Project 3: Demo

Christopher Di Giacomo

40133600

10 November 2023

COMP479 2232 D

Dr. Sabine Bergler

Table of Contents

Environment/Setup

Directories:

- **/reports:** This directory contains the report and demo for this project
- **/src:** This directory contains the *subProject1.py* and *subProject2.py* files
- **/indexes:** This directory contains the indexes in JSON format. It is (re)generated at runtime for either *.py* files.
- **/Corpus:** This directory contains all of the reuters collection *.sgm* files

Notes:

- The current working directory must be the parent directory of the submission folder. Although the /reports subdirectory will be present in the project folder structure, within the text editor of choice, this will not interfere with any computation.
- If using an extension based IDE like VS Code, you must run the pipeline.py file as a Python file, and not via any code runner extensions.

Subproject1: First-Run Walkthrough

Running *subProject1.py* will trigger the *main()* function call and result in the following sequence of operations until terminal output.

Main

The main function below executes, retrieving the collection, building both indexes, and calling *getStats()* to display the performance data.

```
def main():
    """
    Main function to execute the token retrieval, index building, and statistics generation process.
    It retrieves tokens, builds indexes using both naive and SPIMI approaches, and then prints statistics.
    """

    # Measure time taken to create token streams
    start = time.time()
    tokenStream = getTokens()
    end = time.time()
    tokensTime = round(end - start, 3)

    # Build SPIMI and Naive indexes
    spimiIndex, spimiTime = spimi(tokenStream, 10000)
    naiveIndex, naiveTime = naive(tokenStream, 10000)

    # Generate and print statistics
    getStats(naiveIndex, naiveTime, spimiIndex, spimiTime, tokensTime)
```

getTokens()

The first function called in the main is the *getTokens()* function, displayed below. As in the previous report, it iterates through all .sgm files of the collection and calls the tokenizer() function on each document. The resulting tokenized collection is returned, in variable tokensList.

```python
def getTokens():
    """
    Retrieves and tokenizes documents from a specified corpus. Each .sgm file in the
    corpus is parsed, and documents within are tokenized. Tokens are associated with their
    document ID.

    Returns:
        list: A list where each element represents the tokens of a document, indexed by document ID.
    """

    tokensList = []
    # Inform about the start of token retrieval process
    print(f'Retrieving tokens ...')
    # Loop through each file in the corpus
    for file_number in range(22):
        with open(f'./Corpus/reut2-{str(file_number).zfill(3)}.sgm', 'r', encoding='windows-1252') as f:
            # Parse the file content using BeautifulSoup
            soup = BeautifulSoup(f, 'html.parser')
            documents = soup.find_all('text')

            for document in documents:
                doc_type = document.get('type')
                text_parts = []

                # Different handling for 'BRIEF' type documents
                if doc_type == 'BRIEF':
                    title = document.title.get_text() if document.title else ""
                    text_parts.append(title)
                else:
                    title = document.title.get_text() if document.title else ""
                    body = document.body.get_text() if document.body else ""
                    text_parts.append(title)
                    text_parts.append(body)

                # Combine title and body for tokenization
                full_text = ' '.join(text_parts)
                tokens = tokenizer(full_text)
                tokensList.append(tokens)

    # Indicate completion of token retrieval
    print(f'Tokens retrieved!')
    return tokensList
```

tokenizer()

The tokenizer() function is displayed below. As in project 2, it returns the tokenized inputted string with respect to a regex pattern, using NLTK's regexp_tokenize().

```python
def tokenizer(text):
    """
    This function takes in a text and returns a list of tokens. It uses a regular expression to
    handle acronyms, abbreviations, and general word patterns including words with hyphens or apostrophes.

    Args:
        text (str): The text to be tokenized.

    Returns:
        list: A list of tokens extracted from the input text.
    """

    # Pattern captures abbreviations/acronyms (e.g., U.S.A.) and general word forms.
    pattern = r'\b(?:[A-Z]{1,2}\.)*[A-Z]{1,2}\.?\b|\b\w+(?:[-\']\w+)*\b'
    return nltk.regexp_tokenize(text, pattern)
```

