IRTM Homework2 Report

B10705054 劉知微

1 Execution Environment

The code is executed in an Anaconda Jupyter Notebook environment.

2 Programming Language

The programming language used is Python 3.

3 Execution Method

To run the code in a Jupyter Notebook, no additional environment setup is required. However, if you are using VS Code or another IDE, you may need to install the necessary packages, such as `numpy` and `nltk`. The following modules should be included in the code:

```
from nltk.stem import PorterStemmer
import os
import numpy as np
from collections import defaultdict
import math
```

Please ensure that the `data` folder and the `stopwords.txt` file (which is included in the folder) are prepared in the same directory as the code. The folder and stopwords file paths should be defined as follows:

```
document_folder = "./data"
stopword_file = './stopwords.txt'
```

Lastly, to execute the code in Jupyter Notebook, simply click the "Run" button (play icon) in the toolbar section. This will run the code in the active cell.

4 Workflow

4.1 Tokenization

To tokenize the text, I use the built-in function `split()` to separate the words based on spaces. The resulting words are saved as tokens. Then, the `lower()` function is applied to convert all letters in the tokens to lowercase. I utilize the `PorterStemmer()` from `nltk.stem` to apply the Porter Stemming algorithm. For each word in the tokens, if the word is alphanumeric and not in the stopwords set, I stem the word and save it into the list `filtered_tokens`.

```python
# Function for tokenization, lowercasing, and stemming
def tokenization(text):
    tokens = text.split()
    tokens = [word.lower() for word in tokens]
    porter_stemmer = PorterStemmer()
    # Filter tokens: stem words, keep only alphanumeric tokens, and remove stopwords
    filtered_tokens = [
        porter_stemmer.stem(word)
        for word in tokens
        if word.isalnum() and word not in stop_words
    ]
    return filtered_tokens
```

I read the files from the folder `data`. Each file path is generated using `os.path.join()`, and the file is opened for reading. Every file goes through the previously mentioned `tokenization()` function, and the tokenized content is appended to the list `documents`.

```python
# List to store tokenized documents
documents = []

# Read and process each file in the folder
for i in range(1, 1096):
    file_path = os.path.join(document_folder, f"{i}.txt")
    with open(file_path, 'r', encoding='utf-8') as file:
        # Read the file content
        content = file.read()

        # Tokenize the content
        processed_content = tokenization(content)
```

```

    # Append the tokenized content to the documents list
    documents.append(processed_content)
```

4.2 Constructing a Dictionary

I used the `defaultdict` class from the `collections` module to create a dictionary. First, I constructed a set of unique terms from each document, which is an essential step for counting document frequency (DF). This ensures that if a document contains two or more identical words, the DF count is not inflated. For each term, I incremented its corresponding value in `term_df`. The `defaultdict` is ideal for this process as it automatically initializes new terms with a default count of zero.

```
term_df = defaultdict(int)
for doc in documents:
    unique_terms = set(doc) # Create a set of unique terms from each document
    for term in unique_terms:
        term_df[term] += 1 # Increment the DF count for each unique term
```

Next, I sorted the terms alphabetically and assigned an index to each term.

```
sorted_terms = sorted(term_df.items(), key=lambda x: x[0]) # Sort terms alphabetically
dictionary_entries = []
for idx, (term, df) in enumerate(sorted_terms, start=1): # Assign index starting from 1
    dictionary_entries.append((idx, term, df)) # Create entries with index, term, and DF value
```

Lastly, I saved the terms to a file called `dictionary.txt`. The dictionary contains three columns: `t_index`, `term`, and the corresponding `df` value.

```
dictionary_file = os.path.join('./dictionary.txt') # Path to the output file
with open(dictionary_file, 'w', encoding='utf-8') as f:
    # Write header
    f.write(f"t_index\tterm\tdf\n")
    # Write each dictionary entry
    for entry in dictionary_entries:
```

```
    f.write(f"{entry[0]}\t{entry[1]}\t{entry[2]}\n")
```

