PROJECT 1

Text preprocessing with NLTK

Gabriel Martinica

The program is “p1.py” and be run using python such as:

```
python3 p1.py
```

In my computer it took about 20 seconds to run and creates 25 files, divided into 5 folders.

The inputs for the program can be altered in the main() function

```
def main():
    path = "./reuters21578"
    last_index = 5
    stop_words_list = stopwords.words('english')
    docs = read_files(path, last_index)
    docs = tokenize(docs)
    docs = make_lowercase(docs)
    docs = apply_porter_stemmer(docs)
    docs = remove_stop_words(docs, stop_words_list)
```

Here path is a string with the path of the directory which contains the files to be read.

The last index is the number of files to be read, called last index as it is the limit up to which the contents of the directory will be read. In this case 5.

The stop word list by default is the one given by the NLTK module, but in theory it can accept any other list of words.

The 5 modules are separated into 5 functions, which will be explained below:

```
def read_files(path: str, last_index: int) -> dict:
    """Reads sgm files from a given directory and removes metadata.
Returns a dict with the form
    {document: text}"""
```

This is the first function which takes as input the directory and the number of files to read. It proceeds to read the contents of the directory:

```
file_list = [file for file in os.listdir(path) if
file.endswith(".sgm")]
file_list.sort()  # For indexing the first 5 documents
testing_file_list = file_list[:last_index]  # Getting first 5 documents
```

For this, we read the contents using os.listdir() and make a list of only the documents which have the .smg extension

It proceeds to sort the list just to make indexing easier as the assignment required to process the first five documents, so it was assumed it was the first 5 in alphabetical order, and since the OS does not necessarily indexes the documents in that order it was sorted.

We then split the list to get only the first 5 documents

```
directory = "step1files"
make_directory(directory)

docs = {}
```

We create a dictionary that will later contain the contents of each file in the format {file: text}
Here we create a directory with a helper function make_directory

```
def make_directory(path: str):
    """Makes directory with the given path. Name included in path"""
    try:
        os.makedirs(path, exist_ok=True)  # Make directory for new
files
    except OSError:
        print("ERROR MAKING DIRECTORY " + path)
```

which creates a directory with the given name and path and handles errors such as directory already created.

```
for filename in testing_file_list:
    text = get_text_content(path + '/' + filename)
    docs[filename] = text
    try:
        # Write output to new file
        new_file = open(directory + '/' + "altered-" + filename, "w")
        new_file.write(text)
        new_file.close()
    except IOError:
        print("Error: File does not exist")
return docs
```

Now we loop through all the files in the list, and for each file we get the raw text. Here we start with a helper function get_text_content to read the file and return a string with just the text of the articles. Handles any IOError.

```
def get_text_content(path: str) -> str:
    """Gets the text of each article. Reads file line by line up until <BODY>
tag found, appends all the contents of the
    article's body up until it finds the word 'reuter' which is used to end
each article. Does this until EOF"""
    text = ''
    currently_searching = True
    try:
        with open(path, 'rt') as file:
            for line in file:
                if currently_searching:
                    line_index = line.find("<BODY>")
                    body_found = True if line_index > -1 else False  # True if
body tag found
                    if body_found:
                        currently_searching = False
                        text_to_add = line[line_index + 6:]
                        text += text_to_add
                else:
                    if line.lower().find(" reuter") > -1:
                        currently_searching = True
                    else:
                        text += line
        file.close()
    except IOError:
        print("Error: File does not exist")
    return text
```

This helper function has the purpose of obtaining the raw text from the articles. In this case, the idea is that I open a file and start reading it line by line. For each line, I verify if I am in the state of “search”. This means I am looking for the beginning of an article by looking for the tag <BODY> in the line. If found, I append what follows the tag to the text variable which is a string containing the contents of each article. I continue appending line by line up until the article ends, for which the “Reuter” word appears in a single line. I stop appending at this point and come back to the searching state. I rinse and repeat until the end of the file. This provides me with the text inside the BODY tag of each article.