As mentioned, once the getTokens() function has tokenized all .sgm files using tokenizer(), it returns a list of tokenized documents which are in the form of lists of tokens. This is returned to the main method, which has calculated the elapsed time for this phase of subproject 1. Next, it begins by constructing the spimiIndex, whose implementation can be found below.

spimi()

The function builds the inverted index in a single pass using the tokensList, returned by getTokens(). It also has an optional parameter for subproject 1, which halts the construction

```python
def spimi(tokensList, testCount=None):
    """
    Builds an inverted index using the Single-Pass In-Memory Indexing (SPIMI) algorithm.
    It processes tokenized documents, generating a dictionary where each term is associated
    with a list of tuples containing document IDs and term frequency.

    Args:
        tokensList (list): A list of lists, each containing tokens from a document.
        testCount (int, optional): A threshold for the number of term-document pairs to process.

    Returns:
        tuple: A tuple containing the inverted index and the time taken to build it.
    """
    index = {}
    docID = 1
    pairCount = 0
    # Print collection size and start building index
    print(f'Collection size: {len(tokensList)}')
    print(f'Building SPIMI index ...')
    startTime = time.time()
    for tokens in tokensList:
        for token in tokens:
            pairCount += 1
            # Update or add token in the index
            if token in index:
                postingsList = index[token]
                if postingsList[-1][0] == docID:
                    postingsList[-1] = (docID, postingsList[-1][1] + 1)
                else:
                    postingsList.append((docID, 1))
            else:
                index[token] = [(docID, 1)]

            if testCount and pairCount >= testCount:
                break
        docID += 1
        if testCount and pairCount >= testCount:
            break
    # Sort index by keys (terms) before saving
    index = dict(sorted(index.items(), key=lambda item: item[0]))
    endTime = time.time()

    print(f'SPIMI index built!')
    save2json(index, 'spimi.json')
    return index, endTime - startTime
```

after the first *testCount* terms have been processed. In main() we set testCount to 10 000. The function builds the index in one pass over the data within tokensList. It adds terms to the index, along with a postings list of the docIDs it occurs in. The postings are in the form of docID-term frequency tuples, and everytime a term reoccurs in the same document, the term frequency for that document is incremented. Throughout the construction process the function is being timed, and when it returns the final index is also returns the elapsed time.

save2json()

Just before returning, the function calls save2json() which is seen below. It simply outputs the dictionary structure to a JSON file object within the /indexes directory.

```python
def save2json(data, filename):
    """
    Saves a given data object to a JSON file.

    Args:
        data: The data to be saved.
        filename (str): The name of the file to save the data in.
    """

    # Create directory if it doesn't exist
    if not os.path.exists('indexes'):
        os.makedirs('indexes')

    # Save data to a JSON file
    with open(f'./indexes/{filename}', 'w') as f:
        json.dump(data, f, indent=4)
```

naïve()

Back in main(), the spimi index and its construction time is retrieved is retrieved, and naïve() is now called, which can be seen below. It has the same parameters as spimi(), which work the same way. However, this function stores term-docID pairs in a list F, which it then sorts and

```python
def naive(tokensList, testCount = None):
    """
    Builds an inverted index using a naive approach. It processes term-document pairs,
    sorts them, and then groups them to create postings lists.

    Args:
        tokensList (list): A list of tokenized documents.
        testCount (int, optional): A threshold for the number of term-document pairs to process.

    Returns:
        tuple: A tuple containing the inverted index and the time taken to build it.
    """

    F = []  # List to store term-docID pairs
    docID = 1
    print(f'Building Naive index ...')

    startTime = time.time()
    # Create term-docID pairs
    for tokens in tokensList:
        for token in tokens:
            F.append((token, docID))
            if testCount and len(F) == testCount:
                break
        docID += 1
        if testCount and len(F) == testCount:
            break

    # Sort and remove exact duplicates
    F.sort()
    F = list(dict.fromkeys(F))

    # Create postings lists
    index = {}
    for term, docID in F:
        if term in index:
            if index[term][-1] != docID:
                index[term].append(docID)
        else:
            index[term] = [docID]

    endTime = time.time()
    print(f'Naive index built!')
    save2json(index, 'naive.json')
    return index, endTime - startTime
```

