COMP472 – Project 1 Report

Rhina Kim – 40130779

September 22, 2022

Overview

The purpose of this project is to learn how to text preprocess using NLTK library. Given the Reuter's Corpus "Reuters-21578", this project aims to extract the raw text of each article from the corpus, tokenize the text for all articles. Then the project cleans the set of texts by applying Porter Stemmer and removing certain stop words.

Technologies

Language

- Python 3.8

Interpreter

- Pypy3

Python packages

- Beautiful Soup 4.9
- NLTK 3.7
    - *nltk.word_tokenize*
    - *nltk.stem.PorterStemmer*
    - *nltk.corpus.stopwords*

Python 3.8 is used as programming language for this project since it supports natural language processing with Python's NLTK package. Moreover, pypy3 is used as a replacement for original Python 3 interpreter since its runtime interpreter is faster. As this project requires iteration of massive number of large files, using pypy3 as an alternative interpreter seemed to be an optimal choice.

As for the python packages, NLTK is brought to this project in order to tokenize, to apply Porter Stemmer, and to remove stop words from the corpus. Beautiful Soup is given for extracting the text data from .sgm files which is composed of markup-languages like HTML and XML.

Pipeline Steps

Step 0 – Extract tar file (optional)

Given the collection of Reuters files "*Reuters-21578.tar.gz*", we need to extract and unzip the tar file. The program incorporated python file "***extract_tar.py***" for uncompressing the tar file.
The program first imports "tarfile" package which is already in the Python Standard Library. The program then opens tar.gz file (In this case, *Reuters-21578.tar.gz*) after specifying its file address. With the help of the tarfile package, program extracts the file in a specific folder defined in the code. Finally, the tar file is closed after completing all the operations. The following is the snapshot of the code:

```
file = tarfile.open(TAR_NAME + '.tar.gz')  # open tar file
# Extract file and save to the specific folder
file.extractall('./' + TAR_NAME_EXTRACTED)
file.close()
```

Step 1 – Read the Reuter's corpus and extract its raw texts

After extracting the Reuter's packaged folder, we notice that the folder consists of:

- **22 data files collection in SGM format**
  - Each of the first 21 files (reut2-000.sgm through reut2-020.sgm) contain 1000 documents
  - Last (reut2-021.sgm) contrains 578 documents
- SGML DTD file: describing the data file format
- 6 files describing the categories used to index the data

Only the 22 data files collection with SGM format is necessary for this project, thus we can discard the rest of the files.

From this pipeline step, all its code scripts are written inside the file "***p1.py*".**

Firstly, we begin by iterating Reuter's collection folder to read every sgm data files. We ignore other data files if it does not contain sgm format:

```
if not file.endswith(".sgm"):
    continue
```

During the iteration of Reuter's collection, we pass each sgm file to the function called "***extract_documents_from_corpus()***" which extracts its whole text from the file and further separate the text into articles. To do so, we need to find every *<Reuters>* opening and closing tag in the text using Beautiful Soup ***find_all()*** function which returns list of articles separated by *<Reuters>*. In this case with Reuter's corpus, since each Reuter's sgm files contain 1000 articles, the result will return a list of 1000 separated articles per file.
The script below is simplified in this report, which explains the above process:

```
def extract_documents_from_corpus(sgm_file):
    documents = {}

    soup = BeautifulSoup(sgm_file, 'html.parser')
    # find all content with <REUTERS> tag (returns a list of each doc/article)
    file_contents = soup.find_all("reuters")
    for content in file_contents:
        title = content.find("title")
        title = title.text if title else ""
        body = content.find("body")
        body = body.text if body else ""
        # form a text
        text = title + " " + body

        # assign document ID and make a dictionary of documents
        documents[NUM_DOCUMENTS] = text

    return documents
```

