Demo

COMP 479 Deliverable 3

November 7th 2022

Christian Pangia-Henneveld

40034040

DEMO

In the demo I will go over the creation of my 3 main classes and also the logical flow of the program which happens on any of my runs. My indexers class contains all the code to create indexers. This includes the SPIMI and the indexer from project 2.

Indexers: Constructor

```
class Indexers:
    tarxxan
    def __init__(self, timed="naive"):
        if timed == "naive":
            self.naive, self.document_lengths, self.total_doc_length, self.df_term_data = self.naive_indexer()
        elif timed == "p2":
            self.p2, self.start_time = self.timed_indexer_p2()
        else:
            self.SPIMI, self.start_time = self.timed_indexer_spimi()
```

The constructor will allow selection of which indexer is created based on the argument that is passed to it. Both the SPIMI and project 2 indexer use the same logic to start off the indexer. The only changes are the way in which they process the tokens.

Indexers: timed_indexer_spimi, timed_indexer_p2

```
def timed_indexer_spimi(self):
    timed_spimi = {}
    fileList = get_files()
    for file in fileList:
        rt = raw_text(file)
        tokens = self.create_10k_tokens(rt)
        start_time = time.perf_counter()
        timed_spimi = self.create_timed_spimi(tokens)
        break
    return timed_spimi, start_time

tarxxan *
def timed_indexer_p2(self):
    f = []
    fileList = get_files()
    for file in fileList:
        rt = raw_text(file)
        tokens = self.create_10k_tokens(rt)
        start_time = time.perf_counter()
        f = self.create_timed_p2(tokens)
    no_dupes = self.remove_dup(f)
    postings = self.postings_list(no_dupes)
    return postings, start_time
```

The SPIMI will use a very simple and efficient way to create the dictionary.

Indexers: create_timed_spimi

```
def create_timed_spimi(self, token_list):
    spimi = {}
    for articles in token_list:
        for words in articles[1:]:
            if words in spimi and spimi[words][-1] != articles[0]:
                spimi[words].append(articles[0])
            else:
                spimi[words] = [articles[0]]
    return spimi
```

It will take the token list which in this case is set up as [[articleID, words,words"], [articleID, words,words"] ,etc..]. Which is why we use articles[0] to add to the posting list. The only thing that is checked is whether the word is already a key in the dictionary and if the last index in the posting list is equal to the current id in this case we wouldnt add and this filters out duplicates. The timed p2 version puts these article id and words into individual pairs and then passes that to the remove_dup function to be transformed into a dictionary.

The final function postings list will remove any duplicates in the posting list. The main function in my Indexers class is naive indexer. I use this to create the indexer as well as any auxiliary dictionaries we might need when calculating rank for BM25.

Indexers: naive_indexer

```
def naive_indexer(self):
    naive = {}
    new = {}
    document_lengths = []
    total_doc_length = 0
    fileList = get_files()
    for file in fileList:
        new = naive
        rt = raw_text(file)
        naive, document_length, total_doc = self.create_indexer(rt, new)
        document_lengths.extend(document_length)
        total_doc_length += total_doc
    df_term_data = self.term_document_frequency(naive)
    naive = self.remove_posting_dup(naive)
    return naive, document_lengths, total_doc_length, df_term_data
```

This Is responsible for the flow and creation of our SPIMI indexer which we will use in all of subproject 2.

Indexers: create_indexer

```
def create_indexer(self, rt, naive_dict):
    total_doc_length = 0
    document_lengths = []
    document_length = 0
    remove = ["lt", ",", ".", ">", "<", "-", ";", ":", "&", "#", "?",
    for reuters in rt:
        for lists in reuters:
            (id, title, body) = lists
            if title:
                words = word_tokenize(title[0])
                document_length += len(words)
                for word in words:
                    if word not in remove:
                        if word not in naive_dict:
                            naive_dict[word] = [id]
                        else:
                            naive_dict[word].append(id)
            if body:...
            document_lengths.append(document_length)
            total_doc_length += document_length
            document_length = 0
    return naive_dict, document_lengths, total_doc_length,
```

Create indexer is the first function which hasnt been discussed yet. As you can expect it creates the indexer but also allows for tracking of each document's length and total document size in this case a document is a file. This gets passed back to the calling function(naive_indexer) and adds it to the true document length total. As mentioned in the report I decided to keep track of the length as the amount of tokens, not the length of the string which will be tokenized. I think this way is a little more accurate as it reflects the true amount of words we are considering in corpus.

