INHOLLAND UNIVERSITY OF APPLIED SCIENCE

Assignment Information Retrieval

Semida A and Tal B

20/10/2018

Table of Contents

1. Introduction

This paper describes the implementation of a simple and small-scale search engine for multiple documents. Consisting of an index and an information retrieval (IR) model which are wrapped in a working application. The model chosen to be implemented is tf.idf. It is an information retrieval technique, which helps determine how important or rare a term/word is to a document in a collection (see sections *2. System architecture* and *3. Underlying IR model*) (Góralewicz, 2018).

In this assignment, a stemmer was used for the words of the documents. After applying the stemmer on all words of the documents, then the search model is applied. Once the query is imputed the user will get the ranked results of the documents, which best suited the query. (Manning, Raghavan & Schütze, 2008)

The rest of the paper is organized as follows, the architecture of the system is presented in Section 2, Section 3 describes the underline model, the implementation is presented in Section 4, while Section 5 focuses on testing and in the last section a conclusion is provided.

2. System architecture

The architecture of this application is based on simple 200 lines python script. It contains several functions for the information retrieval engine. As can be seen in the UML model from *Diagram 1* there is one script used for the implementation, indexer.py. It holds all the variables and the functions used to create the small-scale information retrieval engine.

The first part is creating the inverted dictionary containing all the stemmed word and their frequencies in the corresponding files. Following, the application writes the inverted dictionary to disk so it can be reusable for future queries.

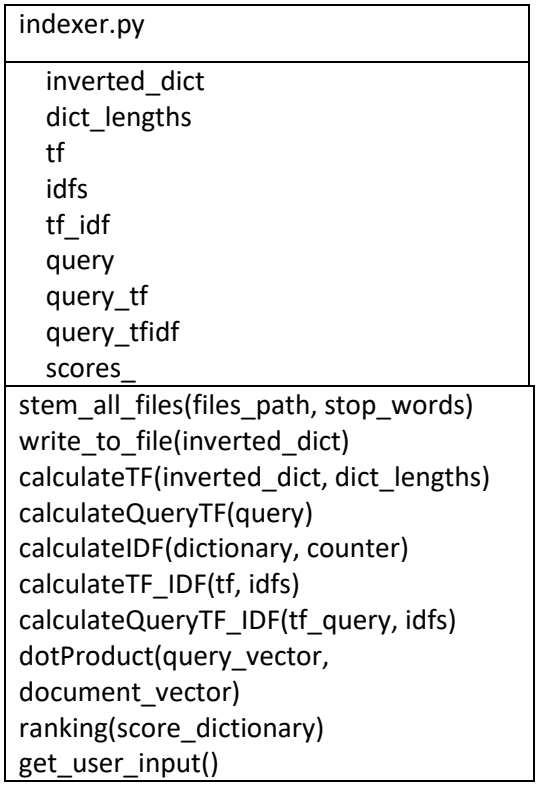

Diagram 1 – Unified Modeling Language (UML) Class Diagram

3. Underlying IR model

The goal of the model is to track similarity between the user query to words in documents. As stated above, the tf.idf model is used. Which takes into consideration the importance of each word in each document and in all documents. Meaning, if a word is repeated in every document, it is for sure not a very important one. While if a word is repeated in a document more than others, then the former is more important than the latter. (Góralewicz, 2018) (Manning, Raghavan & Schütze, 2008)

The calculation for the tf.idf is done using Formula 1. The first part of the formula, $\left(1 + log_2(tf_{t,d})\right)$, refers to the term frequency (tf) value. Where $tf_{t,d}$ looks at the term 't' frequency in document 'd'. While the second part of the formula, $log_2(\frac{N}{df_t})$, refers to the inverted document frequency (idf) value, which looks at df_t the term frequency in all documents. (Góralewicz, 2018) (Manning, Raghavan & Schütze, 2008)