Then proceeds to store the modified text into the dictionary with {filename: text}

I find this option to be very concise to be able to extract the raw content of each file since it has so much metadata. The downside is that the metadata is useful when trying to categorize each article, and this categorization is lost doing it this way. Another thing that is lost is the documentID. However, if we merely want to get the content of each article and we work with the data finding trends in the general corpus instead of categorizing, this approach is simple and works fine.

```python
try:
    # Write output to new file
    new_file = open(directory + '/' + "altered-" + filename, "w")
    new_file.write(text)
    new_file.close()
except IOError:
    print("Error: File does not exist")
```

The rest is just writing the contents of the dictionary to a file, handling errors

Next is tokenize function

```python
def tokenize(docs: dict) -> dict:
    """Given a dict with {document: text}, tokenizes each element"""
    directory = "step2files"
    make_directory(directory)
    punctuation_list = [*".,:;-<>{}()[]~`"]
    double_symbols = ['""', "''", "``", "..."]
    punctuation_list += double_symbols
    for filename in docs:
        docs[filename] = word_tokenize(docs[filename])
        # Write output to new file
        new_file = open(directory + '/' + "altered-" + filename, "w")
        for word in docs[filename]:
            if word in punctuation_list:
                docs[filename].remove(word)
            else:
                new_file.write(word + '\n')
        new_file.close()
    return docs
```

The input for this function is the dictionary created in the previous function that reads the file. This is a dictionary as {document: text} In here, the document is the same as the filename.

I then create a directory which will the store the output files of this function.

I loop through each file in the dictionary, and apply the word_tokenize() function to tokenize the text. The result is a list of words which replaces the old raw text. It returns this new dict.

I create a list of punctuation symbols to remove, as the list contains many punctuation marks that are not helpful. It does not account for all punctuation marks, and it is not perfect of course, it could improve by having a more extending list or by using a more suited regex.

It then proceeds to write the list of words per file that are not part of the punctuation list in another folder called step2files

The next function makes the list of words lowercase

```
def make_lowercase(docs: dict) -> dict:
    """Given a dict with {document: text}, makes all values of the dict
lowercase"""
    directory = "step3files"
    make_directory(directory)
    for filename in docs:
        lower_case = [w.lower() for w in docs[filename]]
        docs[filename] = lower_case
        # Write output to new file
        new_file = open(directory + '/' + "altered-" + filename, "w")
        for word in docs[filename]:
            new_file.write(word + '\n')
        new_file.close()
    return docs
```

Here the input is the dictionary obtained from previous function tokenize().

I loop through each file(document) in the dictionary and proceed to convert all words to lowercase using w.lower() function. I replace the values of the dict with this new list.

I write the contents of the dict to a folder step3files.

```python
def apply_porter_stemmer(docs: dict) -> dict:
    """Given a dict with {document: text}, apply Porter Stemmer
algorithm"""
    directory = "step4files"
    make_directory(directory)

    porter = PorterStemmer()

    for filename in docs:
        docs[filename] = [porter.stem(token) for token in
docs[filename]]
        new_file = open(directory + '/' + "altered-" + filename, "w")
        for word in docs[filename]:
            new_file.write(word + '\n')
        new_file.close()
    return docs
```

Here is the next function it has a similar idea as before. Once all words are lowercase, I apply the Porter Stemmer algorithm to each words in each file(document).

I write the contents of the dict to an output file in a folder step4files and return the new dict.

```python
def remove_stop_words(docs: dict, word_list: list) -> dict:
    """Given a dict with {document: text} and a list of stop words,
remove the stop words from text"""
    directory = "step5files"
    make_directory(directory)
    for filename in docs:
        docs[filename] = [w for w in docs[filename] if w not in
word_list]
        new_file = open(directory + '/' + "altered-" + filename, "w")
        for word in docs[filename]:
            new_file.write(word + '\n')
        new_file.close()
    return docs
```

Last step is removing stop words. In this case the function accepts the dictionary from previous function and a list of words. It proceeds to create a list of words using the document tokenized list only and only if the word is not in the stop word list, effectively removing any stop words in the new list. It assigns this list to the dictionary and proceeds to write the contents of the dict to a new directory step5files.

And that is it, those are all the functions and the pipeline for this assignment.