Gabriel Martinica

ID: 40120255

```python
from nltk.tokenize import RegexpTokenizer
from nltk import word_tokenize
from nltk.stem.porter import PorterStemmer
from tabulate import tabulate
```

First, I import the libraries I am using. Now it is just nltk and tabulate. There would be many more libraries that could accelerate and optimize the program. For example, numpy optimizes arrays as they are faster and occupy less memory. Many more array manipulation could be enhanced this way in the future. If further developed, I would explore these ideas.

```python
content_tokenizer = RegexpTokenizer('<TEXT.*?>(.*?)</TEXT>')
article_tokenizer = RegexpTokenizer('<REUTERS(.*?)</REUTERS>')
id_tokenizer = RegexpTokenizer('NEWID="(.*?)"')
metadata_tokenizer = RegexpTokenizer('<.*?>')
html_entities_tokenizer = RegexpTokenizer('^&.*?;')
postings_list = {}
f = []
reuters_corpus = []
```

In here, I declare my regex tokenizers which will be used to extract all the good tokens from the text. In this case, I know some people would decide to separate each document just by taking the body. In my case, I decided to do a little more processing and decided to extract from the <TEXT> tags. This is because I noticed there are some articles that do not contain a body but do contain text, and I also think the extra information like the title gives a better insight on what the article is ab out.This is why the metadata_tokenizer exists, because later I remove the tags from the articles. I also attach the articles to the NEWID indicated in them. In the articles, there are some HTML entities that appear in the text that I remove as well like &3323;

In here, I declare my regex tokenizers which will be used to extract all the good tokens from the text. In this case, I know some people would decide to separate each document just by taking the body. In my case, I decided to do a little more processing and decided to extract from the <TEXT> tags. This is because I noticed there are some articles that do not contain a body but do contain text, and I also think the extra information like the title gives a better insight on what the article is ab out.This is why the metadata_tokenizer exists, because later I remove the tags from the articles. I also attach the articles to the NEWID indicated in them. In the articles, there are some HTML entities that appear in the text that I remove as well like &3323;

Moreover, the reuters_corpus variable is a list I use later to validate if a query really exists in the text as defined by the inverted index.

```python
def remove_metadata(text):
    tags = metadata_tokenizer.tokenize(text)
    html_entities = html_entities_tokenizer.tokenize(text)
    metadata = tags + html_entities
    if len(metadata) > 0:
        for element in metadata:
            text = text.replace(element, '')
    return text
```

I then have a function that removes the metadata by finding all the elements that are part of the metadata as described in the regex tokenizers before, and just replaces them with nothing (removing them).

I now come to the loop for extracting info from the document:

```python
while True:
    document = input("Please enter document: ")
```

In my loop I have this general idea: The user inputs the name of a file.

If the input is a filename, it would look through the reuters folder in the same directory as the file like this:

```python
else:  # Let user select the file to be added to index
    document = './reuters21578/' + document
```

If the input is reuters_corpus, it would loop through all the sgm files in the reuters folder and extract the text from them like this. It also appends the files to reuters_corpus for query testing later. In the case of file 17 of the reuters files, I had to open it in binary as a workaround as I had trouble opening in UTF-8 encoding.

```python
elif document == "reuters_corpus":  # Get all sgm files from the reuters corpus and 
use that as index (Automatic index creation) (For testing queries)
for i in range(22):
    doc = f"./reuters21578/reut2-0{i:02d}.sgm"
    try:
        if i != 17:
            with open(doc, 'rt') as file:
                file = file.read()
        else:
            # Needed as file 17 gave me UnicodeDecodeError
            file = open(doc, mode="rb")
            file = file.read()
            file = str(file)
        articles = article_tokenizer.tokenize(file)
        reuters_corpus += articles
```

If the input is exit(), it breaks out of the loop.

```
if document == "exit()":  # Exit the loop
```

For either decision of the user, I go through some preprocessing of each article. I extract the ID and clean the content from metadata. I proceed to tokenize it and append a (docID, token) pair to a list F as described by the assignment. One idea I had in order to optimize memory is to write those pairs to a file so that they are stored in disk instead of memory. This is because depending on the number of articles memory would not be able to hold all the articles in memory at the same time and in order to read them again you could just read them from a disk but it would take more time and not necessary for this assignment.

```
for article in articles:
    ID = id_tokenizer.tokenize(article)[0]
    contents = content_tokenizer.tokenize(article)
    contents = ' '.join(contents)
    contents = remove_metadata(contents)
    tokens = word_tokenize(contents)
    for token in tokens:
        f.append((ID, token))
```

```
f.sort(key=lambda x: int(x[0]))
```