Once we have a resulting list of articles, we store them into the variable called "*file_contents*". From there, we again want to extract only the context inside <*title*> and <*body*> tags for each article, since those are only thing necessary for this text preprocessing project (we leave all the other information like metadata, date time, place, etc.).
After all, we make a dictionary of documents with article/document ID as its key and its content (title + body text) as value (In this case we will have 1000 unique document ID per sgm file). Finally, the function returns the dictionary variable which will be used in another function (pipeline step 2).
The dictionary output is as follows:

```
{
   "0": "INCO SEES NO MAJOR IMPACT FROM DOW REMOVAL Inco Ltd said it did …,
   "1": "FORMER EMPIRE OF CAROLINA <EMP> EXEC SENTENCED Mason Benson, …,
   "2": "DOCTORS FIND LINK BETWEEN AIDS, SMALLPOX VIRUS In a discovery …,
}
```

Step 2 – Tokenize texts inside every document, and generate postings

In this pipeline step, the program further splits the document text into words. We use ***word_tokenize()*** function offered by Python NLTK package to perform tokenization. Notice that it is important to remove all the special characters and punctuations from each token since these are unnecessary and can affect text preprocessing process.

```
def tokenize(documents):
    postings = []

    for doc_id, text in documents.items():
        for token in word_tokenize(text):
            # remove all the special characters and punctuations
            token = ''.join(e for e in token if token.isalnum())
            # generate postings
            postings.append((doc_id, token))

    return postings
```

Once tokenization processing is completed, we append document id and its token to empty list called "*postings*". Since there would be many values (tokens) in each unique document ID, we no longer append back the items into dictionary. The result postings list will be used for next pipeline step.
Here is the output of how the postings list looks like:

```
(0, 'INCO')
(0, 'SEES')
…
(1, 'FORMER')
(1, 'EMPIRE')
(1, 'OF')
…
```

Step 3 – Make all texts lowercase

This pipeline step is to make all the texts lowercase in order to avoid duplicated embeddings/terms. Similar as pipeline step 2, we iterate the postings list returned from step 2. The program takes *doc_id* and its corresponding word from iteration, lowercase the word only if the word is a type of string, and then append the updated pair of (*doc_id, word*) back to the postings list. The result postings list will be used for next pipeline step.

```
def lowercase(postings):
    n_postings = [] # new postings list
    for doc_id, word in postings:
        if isinstance(word, str):
            word = word.lower()
        n_postings.append((doc_id, word))

    return n_postings
```

Here is the output of how the postings list looks like:
(0, 'inco')
(0, 'sees')
…
(1, 'former')
(1, 'empire')
(1, 'of')
…

Step 4 – Applying Porter Stemmer

In pipeline step 4, we apply stemming on every word inside the postings list. We use ***PorterStemmer()*** function offered by Python's NLTK package to facilitate the stemming performance. Just like Step 3, we iterate each (*doc_id, word*) element pairs inside the postings list, apply stemming on the word, and append the newly updated (*doc_id, word*) element back to the postings list which will be used for next pipeline step.

```
def porter_stemmer(postings):
    stemmer = PorterStemmer()
    n_postings = []

    for doc_id, word in postings:
        word = stemmer.stem(word)
        n_postings.append((doc_id, word))

    return n_postings
```

Here is the output of how the postings list looks like:
(0, 'inco')
(0, 'see')
…
(1, 'former')
(1, 'empir')
(1, 'of')

…

Step 5 – Remove stop words

In last pipeline step, we remove stop words from the list of postings. The stop words will be defined beforehand through user input via command prompt: the program will give two choices, either to use pre-defined stop words list provided by NLTK package, or to use specific list of stop words entered via command prompt. This process is specified inside the function ***create_stop_words_list()***.

After deciding what to use for stop words, we compare every word inside postings list with the defined list of stop words to see if there's any match. If a word matches with the words inside the defined stop words list, we remove them from the postings list by skipping its iteration.

```
def remove_stop_words(postings, stop_words):
    n_postings = []
    for doc_id, word in postings:
        if word in stop_words:
            num_postings_removed += 1
            continue
        n_postings.append((doc_id, word))

    return n_postings
```

Finally, we append the newly updated (*doc_id, word*) element back to the postings list which will be final output for this project.

Here is the output of how the postings list looks like (Notice: word "of" with document ID #1 is removed since it is detected as one of the stop words):

(0, 'inco')
(0, 'see')
…
(1, 'former')
(1, 'empir')
(1, 'carolina')
…