Indexers: term_document_frequency

```
def term_document_frequency(self, data):
    tdF = {}
    for key in data:
        tdF[key] = []
        """..."""
        tdF[key].extend([len(list(group)) for key, group in groupby(data[key])])
    return tdF
```

The function will grab the completed index without any duplicate removal in the posting list and count the frequency of the word in each document it appears in. Since my code processes articles separately in a list if a word appears more than once it will be repeated together in the posting list. This allows my last line of code to group the numbers and return the frequency of the word in that document. Since when we remove duplicates it will be only one value. Position 0 for word “the” in both postings list (tdf and indexer) will be for the same document. More about the code is discussed in the multiline comment within the function. The last function call in create_indexer is similar to the one used earlier on to remove duplicates in the posting list.

```
def tokenize_text_10000(raw_list, count):
    naive_dict = {}
    remove = ["lt", ",", ".", ">", "<", "-", ";", ":", "&", "#", "?", "!"]
    for reuters in raw_list:
        for lists in reuters:
            (id, title, body) = lists
            if title:
                words = word_tokenize(title[0])
                for word in words:
                    if word not in remove:
                        if word not in naive_dict:
                            naive_dict[word] = [id]
                        elif word in naive_dict:
                            if id not in naive_dict[word]:
                                naive_dict[word].append(id)
                            else:
                                pass
                        count += 1
                        if count == 10000:
                            return naive_dict, -1
            else:
                pass

            if body:
                words = word_tokenize(body[0])
                for word in words:
                    if word not in remove:
                        if word not in naive_dict:
                            naive_dict[word] = [id]
                        elif word in naive_dict:
                            if id not in naive_dict[word]:
                                naive_dict[word].append(id)
                            else:
                                pass
                        count += 1
                        if count == 10000:
                            return naive_dict, -1
```

First function we must take into consideration is our SPIMI inspired indexer. As you can see here we would receive a list of tuples which are tokenized and then process then immediately. The token is split up into id,title and body so I just append the id to the postings list of the term whenever it is not in the dictionary. If it's already there i do

nothing, this is not the case in my other indexer where I create the data for document length and total amount of terms. This also checks to ensure the count only goes up to 10000. When it reaches that number it will exit and return the dictionary(indexer). To avoid repeating screenshots and code the main difference between this indexer and the full indexer is that it removes anything to do with the count. As you can see here everything goes directly into the dictionary and posting list and does not incur additional steps to treat them as we did in project 2. The removal of punctuation happens at the beginning before inserting in the dictionary. This was not originally the case and saved me 2+ seconds on my runtime when testing over the 10000 term-DocID pairs which also made me change code in project 2 so they would be more similar.

Subproject 2

BM25

BM25: create_indexer

```
def create_indexer():
    data = load_indexer()
    df_term_data = term_document_frequency(data)
    data = remove_posting_dup(data)
    return data, df_term_data
```

Subproject 2 will be split into 2 sections to facilitate code reviewing in the demo. We will continue with BM25 and Query as all explanations for document length, total document length, etc.. was given above when explaining the indexers class

BM25: Constructor

```
def __init__(self, indexer):
    self.indexer = indexer
    self.k1 = 1.2
    self.b = 1
    self.words_idf = []
    self.words_tftd = []
    self.document_rsv = {}
```

BM25 is a class which contains everything necessary to calculate the weighting of a document for a term or a query. Almost every part of the calculation has its own instance variable within the class

BM25: Calculation

```
def calculation(self, query):
    query = word_tokenize(query)
    stopword_list = set(stopwords.words('english'))

    for word in query:
        if word in stopword_list:
            query.remove(word)
    self.calculate_idf_tftd(query)
    self.create_rsv()
    count = 0
    # dictionary corresponding to the word in the query
    for tftd in self.words_tftd:
        # dictionary corresponding to the document
        for document in tftd:
            numerator = (self.k1 + 1) * tftd[document]
            ld = self.indexer.document_lengths[document - 1]
            l_avg = self.indexer.total_doc_length / len(self.indexer.document_lengths)
            denominator = (self.k1 * ((1 - self.b) + self.b * (ld / l_avg) + tftd[document]))
            rsvd = self.words_idf[count] * (numerator / denominator)
            if self.document_rsv[document] == 0:
                self.document_rsv[document] = rsvd
            else:
                self.document_rsv[document] += rsvd
        count += 1
    self.document_rsv = sorted(self.document_rsv.items(), key=lambda lst: lst[1], reverse=True)
```

Calling calculation will execute the BM25 algo on the query. Note here as i mentioned i remove any stop words in order to preserve the ranking of the documents should any of them be included in the query.

BM25: Calculate_idf_tftd

```
def calculate_idf_tftd(self, query):
    words_idf = []
    words_tftd = []
    for word in query:
        word_idf = self.idf(word)
        words_idf.append(word_idf)
        word_tftd = self.tftd(word)
        words_tftd.append(word_tftd)
    self.words_idf = words_idf
    self.words_tftd = words_tftd
```

This happens in two steps for each word. The idf is calculated and appended to a list and then the TFtd. After the process is done we assign the class instance variable to the result of those two functions for the query.

BM25: idf

```
def idf(self, word):
    N = len(self.indexer.document_lengths)
    dft = len(self.indexer.naive[word])
    return log(N / dft)
```

This function takes the Total amount of documents N and the length of the posting list of the term and takes the log of the quotient. This happens per term and not per document like the TFtd.

