Gabriel Martinica

ID: 40120255

```python
from nltk.tokenize import RegexpTokenizer
from nltk import word_tokenize
import time
from math import log
```

First, I import the libraries I need to do the programming. The nltk ones to tokenize and get docs. The time one is used to time the indexers SPIMI and naïve to find the differences between them. The log math is used for the BM25 calculation. There are other libraries I wanted to explore if given more time, for example using Collections counter to count occurrences of element in list in faster time than native python implementation, or storing using numpy arrays which are more memory efficient.

```python
content_tokenizer = RegexpTokenizer('<TEXT.*?>(.*?)</TEXT>')
article_tokenizer = RegexpTokenizer('<REUTERS(.*?)</REUTERS>')
id_tokenizer = RegexpTokenizer('NEWID="(.*?)"')
metadata_tokenizer = RegexpTokenizer('<.*?>')
html_entities_tokenizer = RegexpTokenizer('^&.*?;')
token_count = 0
sub_corpus = []
```

Here I declare some variables that I will be using to extract the information from the corpus. These are used to tokenize and extract the info from the articles. The sub_corpus is a variable I define to get a sub_corpus of the 10,000 tokens for the indexing comparison.

```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


def remove_punctuation(tokens):
    punctuation_list = [*".,:;-<>{}()[]~`&*?"]
    double_symbols = ['""', "''", "``", "...", "--", "-", ","]
    punctuation_list += double_symbols
    for t in tokens:
        if t in punctuation_list:
            tokens.remove(t)
    return tokens
```

These are some functions I use to remove both metadata and punctuation.

```
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)

        for article in articles:
            ID = id_tokenizer.tokenize(article)[0]  # Get ID
            contents = content_tokenizer.tokenize(article)  # Get content inside
<TEXT> tags
            contents = ' '.join(contents)
            contents = remove_metadata(contents)
            # Tokenize article content, need to get title and body
            tokens = word_tokenize(contents)
            tokens = remove_punctuation(tokens)
            # Add ID to sub_corpus element
            sub_corpus.append([ID])
            article_tokens = []
            for token in tokens:
                token_count += 1
                if token_count < 10000:
                    article_tokens.append(token)
                else:
                    sub_corpus[int(ID) - 1].append(article_tokens)
                    raise StopIteration
            sub_corpus[int(ID) - 1].append(article_tokens)
    except IOError:
        print("Error: File does not exist")
    except StopIteration:
        print("Reached 10,000 tokens")
        break
```

This is a loop that is used to get a specific amount of tokens from the documents, in this case I loop through all the documents similar as I did in assignment 2 and start tokenizing. The difference here is that I have a count that goes up to 10,000 and stops the iteration. In theory this could be modified to accommodate any other amount of tokens needed. The result is a sub_corpus of the form (ID,tokens). In my case 10,000 tokens where reached when it got to docID 73.

```
f = []

# Start timer
start_time = time.time()

# Get doc,ID pairs

for i in range(len(sub_corpus)):
    for token in sub_corpus[i][1]:
        f.append((sub_corpus[i][0], token))
```

```python
# Sort and remove duplicates
f.sort(key=lambda x: int(x[0]))
f = list(dict.fromkeys(f))
index_naive = {}
for pair in f:
    # Pair[0] is docID and pair[1] is word
    if pair[1] in index_naive:
        index_naive[pair[1]].append(pair[0])
    else:
        index_naive[pair[1]] = [pair[0]]
print(index_naive)
# End timer
print(f"Naive indexer took: {(time.time() - start_time)} seconds")
```