$$tf.idf = \left(1 + log_2(tf_{t,d})\right) * log_2(\frac{N}{df_t})$$

Formula 1 – Formula for calculating tf.idf

4. Implementation

The user is first prompted to enter a path to the collections of documents through the function *get_user_input()* and a path to where the stop_words.txt files is located. Those paths are being used by the function *stem_all_files(files_path, stop_words)* (2. System architecture – *Diagram 1*) (1. CodeSnippet – Preprocessing). This includes replacing stopwords, split the documents into words, indexing the words, stemming the words and checking for the word if it is in the inverted keys list. Then the function *write_to_file(inverted_dict)* takes the returned inverted stemmed words and writes them in a text (.txt) file.

```
def stem_all_files(files_path, stop_words):
    # declaring a dictionary for counting the words
    inverted = {}
    # words in files will be written as strings
    files_as_string = ''
    # counter for the amount of files
    counter = 0
    # dictionary to hold lengths of files
    dict_lengths = {}
    for filename in os.listdir(files_path):
        #print(filename)
        counter += 1
        filepath = path + filename
        f = open(filepath, "r")
        for line in f:
            files_as_string += line
        # data - all the files becoming sting
        # all words in the list
        data = files_as_string
fi = open(stop_words, 'r')
# split each word in the stop_words file
stop_words_ = fi.read().split('\n')
# iterate over the words we collected
for word in data:
    # if a word is not a stop word
    if word not in stop_words_:
        # and if it is not a number
        if not word.isdigit():
            # then append it to the list and stem the word
            stemwords.append(stemmer.stem(word))
word_count = len(stemwords)
print(filename, word_count)
dict_lengths['%s' % filename] = word_count
files_as_string = ''
if True:
    # make the list of stemmed words as set (remove duplicates)
    for i in set(stemwords):
        # checking for the word if it is in the inverted keys list
        if i in inverted.keys():
            # if exist then increment count and write the file it exist in
            inverted[i].append((filename, stemwords.count(i)))
        else:
            # create new entry of the word and the file it exist in
            inverted[i] = [(filename, stemwords.count(i))]
```

1. CodeSnippet – Preprocessing

Then using the weight of a word in a given sentence is calculated using *calculateTF(inverted_dict, dict_lengths*), which takes in as parameter the inverted words and the length of each file. The tf value is calculated for each of the documents by using the tf formula (Section 3) (2. CodeSnippet – Document TF). In the same fashion the tf values for the user query was calculate using the function *calculateQueryTF(query)*, which took the user query as parameter (3. CodeSnippet – Query TF).

```
def calculateTF(inverted_dict, dict_lengths):
    tfDictionary = {}
    # length of the total word count
    for word, cnt in inverted_dict.items():
        for i in range(len(cnt)):
            for filename, length in dict_lengths.items():
                if filename in cnt[i][0]:
                    if word not in tfDictionary.keys():
                        tfDictionary[word] = [(filename, cnt[i][1] / length)]
                    else:
                        tfDictionary[word].append((filename, cnt[i][1] / length))
    return tfDictionary
```

2. CodeSnippet – Calculating the document TF

```
def calculateQueryTF(query):
    data = query.split()
    stemmed_query = []
    for word in data:
        # if a word is not a stop word
        if word not in stop_words:
            # and if it is not a number
            if not word.isdigit():
                # then append it to the list and stem the word
                stemmed_query.append(stemmer.stem(word))
    word_counter = {}
    # length of the total word count
    for word in stemmed_query:
        wordCount = stemmed_query.count(word)
        query_tf = 1 + math.log(wordCount)
        word_counter[word] = query_tf
    return word_counter
```

3. CodeSnippet – Calculating the Query TF

The next step is calculating the IDF of each word in every document. IDF measures how important a term is. While computing TF, all terms are considered equally important. In IDF we need to weigh down the frequent terms while scale up the rare ones. (***4. CodeSnippet*** – CalculateIDF(dictionary, len(dict_lengths))

```
def calculateIDF(dictionary, counter):
    idf_values = {}
    for word, cnt in dictionary.items():
        term_counter = 0
        for i in range(len(cnt)):
            term_counter += cnt[i][1]
            if word not in idf_values.keys():
                idf_values[word] = [(cnt[i][0], term_counter)]
            else:
                idf_values[word].append((cnt[i][0], term_counter))
    new_dict = {}
    for word, frequency in idf_values.items():
        #print(word)
        for i in range(len(frequency)):
            idf = math.log(1 + counter / frequency[i][1])
            if idf < 0:
                print(word, frequency[i], idf)
            if word not in new_dict.keys():
                new_dict[word] = [(frequency[i][0], idf)]
            else:
                new_dict[word].append((frequency[i][0], idf))
    return new_dict
```

4. CodeSnippet – CalculateIDF(dictionary, len(dict_lengths)

After calculating both tf and idf, we are able to calculate the tf * idf using the *calculateTF_IDF(tf, idfs)* function which gives us the score for a word in each document in compare to all the words in all files. (***5. CodeSnippet*** – Calculating the tf_idf)

```
def calculateTF_IDF(tf, idfs):
    tfidf = {}
    for word, value in tf.items():
        for i in range(len(idfs[word])):
            score = value[i][1] * idfs[word][i][1]
            if word not in tfidf.keys():
                tfidf[word] = [(value[0][0], score)]
            else:
                tfidf[word].append((value[i][0], score))
    return tfidf
```

