Tunisian Republic
Ministry of Higher Education and Scientific Research
University of Carthage
Higher Institute of Information and Communication Technologies

Federated Project Report
Course: Software Engineering and Information Systems.

Universal Search Bar with Voice Recognition and Local Indexing

Spidey

By Ilef Boualleg
Completed within ISTIC.

Academic supervisor: Ms. Imèn Hamrouni Trimech

Academic Year: 2022 - 2023

Introduction

Search engines are becoming more powerful by the second. They are crucial in today's world as they help us search for queries quickly and efficiently. In this context, Spidey, the universal search bar, mimics the behaviour of different browsers such as Google, Amazon etc. With features like voice recognition and local indexing.

Code Explanation

1)

```python
# Function to get stemmed words for documents
def get_stemmed_words_for_docs(docs_folder):
    stemmer = SnowballStemmer("french")
    docs_stemmed_words = {}

    for file_name in os.listdir(docs_folder):
        if file_name.endswith('.txt'):
            file_path = os.path.join(docs_folder, file_name)
            with open(file_path, 'r', encoding='utf-8') as file:
                text = file.read()
                words = word_tokenize(text.lower())
                words = [word for word in words if word.isalpha() and not word in stopwords.words('french')]
                stemmed_words = [stemmer.stem(word) for word in words]
                docs_stemmed_words[file_name] = stemmed_words

    return docs_stemmed_words

```

This code defines a function that processes text files present in a specified folder and returns a dictionary containing the stemmed words for each file. To obtain the stemmed words, the function first creates a stemmer object using the SnowballStemmer class from the nltk library for the French language. The text in each file is then converted to lowercase, tokenized into individual words, and filtered to remove non-alphabetic words and stopwords. Finally, the stemmer is applied to each remaining word to obtain the stemmed version, and the resulting dictionary of file names and corresponding stemmed words is returned by the function.

2)

```python
# Function to calculate TF for documents
def get_tf_for_docs(docs_folder):
    stemmer = SnowballStemmer("french")
    docs_tf = {}
    for file_name in os.listdir(docs_folder):
        if file_name.endswith('.txt'):
            file_path = os.path.join(docs_folder, file_name)
            with open(file_path, 'r') as file:
                text = file.read()
                words = word_tokenize(text.lower())
                stemmed_words = [stemmer.stem(word) for word in words]
                freq_dict = {}
                for word in stemmed_words:
                    if word in freq_dict:
                        freq_dict[word] += 1
                    else:
                        freq_dict[word] = 1
                docs_tf[file_name] = freq_dict
    return docs_tf

```

This code defines a function that takes a folder path as input and returns a dictionary where the keys are file names and the values are dictionaries that contain the term frequency (TF) of each word in the corresponding file. To calculate the TF, the function tokenizes each file into words, applies a stemmer to each word to

obtain its base form, and then creates a frequency dictionary that counts the number of times each word appears in the file. The resulting dictionary of file names and corresponding TF dictionaries is returned by the function

3)

```
114  # Function to calculate IDF
115  def get_idf(docs_folder):
116      stem_docs = get_stemmed_words_for_docs(docs_folder)
117      idf = {}
118
119      for doc_id in stem_docs:
120          for word in set(stem_docs[doc_id]):
121              if word in idf:
122                  idf[word] += 1
123              else:
124                  idf[word] = 1
125
126      n_docs = len(stem_docs)
127      for word in idf:
128          idf[word] = np.log(n_docs / idf[word])
129
130      return idf
131
```

The get_idf function calculates the inverse document frequency (IDF) of each word in a collection of documents located in a given folder. It first obtains the stemmed words for each document by calling the get_stemmed_words_for_docs function. It then counts the number of documents in which each word appears and calculates the IDF value for each word as the logarithm of the total number of documents divided by the number of documents in which the word appears. Finally, the function returns a dictionary containing the IDF values for each word in the collection of documents.

4)

```python
# Function to calculate TF-IDF weights for documents
def get_weights(docs_folder):
    stem_docs = get_stemmed_words_for_docs(docs_folder)
    idf = get_idf(docs_folder)
    weights = {}

    for doc_id in stem_docs:
        total_words = len(stem_docs[doc_id])
        tf_idf_dict = {}
        for word in stem_docs[doc_id]:
            tf = stem_docs[doc_id].count(word) / total_words
            tf_idf = tf * idf[word]
            tf_idf_dict[word] = tf_idf
        weights[doc_id] = tf_idf_dict

    return weights

```

The get_weights function calculates the TF-IDF score for each word in a collection of documents located in a given folder. It uses the get_stemmed_words_for_docs and get_idf functions to obtain the stemmed words and IDF values for each word in the collection of documents. For each document, the function calculates the TF-IDF score for each word as the product of its term frequency and IDF value. The result is stored in a dictionary and returned, which can be used to rank the documents based on their relevance to a given query.