This is the code for the naïve indexer. In this case I start the timer just for indexing. The input given to the indexer is the subcorpus I previously created. I first start by getting ID,word pairs from the subcorpus. Then I sort and remove duplicates and finally I start creating my index from the ID,word pairs. In my case, this was very fast as there were not many tokens and it took 0.0221 seconds. As I will say later, this is okay for such a small collection but it is inefficient since we have to store ID,word pairs and sort all of them. It works good as the simple method but for large collections other methods like SPIMI work better.

```python
index_spimi = {}

# Start timer
start_time = time.time()
token_count = 0

for article in sub_corpus:
    ID = article[0]
    for token in article[1]:
        if token in index_spimi:
            if not ID in index_spimi[token]:
                index_spimi[token].append(ID)
        else:
            index_spimi[token] = [ID]
# End timer
print(f"SPIMI indexer took: {(time.time() - start_time)} seconds")
```

Here I have the SPIMI code. I loop through each article and instead of creating id,word pairs I start creating the index straight away. In this case I know which are the ID and the term, so I start putting it into a dictionary in Python in order to create my index. In my case this took 0.00706 which is 3 times faster than naïve indexer and it makes sense, since we are reading straight away and not wasting time doing pairs and sorting as it is sorted by default, we are reading the docs from the beginning to the end.

```python
reuters_corpus = []
index_corpus = {}

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)
        for article in articles:
            ID = id_tokenizer.tokenize(article)[0]  # Get ID
            contents = content_tokenizer.tokenize(article)  # Get content inside
<TEXT> tags
            contents = ' '.join(contents)
            contents = remove_metadata(contents)
            tokens = word_tokenize(contents)
            reuters_corpus.append((ID, tokens))
    except IOError:
        print("Error: File does not exist")
```

In order to work on the whole corpus and in the subproject 2, I create a corpus of the whole reuters corpus and store it in the same way I did before for the subcorpus. This IO takes some time like 30 seconds but the results is worth it since we finish dealing with file reading operations.

```
# Start timer
start_time = time.time()
for article in reuters_corpus:
    ID = article[0]
    tokens = set(article[1])
    for token in tokens:
        if token in index_corpus:
            index_corpus[token].append(ID)
        else:
            index_corpus[token] = [ID]
# End timer
print(f"Whole corpus indexer took: {(time.time() - start_time)} seconds")
```

This part is for indexing for the whole corpus. I take a similar approach as in SPIMI and the result is an indexing of 1.3 seconds which is very fast. If I did it naïve indexing It would probably be 4 seconds so scalability wise this is better. Other thing to notice is that since I am not using any compression techniques it means that it might probably take longer as well. For example if we did some pre processing like lowercasing, porter stemmer it would be faster since it reduces the number of terms. Removing stop words would be the greatest time saver. Still, it was not that slow for me.

```
inverted_index = {}
doc_len_list = []
for doc in reuters_corpus:
    ID = doc[0]
    doc_len_list.append(len(doc[1]))  # Appends doc length for use in BM25
    tokens = remove_punctuation(doc[1])
    no_duplicates = set(tokens)
    for token in no_duplicates:
        freq = tokens.count(token)
        if token in inverted_index:
            if not ID in inverted_index[token]:
                inverted_index[token][ID] = freq
        else:
            inverted_index[token] = {ID: freq}
```

In order to start subproject 2 I create an inverted index from the reuters corpus in where I have the term frequency as well, as it would be needed for BM25. In this case I have a nested dict where the first layer is the term, then it leads to the postings list which is a dict as well. Each doc in the list is a key that has as value the term frequency for that term so it looks like:

{Term -> {ID -> Term frequency}}

I also append the doc len while indexing to a list in order to determine the average doc length used for BM25. This can also help me get the doc len of any docID since I am reading in order and the docID determines the index of the element in the list. So len of doc 1 is the first element and so on.

```
avg_doc_len = int(sum(doc_len_list) / len(doc_len_list))
total_doc_num = len(reuters_corpus)


def compute_bm25(tf, df, doc_len, k, b):
    log_part = log(total_doc_num / df)
    numerator = (k + 1) * tf
    denominator = (k * ((1 - b) + b * (doc_len / avg_doc_len))) + tf
    score = log_part * (numerator / denominator)
    return score
```