cleans for duplicates, all before actually constructing the index. When, building the index, it works exactly as does spimi(), although it does not maintain term frequency. It also writes the index to a JSON file, and returns both the index and the elapsed time.

Back in main(), both indexes are now built, and both of their elapsed times have been retrieved.

getStats()

The next function call is to getStats() which can be seen below. This function simply takes all of the previously gathered data as input parameters, and displays it in a proper manner via

```python
def getStats(naiveIndex, naiveTime, spimiIndex, spimiTime, tokensTime):
    """
    Prints statistics comparing the Naive and SPIMI indexing methods. It displays the time taken
    to create token streams, build indexes, and the sizes of the created indexes.

    Args:
        naiveIndex (dict): The inverted index created using the naive approach.
        naiveTime (float): The time taken to build the naive index.
        spimiIndex (dict): The inverted index created using the SPIMI approach.
        spimiTime (float): The time taken to build the SPIMI index.
        tokensTime (float): The time taken to create token streams.
    """
    print('\n========== STATISTICS ==========\n')
    # Token streams creation time
    print(f'Token streams creation time (sec): {tokensTime}')
    # SPIMI Index Construction Statistics
    print('\n--------------------\n| SPIMI Construction |\n--------------------')
    print(f'\nTime taken for SPIMI index to process 10 000 terms (ms): {round(spimiTime * 1000, 3)}')
    print(f'Size of SPIMI index: {len(spimiIndex)} terms')
    # Naive Index Construction Statistics
    print('\n--------------------\n| Naive Construction |\n--------------------')
    print(f'\nTime taken for Naive index to process 10 000 terms (ms): {round(naiveTime * 1000, 3)}')
    print(f'Size of Naive index: {len(naiveIndex)} terms')
    # Differences between SPIMI and Naive Methods
    print('\n--------------\n| Differences |\n--------------')
    time_diff = naiveTime - spimiTime
    time_diff_percent = (time_diff / naiveTime) * 100
    print(f'\nTime difference (ms): {round(time_diff * 1000, 3)}')
    print(f'Time difference (%): {round(time_diff_percent, 3)}%')
```

terminal output. It displays the elapsed time for both index construction techniques, as well as the time difference in ms and %, as follows.

```
Retrieving tokens ...
Tokens retrieved!
Collection size: 21578
Building SPIMI index ...
SPIMI index built!
Building Naive index ...
Naive index built!

========== STATISTICS ==========

Token streams creation time (sec): 10.812

--------------------
| SPIMI Construction |
--------------------

Time taken for SPIMI index to process 10 000 terms (ms): 4.77
Size of SPIMI index: 2769 terms

--------------------
| Naive Construction |
--------------------

Time taken for Naive index to process 10 000 terms (ms): 6.836
Size of Naive index: 2769 terms

--------------
| Differences |
--------------

Time difference (ms): 2.066
Time difference (%): 30.224%
```

Subproject 2: First-Run Walkthrough

Running *subProject2.py* will trigger the *main()* function call and result in the following sequence of operations until terminal output.