I now sort all the keys by converting the docID to an int and sorting with respect to that ID in ascending order. I thought also of having the IDs stored as int, but an int is 4 bytes and since the IDs of the files are around 3-5 chars that is around 3-5 bytes and so the difference is not that much. Maybe if the IDs were longer it would be worth it to save space by storing them as ints.

```
f = list(dict.fromkeys(f))
```

This code takes the list and removes all duplicates by converting first into a set and then reverting back to a list.

Now it is time to build the inverted index:

```
postings_list = {}
for pair in f:
    # Pair[0] is docID and pair[1] is word
    if pair[1] in postings_list:
        postings_list[pair[1]].append(pair[0])
    else:
        postings_list[pair[1]] = [pair[0]]
```

Here I decided to use a dict which is a hash table in Python. This way the lookup time for queries is constant and since the purpose is to find queries this would be the best way in my opinion to store it. I loop through each pair of docId and word and find if the word is already in the dict, for which I just append the docID. If not I create it.

```
def query_processor(postings, query):
    res = ''
    if query in postings:
        print(f"Query for word {query} retrieved successfully. The results 
are:\n{postings[query]}")
        res = postings[query]
    elif query == "exit()":
        print("Thank you for using the query processor. See you next time!")
    else:
        print("Query not found in postings list")
    return res
```

Once that is done I have a query processor function as described in requirements. This query takes as input the posting list and a query and outputs the documentIDs in which the query appears.

```
query = None
while query != "exit()":
    query = input("Please enter the word to search: (Type exit() to stop)")
    result = query_processor(postings_list, query)
```

I loop through it so that user can input what queries they want to know. The user would type exit() to break out of the loop.

I proceed to validate 3 queries I decided: exit, stocks and business. For validating I have a for loop that uses as helper the reuters_corpus variable I created before. This variable would be a list of each article as a string. So in total for the reuters corpus it would have around 21k elements.

```
query1 = "exit"
query2 = "business"
query3 = "stocks"

test_queries = {query1: query_processor(postings_list, query1),
                query2: query_processor(postings_list, query2),
                query3: query_processor(postings_list, query3)}

# Validate queries
for query in test_queries:
    ids = test_queries[query]
    for number in ids:
        newID = int(number) - 1
        if query in reuters_corpus[newID]:
            print(f"✓ {query} was FOUND in the article with id: {newID + 1}")
        else:
            print(f"×{query} was NOT FOUND in the article with id: {newID + 1}")
```
This is very helpful because I know that the article with ID 2000 is the element with index [2000 – 1] since indexes start from 0 but the docIds start from 1. Knowing this, I can go through each docID outputted of my queries and validate if it is true that the word appears in that specific article. Example output:

Query for word exit retrieved successfully. The results are:

['386', '583', '643', '2375', '2878', '3196', '3534', '7405', '8325', '8940', '8980', '11471', '16082', '16864', '16904', '20705']

✓ exit was FOUND in the article with id: 2375

✓ exit was FOUND in the article with id: 2878

... and so on

```
challenge_queries = ['copper', 'Samjens', 'Carmark', 'Bundesbank']
res_txt = open("challenge_queries_results.txt", "w")
for word in challenge_queries:
    result = query_processor(postings_list, word)
    res_txt.write(f"\nResults for query {word} are the following:\n{result}")

res_txt.close()
```
I run here through my query processor the challenge queries posted by the professor and output the result to a file. In my case I got some nice results.