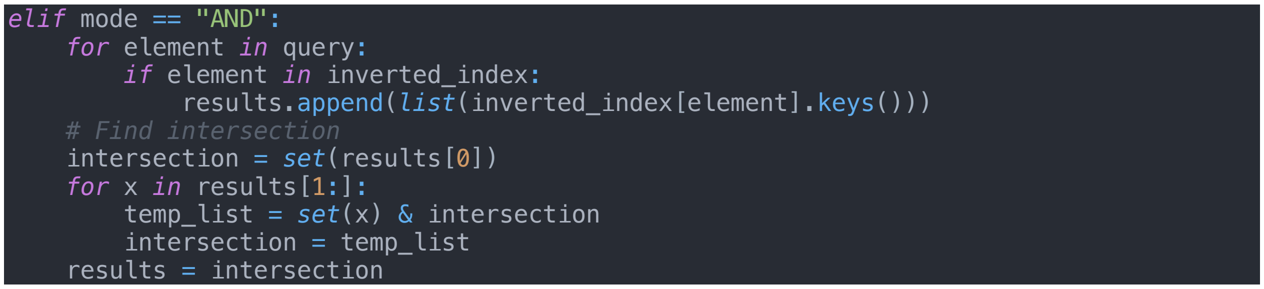
Here I define the average doc len and I the total number of documents which are used for BM25. I create a function that is used to calculate it.

```
# Modes are bm25, AND, OR
def process_query(query, mode):
    query = set(word_tokenize(query))
    results = []
    # OR query
    if mode == "OR":
        results = {}
        for element in query:
            if element in inverted_index:
                for ID, tf in inverted_index[element].items():
                    if ID in results:
                        results[ID] += tf
                    else:
                        results[ID] = tf
        results = sorted(results.items(), key=lambda item: item[1], reverse=True)
        results = [x[0] for x in results]
```

I start with the function process_query which has three modes: AND, OR, BM25.

What it does is that first we tokenize the query from the user and remove duplicate words (because some queries could have duplicates and it would be inefficient to search something twice) Ex: my friend does not like my mom

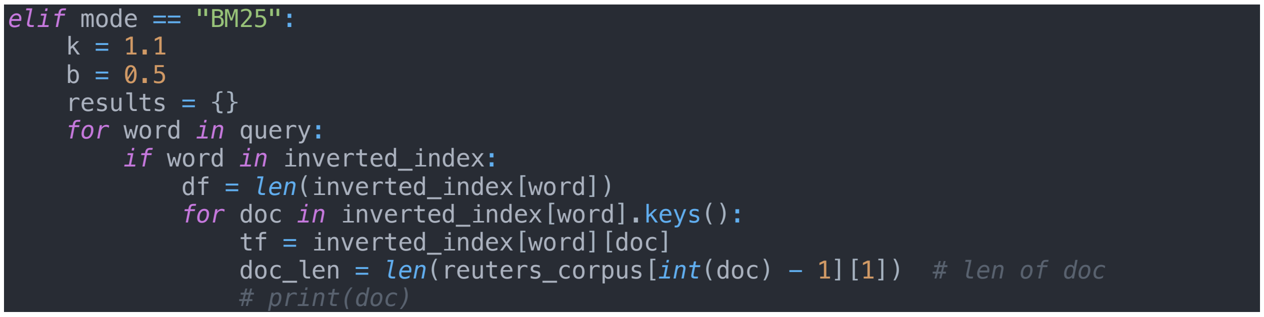
We can start with OR. It starts looking for each word in the query, if it is in the index, then it looks for the postings list and determines the term frequency. Since we need to rank results according to term frequency I first create a dict which has as key the docID where the queries are present and as values the collective term frequency. That is the sum of the term frequencies for each term in a specific document. Then I convert this list to a dict and sort it in order to get the documents with the highest term frequency first. The downside of this is that it can favor stop words or very common words for queries. For example, if I search something like “the wolf” it would give me first documents that have a lot of “the” and not necessarily a lot of “wolf”. While in this case wolf is more important. This could be fixed by preprocessing the query as well and removing those stop words. Could try if given more time.



```
elif mode == "AND":
    for element in query:
        if element in inverted_index:
            results.append(list(inverted_index[element].keys()))
    # Find intersection
    intersection = set(results[0])
    for x in results[1:]:
        temp_list = set(x) & intersection
        intersection = temp_list
    results = intersection
```