Main

The main function below executes, retrieving the collection, building both indexes, and calling the queryManager() to handle continuous prompts for queries. The function begins by calling the same three initial functions as in the previous subproject. It calls getTokens(), as well as spimi() and naïve(), however it does not set thresholds for the indexes, and neither does it bother retrieving their construction times. Since these have all been seen in subproject1, we will skip to the last function call to queryManager()

```python
def main():
    """
    The main function to initiate the search engine application. It retrieves token streams, builds indexes,
    and launches the query manager to handle user queries.

    This function is the starting point of the application.
    """
    tokenStreams = getTokens()
    spimiIndex, _ = spimi(tokenStreams)
    naiveIndex, _ = naive(tokenStreams)
    queryManager(spimiIndex, naiveIndex, tokenStreams)
```

queryManager()

One of the main functions which handles all functionality to this subproject is the queryManager(), which is seen bellow. This function is responsible for continuously prompting the user for input parameters to the auxiliary functions necessary to the overall functioning of subproject2. It continuously prompts the user for a query, until 'q' is entered, which terminates the engine. For a single term query, the function would skip the prompt for operations, but in the case of a multi-term query the first function call would be to getOperation().

```python
def queryManager(spimiIndex, naiveIndex, collection):
    """
    Manages the querying process. It prompts the user to enter queries and handles the retrieval
    and display of results using the chosen index and ranking method.

    Args:
        spimiIndex (dict): The SPIMI index.
        naiveIndex (dict): The Naive index.
        collection (list): The document collection, used for BM25 ranking.

    This function continues to prompt for queries until the user chooses to quit.
    """
    print('\n====== WELCOME TO THE REUTERS21578 SEARCH ENGINE ======\n')
    print('\n** Enter \'q\' to quit **\n')
    print('** Queries do not require boolean operators **\n')
    while True:
        query = input('Enter your query: ')
        if query == 'q':
            break
        numKeywords = len(tokenizer(query))
        operation = getOperation() if numKeywords > 1 else None
        spimi, naive = getIndexes(spimiIndex, naiveIndex)
        queryTermRanking, bm25Ranking = getRanking(operation, spimi)
        queryTest(query=query, spimiIndex=spimi, naiveIndex=naive, collection=collection, operation=operation, queryTermRanking=queryTermRanking, bm25Ranking=bm25Ranking)
    print('\n====== SEE YOU LATER :P ======\n')
```

getOperation()

This function, as seen below, is simply responsible for continuously prompting the user for a Boolean operation (AND, OR) until a valid input is received.

```python
def getOperation():
    """
    Prompts the user to choose a boolean operation ('AND' or 'OR') for the query processing.

    Returns:
        str: The chosen boolean operation ('AND' or 'OR').
    """
    valid = False
    while not valid:
        operation = input('Enter your operation (\'AND\', \'OR\') : ')
        # Check if the entered operation is valid
        if operation in ['AND', 'OR']:
            valid = True
        else:
            print(f'Sorry, {operation} is not a valid operation.')
    return operation
```

Once back in queryManager() with an operation retrieved, the next function call is to getIndexes() which can be seen below. This function works the same way in principle, prompting the user until a valid input is entered. The user could select to query either index, or both. The function will return the required, pre-built indexes depending on the selection.

```python
def getIndexes(spimiIndex, naiveIndex):
    """
    Prompts the user to choose which index(es) to query - SPIMI, Naive, or both.

    Args:
        spimiIndex (dict): The SPIMI index.
        naiveIndex (dict): The Naive index.

    Returns:
        tuple: A tuple of the selected indexes. None is used for unselected indexes.
    """
    valid = False
    while not valid:
        answer = input('Which index would you like to query? \'n\' = naive, \'s\' = spimi, \'b\' = both : ')
        # Validate the user input and return the appropriate indexes
        if answer in ['n', 's', 'b']:
            valid = True
        else:
            print(f'Sorry, {answer} is not a valid answer.')
    # Return the appropriate combination of indexes based on user choice
    if answer == 's':
        return spimiIndex, None
    elif answer == 'n':
        return None, naiveIndex
    elif answer == 'b':
        return spimiIndex, naiveIndex
```

