compute the cosine similarity by multiplied normalized weight of document with query’s weight. Use this as a score to store in the dictionary. Then the dictionary showed all the score of the document for this query. Sorted by descended and 0 scores of documented were removed. Then return the result to the user. The result showed the document with the score.

For improve the indexing and searching quality, we use Hashtable in Python for both indexing and searching. The dictionary component in Python are using the data structure of Hashtable.