For the AND I do intersection of the postings list. Since it is unranked retrieval the way it works is that I convert them into sets and intersect them. The results go back to the user. I find this the least intelligent query mode as it just looks for docs containing all the words, not good for search engines in my opinion.

BM25 solves most of these issues:



```
elif mode == "BM25":
    k = 1.1
    b = 0.5
    results = {}
    for word in query:
        if word in inverted_index:
            df = len(inverted_index[word])
            for doc in inverted_index[word].keys():
                tf = inverted_index[word][doc]
                doc_len = len(reuters_corpus[int(doc) - 1][1])  # len of doc
                # print(doc)
                # print(tf)
                # print(df)
                # print(doc_len)
                score = compute_bm25(tf, df, doc_len, k, b)
                if word in results:
                    results[doc] += score
                else:
                    results[doc] = score
    results = sorted(results.items(), key=lambda item: item[1], reverse=True)
    results = [x[0] for x in results]
return results
```

For BM25 I get the document length, the term frequency, the doc frequency and the average doc length from all the previous steps I have done. In this case the k and b values were interesting to play around with. I notice that my k terms determined the way that the frequency of a word affected the score. A higher k meant that the more frequent the term was the higher the score for that document is. This is not good to have it high since at high frequencies the word starts losing value and it is starting to be considered less important overall (like stop words).

For b, a higher b affects the length of the document. I noticed in my results that the higher b means that if the document was shorter it was ranked higher.

Overall, my BM25 ranked documents highest if they were shorter than average, and had more frequent instances of that word.

I do some queries like test queries:

For samjens in single term:

```
test_query_a = "Samjens"
```

['18071', '17863', '19419', '17837']

It gave me the same docs as the previous assignment but in different order.

```
test_query_b = "Smith likes play football"
```

Top results are:

['1294', '6918', '15592', '3699', '19331', '8927', '13197', '20457', '21090', '11971', '15589', '13908']

In this case the Smith word takes a lot of importance, and the top documents are short and contain a lot of the word Smith or football in the body. Goes in accordance with what I mentioned before about it.

```
test_query_c = "Hong Kong investment firm"
```

This is the AND query. It gives me documents which contains all those words which are:

{'960', '16872', '12397', '12610', '9717', '14888', '8085', '10291', '12427', '3933', '125', '11782', '9206', '223'}

Definitely more than I expected

```
test_query_d = "Concordia university"
```

Last one is this one to test

['9479', '10134', '15075', '153', '340', '8729', '8763', '10179', '10199', '10346', '12431', '14635', '17428', '18448', '19594']

Only the last document has the word Concordia, so for the OR it just matters if one word appears a lot in one document. SO in this case in the first docID 9479 it has university the most

Last queries in digestible format: Top 20 results

Query: Democrats' welfare and healthcare reform policies

['7467', '7806', '18161', '3113', '18159', '7549', '1857', '11955', '12455', '1969', '7025', '21455', '15140', '17242', '12415', '4940', '6600', '15224', '9549', '17217']

Digestible format

Query: Drug company bankruptcies

['16771', '8209', '7125', '12808', '12461', '2679', '12242', '18072', '14165', '4435', '11553', '3776', '7506', '9546', '3756', '8789', '18059', '15257', '9085', '3151']

Digestible format

Query: George Bush

['16824', '2711', '4008', '16780', '4853', '2766', '10400', '6564', '5459', '2733', '255', '10682', '15284', '16115', '21393', '5334', '17150', '7469', '16229', '17647']