4.3 TF-IDF Unit Vector

I created a dictionary based on terms, with each entry containing the term index and its inverse document frequency (IDF) value. The `math.log` function is used to calculate the IDF, which is computed as $\log(n/\text{df})$, where n is the total number of documents, and df is the document frequency of the term. Then, I stored both the IDF value and the index in the dictionary `idf_dict`.

```
# Calculate IDF value
N = 1095
idf_dict = {}
for idx, term, df in dictionary_entries:
    idf_value = math.log10(N / df) if df > 0 else 0
    idf_dict[term] = (idx, idf_value)
    # Store the term index and its corresponding IDF value
```

Next, I calculated the term frequency (TF) by counting the occurrences of each term in a document using a `defaultdict` called `term_count`. The TF value is computed as the number of occurrences of the term divided by the total number of terms in the document. I then stored the product of TF and IDF in `tfidf_vector` for each document, indexed by the term index. Following that, I normalized the vectors to unit vectors. Finally, I sorted the dictionary by term index and appended the sorted TF-IDF vector for each document to the list `tfidf_results`.

```
# Create a list to hold TF-IDF results for each document
tfidf_results = []

for doc in documents:
    # Count term frequencies
    term_count = defaultdict(int)
    for term in doc:
        term_count[term] += 1
    # Total number of terms in the document
    total_terms = len(doc)
    tfidf_vector = {}
    for term, count in term_count.items():
        tf = count / total_terms # Calculate TF
        idx, idf_value = idf_dict.get(term, (0, 0))
```

```
        tfidf_vector[idx] = tf * idf_value # Calculate TF-IDF

    magnitude = np.linalg.norm(list(tfidf_vector.values())) # Compute the magnitude
    if magnitude > 0:
        tfidf_vector = {idx: value / magnitude for idx, value in tfidf_vector.items()} # Normalize values

    # Sort tfidf_vector by idx (the key)
    sorted_tfidf_vector = dict(sorted(tfidf_vector.items()))
    tfidf_results.append(sorted_tfidf_vector)
```

Finally, I stored all the TF-IDF values in the `output` folder, with each document file containing two columns: `t_index` and `tf_idf` values.

```
output_folder = os.path.join( "./output/")
os.makedirs(output_folder, exist_ok=True)
for i, tfidf in enumerate(tfidf_results):
    filename = os.path.join(output_folder, f"{i + 1}.txt")
    with open(filename, 'w', encoding='utf-8') as file:
        file.write(f"t_index\ttf_idf\n") # Header
        for idx, tfidf_value in tfidf.items():
            file.write(f"{idx}\t{tfidf_value}\n") # Write each TF-IDF pair
```

4.4 Cosine Similarity

Cosine similarity is calculated by first creating a union of the terms that documents x and y contain. For each document, a vector is formed by checking if the document has a corresponding *tf-idf* value for each term. If a term is absent, the *tf-idf* value is set to zero. Once the vectors are created, the cosine similarity is computed using the formula:

$$\text{Cosine Similarity} = \vec{x} \cdot \vec{y}$$

where $\vec{x} \cdot \vec{y}$ represents the dot product of the two unit vectors.

```
def cosine(Docx, Docy):
    tfidf_x = tfidf_results[Docx - 1]
    tfidf_y = tfidf_results[Docy - 1]

    # Get the union of keys from both TF-IDF dictionaries
```

```
    keys = set(tfidf_x.keys()).union(set(tfidf_y.keys()))

    # Create vectors for each document based on the keys
    vector_x = np.array([tfidf_x.get(key, 0) for key in keys])
    vector_y = np.array([tfidf_y.get(key, 0) for key in keys])

    # Calculate cosine similarity
    cosine_similarity = np.dot(vector_x, vector_y)
    return cosine_similarity
similarity = cosine(1, 2) # Calculate similarity between Document 1 and Document 2
print(f"Cosine similarity between Document 1 and Document 2: {similarity}")
```

Cosine similarity between Document 1 and Document 2: 0.1672930057087566