Now it comes the lossy compression table to implement the table described in the book.

```
def find_stop_words(postings):
    sorted_postings = sorted([(len(v), k) for k, v in postings.items()], reverse=True)
    sorted_postings = sorted_postings[:150]
    common_stop_words = [x[1] for x in sorted_postings]
    return common_stop_words
```
I create a function to find the most common stop words in the dict. My logic is that the most common 150 words would be the 150 words with the longest list since they would appear in many docs. Here I loop through each key and look at the length of the values. For which I create another list that has as output the len(value) and the key (word). I sort this list in descending order to have the most common words first and return it.

```
# Distinct terms
unfiltered_terms = [key for key in postings_list]
no_numbers_terms = [word for word in unfiltered_terms if not word.isnumeric()]
case_folding_terms = {w.lower() for w in no_numbers_terms}  # set
stop_words_150 = find_stop_words(postings_list)
```

```
stop_words_30 = stop_words_150[:30]
remove_30_stop_words = {w for w in case_folding_terms if w not in stop_words_30}
remove_150_stop_words = {w for w in case_folding_terms if w not in stop_words_150}
porter = PorterStemmer()
porter_stemmed = {porter.stem(word) for word in remove_150_stop_words}
```

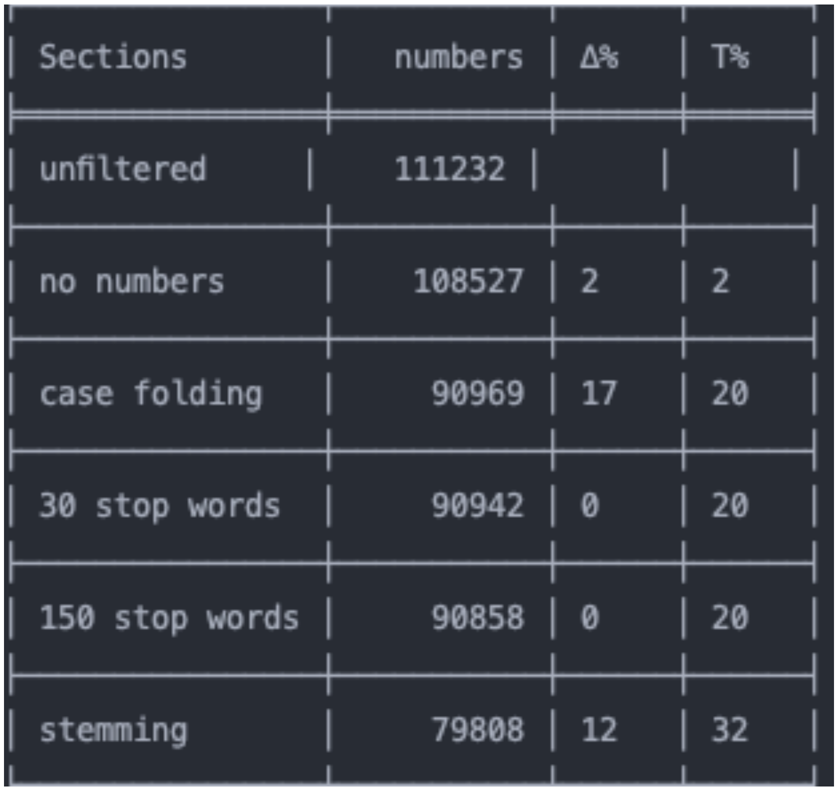
I create different variables to get all the elements of the table so for example. The unfiltered terms are all the keys in the dict since dict keys are unique. The numbers terms uses list comprehension to remove numbers from the vocab. Then case folding lowercases all the tokens. And the rest is more straightforward. In summary it uses list comprehension to check if the word has the condition we are looking for.

```
def percentage_diff(num1, num2):
    average = (num1 + num2) / 2
    diff = abs(num1 - num2)
    return int(diff / average * 100)
```

I create a function to determine the percentage difference between two numbers as it is used a lot in the table.

```
table_terms = {'Sections': ['unfiltered', 'no numbers', 'case folding', '30 stop 
words', '150 stop words', 'stemming'],
               'numbers': [len(unfiltered_terms), len(no_numbers_terms), 
len(case_folding_terms),
                           len(remove_30_stop_words),
                           len(remove_150_stop_words), len(porter_stemmed)],
               'Δ%': ['', percentage_diff(len(unfiltered_terms), 
len(no_numbers_terms)),
                      percentage_diff(len(no_numbers_terms), len(case_folding_terms)),
                      percentage_diff(len(case_folding_terms), 
len(remove_30_stop_words)),
                      percentage_diff(len(case_folding_terms), 
len(remove_150_stop_words)),
                      percentage_diff(len(remove_150_stop_words), 
len(porter_stemmed))],
               'T%': ['', percentage_diff(len(unfiltered_terms), 
len(no_numbers_terms)),
                      percentage_diff(len(unfiltered_terms), len(case_folding_terms)),
                      percentage_diff(len(unfiltered_terms), 
len(remove_30_stop_words)),
                      percentage_diff(len(unfiltered_terms), 
len(remove_150_stop_words)),
                      percentage_diff(len(unfiltered_terms), len(porter_stemmed))]}
print(tabulate(table_terms, headers="keys", tablefmt='fancy_grid', missingval='N/A'))
```