Once back in queryManager(), with the selected indexes, the next function call is to getRanking(), which principally works the same as the previous two functions. It prompts the user until it receives a valid ranking selection. The user can select query term ranking if they previously selected their operation as *OR*. They can select bm25 ranking if at least one of their selected indexes is the spimi index, or they can input *None* for no ranking. The function returns Boolean flags for each ranking technique.

```python
def getRanking(operation, spimiIndex):
    """
    Prompts the user to choose a ranking method for the query results.

    Args:
        operation (str): The chosen boolean operation ('AND' or 'OR').
        spimiIndex (dict): The SPIMI index.

    Returns:
        tuple: A tuple (queryTermRank, bm25) indicating whether query term ranking or BM25 should be used.
    """
    queryTermRank = False
    bm25 = False
    valid = False
    while not valid:
        answer = input('Which ranking would you like? \'q\' = query term ranking, \'b\' = bm25, \'n\' = none : ')
        # Validate the user input and set the ranking method accordingly
        if answer in ['q', 'b', 'n']:
            if answer == 'q' and operation != 'OR':
                print('Sorry, query term ranking is only for multi-term OR queries.')
            elif answer == 'b' and not spimiIndex:
                print('Sorry, bm25 ranking is only available for the spimi index.')
            else:
                valid = True
                if answer == 'q':
                    queryTermRank = True
                elif answer == 'b':
                    bm25 = True
        else:
            print(f'Sorry, {answer} is not a valid answer.')

    return queryTermRank, bm25
```

queryTest()

Once back in queryManager(), now with the ranking flags set, queryManager() makes its last function call of the current iteration to queryTest(). The queryTest() function as shown below, is the function responsible for processing the queries depending on the inputted runtime parameters, which is why it is the longest. However it works rather simply. It begins by calling the tokenizer() on the query to retrieve a list of the tokenized query terms. If the list is empty then that means the inputted query was invalid. If it has a length of 1 then that means it is a single term query, and if more than 1 it is a multi-term query. The function begins by verifying if the naïve index has been initialized, if so, it steps into the if block and verifies the nature of the query. If it's a single term query, it queries the naïve index for the query term and outputs the

```python
def queryTest(query, spimiIndex=None, naiveIndex=None, collection=None, operation='OR', queryTermRanking=False, bm25Ranking=False):
    """
    Processes a search query against provided SPIMI and/or Naive indexes. It supports single or multi-term
    queries with AND/OR operations and optional ranking (Query Term Ranking or BM25).

    Args:
        query (str): The search query.
        spimiIndex (dict): The SPIMI index (optional).
        naiveIndex (dict): The Naive index (optional).
        collection (list): The document collection (used for BM25 ranking).
        operation (str): The boolean operation to apply ('AND', 'OR').
        queryTermRanking (bool): Flag to use Query Term Ranking.
        bm25Ranking (bool): Flag to use BM25 ranking.

    The function tokenizes the query, checks the appropriate index for each term, and then applies the
    specified operation (AND/OR) to compute the final result set. It also handles ranking if specified.
    """
    # Tokenize the input query
    queryTerms = tokenizer(query)

    # Check if the query is empty after tokenization
    if len(queryTerms) == 0:
        print(f'Sorry, \'{query}\' is not a valid query')
        return

    # Process with Naive Index if available
    if naiveIndex:
        print("\n====== NAIVE INDEX ======\n")
        # Process single-term queries
        if len(queryTerms) == 1:
            result = naiveIndex.get(queryTerms[0])  # Get postings for the single query term.
        else:
            # Process multi-term queries
            postingsLists = []
            result = None
            for term in queryTerms:
                postingsList = naiveIndex.get(term)  # Get postings for each query term.
                postingsLists.append(postingsList)
            # Apply boolean operations
            if operation == 'AND':
                result = conjunction(postingsLists)
            elif operation == 'OR':
                result = disjunction(postingsLists, queryTermRanking)

        # Print the results from Naive Index
        if result:
            print(f"QUERY: \'{query}\'\nOPERATION: {operation}\nRANKING: {'Query Term Ranking' if queryTermRanking else 'None'} \nPOSTINGS LIST SIZE: {len(result)}\nPOSTINGS LIST: {str(result)}\n")
        else:
            print(f"QUERY: \'{query}\' - Your query returned no results with operation: {operation}.\n")
```

result. In the case of a multi-term query, the function iterates through the query terms, and retrieves their postings lists.

