COMP 479 - Project 1

François LaBerge

October 2, 2020

The python file packaged with this report contains the main logic required to cleans up the files of the raw Reuters corpus and extract its words to prepare it for indexation. The functions are meant to be called in a pipeline fashion.

Block 1 The purpose of block 1 is simply to extract the contents of Reuters ".sgm" files given a file path.

```
def block_reader(path):
    # WRITE YOUR CODE HERE vvvvvvvvvvvvvvvvv
    for file in [entry for entry in listdir(path) if entry[-4:] == '.sgm']:
        filePath = '{}/{}'.format(path, file)
        with open(filePath, 'r', encoding='latin-1') as f:
                yield f.read()
    # WRITE YOUR CODE HERE ^^^^^^^^^^^^^^^^
```

In the first line we begin by creating a list of all the ".sgm" file names. In order to do so, we get the list of all files in the directory, then filter it by selecting the files that finish with ".sgm". When then iterate of that list. With each file we first create a path to the file, then open a the associated file and read it's full content while outputting it to the generator.

Block 2 In block 2, we begin breaking down the corpus by splitting the file contents into separate documents, which contain news article.

```
def block_document_segmenter(INPUT_STRUCTURE):
    # WRITE YOUR CODE HERE vvvvvvvvvvvvvvvvv
    for file in INPUT_STRUCTURE:
        remainingContent = file
        startIndex = remainingContent.find('<REUTERS')
        stopIndex = remainingContent.find('<REUTERS', startIndex+1)
        while startIndex != -1:
            if(stopIndex == -1):
                yield remainingContent[startIndex:]
            else:
                yield remainingContent[startIndex: stopIndex-1]
            remainingContent = remainingContent[stopIndex:]
            startIndex = remainingContent.find('<REUTERS')
            stopIndex = remainingContent.find('<REUTERS', startIndex+1)
    # WRITE YOUR CODE HERE ^^^^^^^^^^^^^^^^
```

The documents are bounded by "⟨REUTERS⟩" XML tags, therefore we will use these to split the file's contents into individual documents. We start by finding the position in the file of the next and second next "⟨REUTERS⟩" tag. Since articles are right next to each other, we can get the position of the closing "⟨/REUTERS⟩" tag for the first opening tag using the second opening tag. Using these two positions, we take the sub-string containing a document and yield it. We then remove the everything that came before the document from the file string, allowing use to repeat the process to retrieve all documents efficiently. We repeat this process until no documents are left in the file as seen by the while loop condition. A special case occurs for the last document which is easily handled with an *if* statement.

Block 3 In the third step of the pipeline, we clean the documents by extract only the valuable text of the document and leaving the metadata behind.

```
def block_extractor(INPUT_STRUCTURE):
    # WRITE YOUR CODE HERE vvvvvvvvvvvvvvvvv
    for document in INPUT_STRUCTURE:
        root = BeautifulSoup(document, "xml")
        docId = int(root.find('REUTERS')['NEWID'])
        textRoot = root.find('TEXT')
        if textRoot.has_attr('TYPE'):
            if textRoot['TYPE'] == 'BRIEF':
                content = textRoot.find('TITLE').text
                yield {"ID": docId, "TEXT": content}
        else:
            content = textRoot.find('TITLE').text
            content += textRoot.find('BODY').text
            yield {"ID": docId, "TEXT": content}
# WRITE YOUR CODE HERE ^^^^^^^^^^^^^^^^
```

This step requires a little bit more granularity in our approach than the other steps. This is why the BeautifulSoup library is used in this block. We start by parsing the document with XML using BeautifulSoup. In a single line, we get the docID from the "NEWID" attribute of the "REUTERS" tag using BeautifulSoup. This ID will be used later on to construct the docID-text dictionaries. Then, we need to figure out which type of text document we are dealing with to parse it properly. The type of document can be determined by the "TYPE" attribute of the "TEXT" tag. We use BeautifulSoup to retrieve the attribute and the tag. Three types of documents are relevant to us in this step:

- The type is "UNPROC". This type of stands for "unprocessed". These documents required are not valuable for indexing, therefore we want to ignore those documents altogether.
- The type is "BRIEF". This type of document contains only a title and no actual article body. Therefore, we want to only retrieve the title of these documents
- If the "TEXT" tag does not have a "TYPE" attribute, then it is a normal news article. We want to retrieve both the title and body of these documents.

In order to handle these three cases we use a series of simple *if* statements. In the case of a brief new article we use BeautifulSoup to retrieve the "TITLE" tag then access the text property to retrieve its contents. Regular news document are similar, except we also retrieve and the contents of the "BODY" tag and concatenate it with the "TITLE" tag. We then create a docID-text dictionary and yield it.

Block 4 For block 4, we further split the document text into words, in a process known as tokenization.

```
def block_tokenizer(INPUT_STRUCTURE):
    # WRITE YOUR CODE HERE vvvvvvvvvvvvvvvvv
    for dictionnary in INPUT_STRUCTURE:
        for token in word_tokenize(dictionnary['TEXT']):
            yield (dictionnary['ID'], token)
    # WRITE YOUR CODE HERE ^^^^^^^^^^^^^^^^
```

To perform tokenization, we leverage NLTK's *word_tokenize* method. We parse through every dictionary of document applying the *word_tokenize* method to the "TEXT" attribute of the dictionary. This method splits to text into a list of token, which we can then parse through yielding a tuple of the dictionary's ID and the token for every token.

Block 5 In block 5, we perform stemming on the words that we tokenized in the previous step.

```
def block_stemmer(INPUT_STRUCTURE):
    # WRITE YOUR CODE HERE vvvvvvvvvvvvvvvvv
    porter = PorterStemmer()
    for tup in INPUT_STRUCTURE:
        yield(tup[0], porter.stem(tup[1]))
    # WRITE YOUR CODE HERE ^^^^^^^^^^^^^^^^
```

Stemming the token is also rather easy using the *PorterStemmer* class of NLTK. We first initialize a new *PorterStemmer*, then we sift through the docID-token pairs and yield a new pair of docID-StemmedToken. To stem the token we use the *stem* method of the *PorterStemmer* instance.

Block 6 In the last step, we remove a stop words from the list of words previously stemmed from an input list of stop words.

```
def block_stopwords_removal(INPUT_STRUCTURE, stopwords):
    # WRITE YOUR CODE HERE vvvvvvvvvvvvvvvvv
    porter = PorterStemmer()
    stemmedStopWords = [] if stopwords == None else [porter.stem(word) for word in stopwords.split()]
    for tup in INPUT_STRUCTURE:
        if not tup[1] in stemmedStopWords:
            yield(tup[0], tup[1])
    # WRITE YOUR CODE HERE ^^^^^^^^^^^^^^^^
```

We begin by processing to input stowords string to make it easier to use down the line. Using python's list comprehension feature, we start by splitting the stopwords string on whitespace characters so that words stopwords like "apple orange" are understood like ['apple', 'orange'] instead of ['a','p','p','l','e',' ','o','r','a','n','g','e']. Then we stem the stopwords in that list to that if the term of a stopword has been stemmed, those tokens will also be removed. Removing the stopwords from the docID-token pair stream is then easier. We sift through the stream then check if the token is in then stopwords list. If it is not we yield the same pair, otherwise we ignore it.