Here I use the tabulate module to create the table. I create a dict where each key is the header and the elements of the key are the contents of the column. In the end the table looks like this:



```
│ Sections        │   numbers │ Δ%   │ T%   │
╞═════════════════╪═══════════╪══════╪══════╡
│ unfiltered      │    111232 │      │      │
├─────────────────┼───────────┼──────┼──────┤
│ no numbers      │    108527 │ 2    │ 2    │
├─────────────────┼───────────┼──────┼──────┤
│ case folding    │     90969 │ 17   │ 20   │
├─────────────────┼───────────┼──────┼──────┤
│ 30 stop words   │     90942 │ 0    │ 20   │
├─────────────────┼───────────┼──────┼──────┤
│ 150 stop words  │     90858 │ 0    │ 20   │
├─────────────────┼───────────┼──────┼──────┤
│ stemming        │     79808 │ 12   │ 32   │
```

Which is in fact very similar to the results obtained from the book table in terms of percentage difference. The number of terms are different since they use 800,000 documents and we are only using around 21k. However, due to zipf's law we know that the most frequent terms are used most of the time. So it means that my vocab only lacks mostly rare words that are almost not used in the corpus. This is why the differences like cutting stop words reduces it in a similar manner because the stop words are pretty much the same between my postings and the book's. It is interesting to note that there is 0 in the stop words removal percentage since it is just removing 150 elements from a list of 90k.

```python
print(tabulate(table_terms, headers="keys", tablefmt='fancy_grid', missingval='N/A'))
#%%
def len_posts(postings):
    length = 0
    for key in postings:
        length += len(postings[key])
    length += len(postings)
    return length
```

I use a function to calculate length of keys and values of postings lists

```python

unfiltered_postings = len_posts(postings_list)

# Remove numbers
```

```
for word in unfiltered_terms:
    if word.isnumeric():
        del postings_list[word]

no_numbers_postings = len_posts(postings_list)
```
I remove all numbers

```
# Casefold
for word in no_numbers_terms:
    if word.lower() in postings_list:
        if postings_list[word.lower()] != postings_list[word]:
            # Find union of lists
            union = list(set().union(postings_list[word.lower()],
postings_list[word]))
            postings_list[word.lower()] = union
            del postings_list[word]
    else:
        postings_list[word.lower()] = postings_list[word]
        del postings_list[word]

case_folding_postings = len_posts(postings_list)
```
I casefold by this algorithm. I look if the lowercase version of the word is in dict, for which I check if I am looking if the postings lists of the current word I am looking and the lowercase are similar. If not, it means that the word I am looking is the uppercase version and so I add that to the lowercase version of the postings by unioning the lists and removing duplicates. If the lowercase version is not found I move all postings to the lowercase version instead.