conjunction()

It then calls either conjunction() or disjunction() depending on the operation chosen at runtime. If AND, then conjunction is called.

```python
def conjunction(postingsLists):
    """
    Performs an AND operation on a list of postings lists, returning the intersection of these lists.

    Args:
        postingsLists (list of lists): A list where each element is a postings list (list of docIDs).

    Returns:
        list: The intersection of the postings lists, representing documents that contain all terms.
    """
    # Return None if any of the postings lists is None
    if None in postingsLists:
        return None

    # Sort postings lists by length for efficient intersection
    sortedPostingsLists = sorted(postingsLists, key=len)
    # Start with the shortest postings list
    finalPostingsList = sortedPostingsLists[0]

    # Intersect with each subsequent postings list
    for postingsList in sortedPostingsLists[1:]:
        finalPostingsList = intersect(finalPostingsList, postingsList)
        # If intersection is empty, no further processing is needed
        if not finalPostingsList:
            break
    return finalPostingsList
```

Intersect()

The conjunction function calls the intersect() function on each of the postings lists that have been passed as a parameter beginning with the two smallest. It does so two at a time, and combines the results in order. The intersection function can be seen below, which functions as a regular intersection, combining postings lists by common docIDs while avoiding duplicates and incrementing with respect to the list positioned at the smallest docID, until the end of one of the lists has been reached.

```python
def intersect(pList1, pList2):
    """
    Computes the intersection of two postings lists.

    Args:
        pList1 (list): The first postings list.
        pList2 (list): The second postings list.

    Returns:
        list: The intersection of pList1 and pList2.
    """
    result = []
    i = j = 0
    # Iterate through both lists to find common elements
    while i < len(pList1) and j < len(pList2):
        if pList1[i] == pList2[j]:
            result.append(pList1[i])
            i += 1
            j += 1
        elif pList1[i] < pList2[j]:
            i += 1
        else:
            j += 1
    return result
```

Back in the queryTest() function, in the case of a conjunction, the resulting list of docIDs is returned and displayed.

disjunction()

However, if the selected operation is *OR* then the disjunction() function is called which can be seen below. It simply unions the postings lists in order and without duplicates, unless they are all *None*.

```python
def disjunction(postingsLists, queryTermRanking=False):
    """
    Performs an OR operation on a list of postings lists, returning the union of these lists. Can also
    apply query term ranking based on term frequency across the postings lists.

    Args:
        postingsLists (list of lists): A list where each element is a postings list (list of docIDs).
        queryTermRanking (bool): Flag to apply query term ranking.

    Returns:
        list: The union of the postings lists, with optional query term ranking applied.
    """
    # Filter out None postings lists (terms not found)
    validPostingsLists = [plist for plist in postingsLists if plist is not None]

    # If all postings lists are None, return None
    if not validPostingsLists:
        return None

    # Flatten the list of lists into a single list using itertools.chain
    unionPostings = list(chain(*validPostingsLists))

    # If query term ranking is not applied, remove duplicates and sort
    if not queryTermRanking:
        finalPostings = sorted(set(unionPostings))
    else:
        # Sort by document ID and apply query term ranking
        finalPostings = sorted(unionPostings)
        finalPostings = queryTermRank(finalPostings)

    return finalPostings
```

queryTermRank()

In the case of queryTermRanking flag is set to true, then this is handled in the disjunction() function by calling the queryTermRank() function on the resulting disjunct set. The queryTermRank() can be seen below, where it maintains a local dictionary of docIDs, incrementing each of their counts everytime they are encountered in the postingsList. It then sorts the dictionary with respect to value, and returns the dictionary as a list of ordered docID-count tuples.

```python
def queryTermRank(postingsList):
    """
    Applies query term ranking to a postings list. Ranks documents based on the number of query terms they contain.

    Args:
        postingsList (list): A postings list (list of docIDs).

    Returns:
        list: A list of tuples (docID, score) where score is the count of query terms in the document.
    """
    # Count the frequency of each document ID in the postings list
    docFreq = {}
    for docID in postingsList:
        docFreq[docID] = docFreq.get(docID, 0) + 1

    # Sort documents by frequency (score) in descending order
    rankedDocs = sorted(docFreq.items(), key=lambda item: item[1], reverse=True)
    return [(docID, score) for docID, score in rankedDocs]
```