4)

```python
# Function to read the user query
def get_query():
    query = questionField.get()

    stemmer = SnowballStemmer("french")  # Define the stemmer

    query = query.lower()
    query = word_tokenize(query)
    query = [word for word in query if word.isalpha() and word not in stopwords.words('french')]
    query = [stemmer.stem(word) for word in query]
    print(query)
    return query
```

The get_query function obtains a user input query from a GUI text field and processes it for use in a search engine. It uses the SnowballStemmer from the nltk package to obtain the stem of each word in the query. The function also tokenizes the query into individual words, removes any non-alphabetic characters and stop words in French, and converts each word to lowercase. The processed query is returned as a list of stemmed words. The function also prints the processed query to the console for debugging purposes.

5)

```python
# Function to search documents based on the query
def search_documents(query):
    # Get the TF-IDF weights for each document in the collection
    docs_weights = get_weights('../docs_folder')

    # Calculate the term frequency of each word in the query
    query_word_counts = Counter(query)

    # Find the maximum term frequency in the query
    max_word_count = max(query_word_counts.values())

    # Normalize the term frequencies in the query by the maximum term frequency
    query_word_freq = {word: count / max_word_count for word, count in query_word_counts.items()}

    # Calculate the TF-IDF weight for each word in the query
    query_weights = {}
    idf = get_idf('../docs_folder')

    for word, count in query_word_counts.items():
        if word in idf:
            tf = count / max_word_count
            tf_idf = tf * idf[word]
            query_weights[word] = tf_idf
```

```python
    # Calculate the cosine similarity between the query and each document
    similarities = {}

    for doc_id, doc_weights in docs_weights.items():
    # Get the weights for the current document
        doc_vector = np.array(list(doc_weights.values()))

        # Create a query vector by assigning weights to the words in the document, and 0 for those not present in the document
        query_vector = np.array([query_weights.get(word, 0) for word in doc_weights.keys()])

        # Calculate the magnitude (length) of the document and query vectors
        doc_magnitude = np.linalg.norm(doc_vector)
        query_magnitude = np.linalg.norm(query_vector)

        # Check if the magnitude of the document or query vector is 0 (to avoid division by 0)
        if doc_magnitude == 0 or query_magnitude == 0:
            similarity = 0
        else:
            # Calculate the cosine similarity between the document and the query vectors
            similarity = np.dot(doc_vector, query_vector) / (doc_magnitude * query_magnitude)

        # Add the similarity score for the current document to the dictionary of similarities
        similarities[doc_id] = similarity
```

```python

    # Sort the documents by relevance
    relevant_documents = [(doc_id, similarity) for doc_id, similarity in similarities.items() if similarity > 0]
    relevant_documents = sorted(relevant_documents, key=lambda x: x[1], reverse=True)

    # Return a list of relevant documents, sorted by relevance
    return relevant_documents
```

The cosine similarity score indicates the degree of similarity between the query and each document, with higher scores indicating greater similarity.

6) Main execution:

```
def search():
    if questionField.get()!='':
        if temp.get()=='google':
            webbrowser.open(f'https://www.google.com/search?q={questionField.get()}')
            print(questionField.get())

        if temp.get()=='duck':
            webbrowser.open(f'https://duckduckgo.com/?q={questionField.get()}')

        if temp.get()=='amazon':
            webbrowser.open(f'https://www.amazon.com/s?k={questionField.get()}&ref=nb_sb_noss')

        if temp.get() == 'youtube':
            webbrowser.open(f'http://youtube.com/results?search_query={questionField.get()}')
        if temp.get() == 'local':
            # Main execution
            query = get_query()
            if not(query):
                messagebox.showerror("Error","After deleting the stopwords, the query became empty")
            else:
                relevant_documents = search_documents(query)

                # Print the relevant documents
                if relevant_documents:
                    resultText.delete('1.0', END) #Clear the result box
                    resultText.insert(END,"Relevant Documents:\n")
                    for doc_id, similarity in relevant_documents:
                        resultText.insert(END, f"{doc_id} Similarity score: {similarity}\n")


                else:
                    resultText.delete("1.0",END)
                    resultText.insert(END,"No relevant documents found.")
    else:
        messagebox.showerror('Error','There is nothing to be searched')
```

7) Check out the code with its resources and explanation:
https://github.com/henlo-ilef/SearchBar

Conclusion

In conclusion, the code implements a search engine that can find the most relevant documents in a given set of documents, based on a user's query. The search engine first preprocesses the documents and query by tokenizing, stemming, and removing stop words. Then, the engine calculates the term frequencies and inverse document frequencies to assign weights to each term in the query and document. Finally, the engine computes the cosine similarity between the query and each document, and ranks the documents by their relevance to the query.