5. CodeSnippet – Calculating the tf_idf

When tf_idf is finished, we can calculate the tf_idf of the query by looking at the tf of the query and the idf of all documents. The multiplication of the two will give a score for each word in the query. (**6. CodeSnippet** – calculateQueryTF_IDF(tf_query, idfs))

```
def calculateQueryTF_IDF(tf_query, idfs):
    query_tfidf = {}
    for word, value in tf_query.items():
        if word not in idfs.keys():
            score = 0
        else:
            for i in range(len(idfs[word])):
                score = value * idfs[word][i][1]
        query_tfidf[word] = score
    return query_tfidf
```

6. CodeSnippet – calculateQueryTF_IDF(tf_query, idfs)

The next step is computing the cosine similarity between the query and each document containing at least some of the words of the query (***7. CodeSnippet*** – dotProduct(query_vector, document_vector)).

```
def dotProduct(query_vector, document_vector):
    scores_dict = {}
    score = 0
    some_score = 0
    new_score = 0
    other_score = 0
    for word, value in query_vector.items():
        if word not in document_vector.keys():
            some_score += query_vector[word] * 0
        else:
            for i in range(len(document_vector[word])):
                score += query_vector[word] * document_vector[word][i][1]
                new_score += query_vector[word]**2
                other_score += document_vector[word][i][1]**2
                final = score / (math.sqrt(new_score) * math.sqrt(other_score))
                scores_dict[document_vector[word][i][0]] = final
    return scores_dict
```

7. CodeSnippet – dotProduct(query_vector, document_vector)

```
def ranking(score_dictionary):
    sorted_dict = sorted(score_dictionary.items(),  key=lambda x: x[1], reverse=True)
    pprint.pprint(sorted_dict)
    return sorted_dict
```

8. CodeSnippet – ranking(score_dictionary)

The last function in the script is for prompting the console and receiving the user input. The application will not stop until it will be stopped manually. Meaning, that after each query and receiving the top ranked documents (***8. CodeSnippet*** – ranking(score_dictionary)), another prompt will be done for asking another query from the user.

5. Testing

For the testing several documents, containing different content were used. Then, a query was entered and checked if any of the documents contain the query. Then in the output the names of the documents with the highest rank were displayed. For instance, when the query “i would like to buy furniture, airconditioner and would like help from an expert” was entered the documents most relevant were displayed. Figure 1 contains a snippet of the console testing and the output result.

```
Please Enter path to files: >? C:/Users/Home/Desktop/New folder/words/
Please Enter path to stop_words.txt: >? C:/Users/Home/Desktop/New folder/stop_eng.txt
adv_alad.txt 1037
aircon.txt 210
aminegg.txt 201
dopedenn.txt 958
empty.txt 4718
excerpt.txt 605
imonly17.txt 387
omarsheh.txt 883
poplstrm.txt 450
rainda.txt 1676
shulk.txt 1732
space.txt 469
traitor.txt 85
wolfcran.txt 264
Please Enter a Query: >? i would like to buy furniture, airconditioner and would like help from an expert
ranked documents
[('excerpt.txt', 0.9184213430027982),
 ('wolfcran.txt', 0.4875842412384256),
 ('adv_alad.txt', 0.4788338116402737),
 ('poplstrm.txt', 0.4654887873873258),
 ('shulk.txt', 0.45896066289816395),
 ('rainda.txt', 0.45810122355422295),
 ('empty.txt', 0.4496398055616656),
 ('space.txt', 0.44489983908395464),
 ('omarsheh.txt', 0.436488520443605),
 ('imonly17.txt', 0.4276657322853203),
 ('dopedenn.txt', 0.42434064440995384),
 ('aminegg.txt', 0.4149477595525123)]
Please Enter a Query:
>?
```

Figure 1– Console testing & output

6. Conclusions

The aim of this project was to implement a tf.idf model a search engine. The implemented application can parse documents in a folder and rank the documents based on the similarity to the query. The documents are ranked and displayed to the user.

7. Discussion

The search engine is scalable, as the model and the stemmer used can be change easily. Another aspect is that the application can only handles one word at a time. So, if a query containing a sentence were to be handled a more sophisticated implementation must be done.

8. References

Góralewicz, B. (2018). *The TF*IDF Algorithm Explained | Elephate*. [online] Elephate. Available at: https://www.elephate.com/blog/what-is-tf-idf/ [Accessed 19 Oct. 2018].

Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to information retrieval*. Cambridge: Cambridge university press