Back in queryTest(), now with the resulting disjunct set of docIDs in the case of an *OR* operation, the results are displayed to terminal. The function then breaks from the *if naïve* if block, and checks if the spimi index parameter was initialized. If yes, then it enters the if block and goes through the same steps as above for the naïve if block. **The only difference**, is two main points.

convertPostingsLists()

Firstly, since the spimi index's postings lists are lists of tuples and not regular docIDs, it calls a function to normalize the postings lists to a simple list of docIDs. The function can be seen below. It does so before calling either conjunction() or disjunction().

```python
def convertPostingsLists(postingsLists):
    """
    Convert a list of postings lists with (docID, tf) tuples to a list of lists of docIDs.
    If a postings list is None, it remains None.

    Args:
        postingsLists (list): A list of postings lists, where each postings list contains tuples of (docID, tf).

    Returns:
        list: A list of lists, where each inner list contains docIDs or is None.
    """
    # Convert each postings list to a list of docIDs, or keep as None if postings list is None
    return [[docID for docID, tf in postingsList] if postingsList is not None else None for postingsList in postingsLists]
```

bm25()

Secondly, the only other difference is that it checks if the bm25 flag has been set. If yes, it makes a function call to the bm25() function below. It maintains the scores of documents in a local dictionary rankedResults. It does so by retrieving and iterating over the docIDs in each of the queryTerms' postings lists. When the docID matches one of the docIDs in the result parameter, then it computes the score, and stores/updates it in the rankedResults dictionary. Once all queryTerms have been iterated over the dictionary is ordered by score and returned as a sorted list of docID-score tuples.

```python
def bm25(queryTerms, result, index, collection, k1, b):
    """
    Applies the BM25 ranking formula to the result set of a query. It calculates scores for documents
    based on term frequency, document frequency, and document length.

    Args:
        queryTerms (list): List of tokenized query terms.
        result (list): List of document IDs obtained from the initial search.
        index (dict): The search index.
        collection (list): The document collection, to compute document lengths.
        k1 (float): BM25 parameter k1.
        b (float): BM25 parameter b.

    Returns:
        list: A sorted list of tuples (docID, score), ranked according to BM25.
    """
    N = len(collection)  # Total number of documents
    L_total = sum(len(doc) for doc in collection)  # Total length of all documents
    L_avg = L_total / N  # Average document length

    rankedResults = {}

    if result is None:
        return None

    # Calculate BM25 score for each term in each document
    for term in queryTerms:
        postingsList = index.get(term, [])
        df = len(postingsList)  # Document frequency
        idf = math.log(N / df) if df != 0 else 0  # Inverse document frequency

        for docID, tf in postingsList:
            if docID and docID in result:
                L_d = len(collection[int(docID)])  # Document length for the current docID
                # BM25 formula
                score = idf * ((tf * (k1 + 1)) / (k1 * ((1 - b) + b * (L_d / L_avg)) + tf))
                # Accumulate score for each document
                if docID not in rankedResults:
                    rankedResults[docID] = 0.0
                rankedResults[docID] += score

    # Check if all scores are zero
    if all(score == 0.0 for score in rankedResults.values()):
        return None

    # Sort the results by score in descending order
    sortedRankedList = sorted(rankedResults.items(), key=lambda item: item[1], reverse=True)
    return sortedRankedList
```

Once this is complete, the queryManager() then prompts the user for another query, restarting the whole process.

***** For output logs, please refer to Appendix A and Appendix B of 4013360_report.pdf**