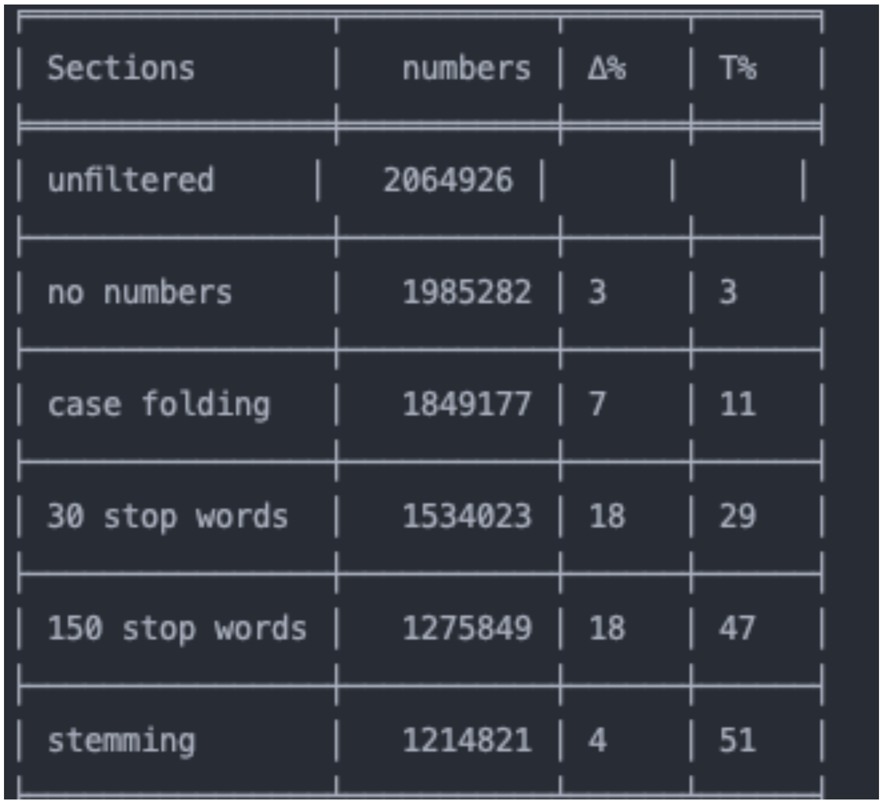
```
# Remove 30 stop words
for word in stop_words_30:
    if word in postings_list:
        del postings_list[word]

remove_30_stop_words_postings = len_posts(postings_list)
# Remove 150 stop words
for word in stop_words_150:
    if word in postings_list:
        del postings_list[word]

remove_150_stop_words_postings = len_posts(postings_list)
```
I remove stop words here by looping and deleting from the postings lists

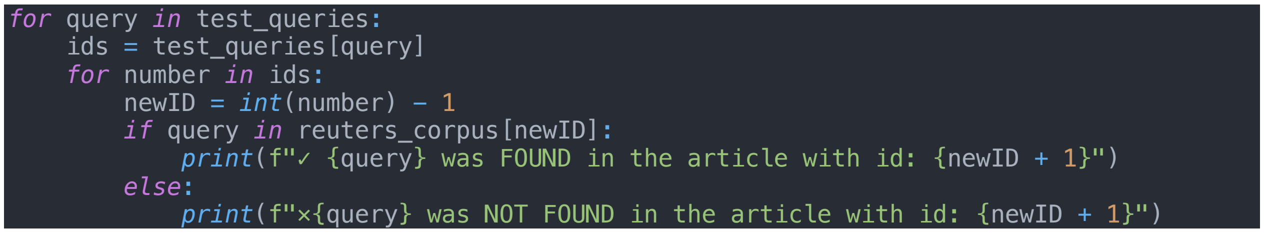
```
# Stemming
for word in remove_150_stop_words:
    if porter.stem(word) in postings_list:
        if postings_list[porter.stem(word)] != postings_list[word]:
            # Find union of lists
            union = list(set().union(postings_list[porter.stem(word)],
postings_list[word]))
            postings_list[porter.stem(word)] = union
            del postings_list[word]
    else:
        postings_list[porter.stem(word)] = postings_list[word]
        del postings_list[word]
stem_postings = len_posts(postings_list)
```

And I apply porter stemmer with a similar algorithm to the one casefolding. It is similar except that it applies porter stemmer.



Sections	numbers	Δ%	T%
unfiltered	2064926		
no numbers	1985282	3	3
case folding	1849177	7	11
30 stop words	1534023	18	29
150 stop words	1275849	18	47
stemming	1214821	4	51

The table I get is very similar. In this case the biggest difference for the postings list is that the stop words percentage is different since the number of documents is longer in the book. Since there are so many stop words the postings lists of those stop words grows exponentially and so the size reduce of removing those big lists is more with a bigger document base. Apart from that the results in the end cumulative are similar so it makes sense.



```
for query in test_queries:
    ids = test_queries[query]
    for number in ids:
        newID = int(number) - 1
        if query in reuters_corpus[newID]:
            print(f"✓ {query} was FOUND in the article with id: {newID + 1}")
        else:
            print(f"×{query} was NOT FOUND in the article with id: {newID + 1}")
```

In my case since my queries were lowercase there was not significant difference between the query processes between the first index and this one.