BM25: tftd

```
def tftd(self, word):
    tftd = {}
    count = 0
    for val in self.indexer.naive[word]:
        tftd[val] = self.indexer.df_term_data[word][count]
        count += 1
    return tftd
```

TFtd is slightly more complicated. I get every posting list for the word and then have a new dictionary with the id as the key. The value for that key will be the frequency that it shows up in that document. Since my document frequency data is stored in a list that has preserved the order of appearance in documents.we increment the count each time to get the frequency for the next document that the word appears in. This results in one TFtd dictionary per word in the query.

BM25: create_rsv

```
def create_rsv(self):
    document_rsv = {}
    for keys in self.words_tftd:
        for key in keys:
            document_rsv[key] = 0
    self.document_rsv = document_rsv
```

The create_rsv function which follows from the two calculating functions creates a dictionary with keys belonging to each document. This allows us to separate and get a value for each document. The value of all words will later be combined in order to determine overall document relevancy/ranking.

```
for tftd in self.words_tftd:
    # dictionary corresponding to the document
    for document in tftd:
        numerator = (self.k1 + 1) * tftd[document]
        ld = self.indexer.document_lengths[document - 1]
        l_avg = self.indexer.total_doc_length / len(self.indexer.document_lengths)
        denominator = (self.k1 * ((1 - self.b) + self.b * (ld / l_avg) + tftd[document]))
        rsvd = self.words_idf[count] * (numerator / denominator)
        if self.document_rsv[document] == 0:
            self.document_rsv[document] = rsvd
        else:
            self.document_rsv[document] += rsvd
    count += 1
self.document_rsv = sorted(self.document_rsv.items(), key=lambda lst: lst[1], reverse=True)
```

The final step in the calculation combines all these lists and numbers in the same way specified by the book. I dont believing any additional explanation is needed but ill give some reasoning behind the code. As mentioned before we have a dictionary for each word. In each of those words we grab the document ID which is the key of the dictionary. With this key we can use it to access the proper TFtd as well as assign it to the final calculation to the proper key-value pair in docuemnt_rsv. Finding the proper document in document length must always subtract 1 due to 0 based indexing of lists. The last line in the function will sort the dictionary items using a lambda function. It creates a list of tuples which can then be sorted off of their score. The Reverse=true just flips the original order so we are left with things in descending instead of ascending order.

QUERY

Query: query_operation

```
def query_operation(self, query, operation):
    list_of_dictionary = []
    stopword_list = set(stopwords.words('english'))
    query_tokenized = query.split(" ")
    for word in query_tokenized:
        if word not in stopword_list:
            if word in self.indexer.naive:
                list_of_dictionary.append(self.indexer.naive[word])
        self.result = sorted(list_of_dictionary, key=len)
    if operation == "and":
        return self.intersect()
    else:
        return self.union()
```

My third and final class is my query class. It contains one main function which splits itself into two smaller functions handling both the “and” and “or” query. The first thing I do is remove stop words again to shorten the lists we deal with. We then order the posting list belonging to every word in descending order so when we take the intersection we are always doing so with the smallest list possible thus increasing speed and efficiency.

QUERY:Union

```
def union(self):
    or_list = []
    or_dict = {}
    result = []
    for i in range(len(self.result)):
        or_list.extend(self.result[i])

    for i in or_list:
        if i in or_dict:
            or_dict[i] += 1
        else:
            or_dict[i] = 1

    or_list = sorted(or_dict.items(), key=lambda lst: lst[1], reverse=True)

    for i in or_list:
        result.append(i[0])

    self.result_or = result
```

Union gets used for the “or” operation. We put all the lists into one large list and then create a dictionary which allows us to get the frequency of each document and leave that as the value of the dictionary. For example if document 1 shows up 2 times for the query Hello World then we would have a key of 1 and a value of 2. The dictionary is then transformed as I did with the ranking document except now the new list is sorted by frequency instead of score. The key is appended into a new list before being saved in the object.

We now have most of what's needed in order to make the probabilistic search engine. We must create a new indexer in order to get each document’s length as well the total

Documented Sample runs

Sample runs are given for the queries posted at the bottom of the assignment. Not much discussion will be done in the demo or the report besides the results that are printed into the file. This shows the logic on how I go about producing the results of the sample runs.

```
query = input("What is your query?")
BM25 = BM25(indexer)
BM25.calculation(query)

print("Document ranked by BM25 in descending order")
with open(query + ' ranked.txt', 'w') as f:
    f.write(str(BM25.document_rsv))

query = input("What is your query?")
Query = Query(indexer)

Query.query_operation(query, "and")
Query.query_operation(query, "or")

with open(query + '.txt', 'w') as convert_file:
    convert_file.write("And\n")
    convert_file.writelines(str(Query.result_and))
    convert_file.write("\nOr\n")
    convert_file.writelines(str(Query.result_or))
```