Assignment 1

Srijan Garg (2019448)
Mohd. Siraj Ansari (2019176)

Q1 Data Preprocessing

(i) Relevant Text Extraction

To begin, we extracted the necessary data from the files by retrieving the text that fell between the <Title> and <Text> tags. Our approach involved utilizing Beautiful Soup, an HTML parsing tool, to locate the <title> and <text> tags and subsequently extract their contents. To combine the two strings, we simply inserted a space between them.

Text before Extraction:

```
<DOC>
<DOCNO>
1
</DOCNO>
<TITLE>
experimental investigation of the aerodynamics of a
wing in a slipstream .
</TITLE>
<AUTHOR>
brenckman,m.
</AUTHOR>
<BIBLIO>
j. ae. scs. 25, 1958, 324.
</BIBLIO>
<TEXT>
  an experimental study of a wing in a propeller slipstream was
made in order to determine the spanwise distribution of the lift
increase due to slipstream at different angles of attack of the wing
and at different free stream to slipstream velocity ratios .  the
results were intended in part as an evaluation basis for different
theoretical treatments of this problem .
  the comparative span loading curves, together with supporting
evidence, showed that a substantial part of the lift increment
produced by the slipstream was due to a /destalling/ or
boundary-layer-control
effect .  the integrated remaining lift increment,
after subtracting this destalling lift, was found to agree
well with a potential flow theory .
  an empirical evaluation of the destalling effects was made for
the specific configuration of the experiment .
</TEXT>
</DOC>
```

Text after extraction:

```
experimental investigation of the aerodynamics of a
wing in a slipstream .

  an experimental study of a wing in a propeller slipstream was
made in order to determine the spanwise distribution of the lift
increase due to slipstream at different angles of attack of the wing
and at different free stream to slipstream velocity ratios .  the
results were intended in part as an evaluation basis for different
theoretical treatments of this problem .
  the comparative span loading curves, together with supporting
evidence, showed that a substantial part of the lift increment
produced by the slipstream was due to a /destalling/ or
boundary-layer-control
effect .  the integrated remaining lift increment,
after subtracting this destalling lift, was found to agree
well with a potential flow theory .
  an empirical evaluation of the destalling effects was made for
the specific configuration of the experiment .
```

(ii) Preprocessing

1. First we converted the text to lowercase.
2. Fixed the contractions as well
3. Performed tokenization on the text with the help of nltk library
4. We then removed all the stopwords from our tokens.
5. Punctuations were removed from the tokens
6. Blank spaces were removed from the tokens

After implementing the preprocessing steps mentioned above, the result is the following set of tokens, using the example mentioned in the previous section.

```
['experimental', 'investigation', 'aerodynamics', 'wing',
'slipstream', 'experimental', 'study', 'wing', 'propeller',
'slipstream', 'made', 'order', 'determine', 'spanwise',
'distribution', 'lift', 'increase', 'due', 'slipstream', 'different',
'angles', 'attack', 'wing', 'different', 'free', 'stream',
'slipstream', 'velocity', 'ratios', 'results', 'intended', 'part',
'evaluation', 'basis', 'different', 'theoretical', 'treatments',
'problem', 'comparative', 'span', 'loading', 'curves', 'together',
'supporting', 'evidence', 'showed', 'substantial', 'part', 'lift',
'increment', 'produced', 'slipstream', 'due', 'destalling',
'boundarylayercontrol', 'effect', 'integrated', 'remaining', 'lift',
'increment', 'subtracting', 'destalling', 'lift', 'found', 'agree',
'well', 'potential', 'flow', 'theory', 'empirical', 'evaluation',
'destalling', 'effects', 'made', 'specific', 'configuration',
'experiment']
```

Q2 Boolean Queries

(i) Creating the inverted index

We have implemented a unigram inverted index to store the frequency and posting lists for each term in a collection of documents. We have used Python's defaultdict and os libraries to read the contents of each document and parse them into individual tokens.

We have initialized two defaultdicts: inv_idx_freq and inv_idx_postings. The inv_idx_freq dictionary stores the document frequency of each term, while inv_idx_postings stores the postings list for each term. The postings list is a list of document IDs in which the term appears.

We then iterate through each file in a folder specified by folder_path, read the contents of the file, and extract the document ID and set of unique tokens. The helper function is then called to update the frequency and postings list for each term in the set of tokens.

The helper function updates the postings list for each term by appending the current document ID. It also updates the frequency of each term by incrementing the value of the corresponding key in inv_idx_freq.
Finally, we sort the index in alphabetical order with respect to terms and sort the individual posting lists corresponding to each term in the index.

(ii) Creating Inverted Index Operation Functions

We have provided support for all the 4 boolean queries required for the question.

1. AND query: To provide the support for AND query in the code, we have defined a function **t1_and_t2** which takes two input parameters - **posting1** and **posting2**. These postings are essentially lists of document IDs in which the respective words have occurred. The function then proceeds to find the intersection of these two lists and returns a list of common document IDs.
   We first check if any of the two postings is empty. If so, then we return an empty list as the intersection of empty and non-empty lists is always empty. We then set two pointers **ptr1** and **ptr2** at the beginning of each list and start iterating over them. We initialize variable **comparisons** to keep track of the number of comparisons done during the iteration.
   During the iteration, we compare the document IDs pointed by the two pointers. If the two document IDs are the same, we add it to the **ans** list and move both pointers forward. If the document ID of the first posting is less than that of the second, we move the first pointer forward. If the document ID of the second posting is less than that of the first, we move the second pointer forward.

   At the end of the iteration, we return the **ans** list, which contains the common document IDs in both postings, along with the number of comparisons performed during the iteration.

2. AND NOT query: To provide the support for AND NOT query in the code, we have defined a function **t1_and_not_t2** that takes two postings as input: **posting1** and

posting2. The function returns the documents that contain the term in posting1 but not the term in posting2. The algorithm for this function is similar to the previous function **t1_and_t2**, but with some additional steps.
We first initialize two pointers, **ptr1** and **ptr2**, to iterate over the two postings. We also initialize an empty list ans to store the documents that contain the term in **posting1** but not in **posting2**. We keep track of the number of comparisons we make between the two postings with the variable **comparisons**.
We then enter a loop that continues as long as both pointers have not reached the end of their respective postings. At each iteration of the loop, we compare the document IDs at the current positions of the two pointers. If they are equal, we increment both pointers to move on to the next documents. If the document ID in **posting1** is less than the document ID in **posting2**, we add the document ID in **posting1** to **ans** and increment the **ptr1** pointer. Otherwise, we increment the **ptr2** pointer.
Once we have finished iterating over both postings, we add any remaining documents in **posting1** to **ans**. Finally, we return **ans** and the number of comparisons we made.

3. OR query: To provide the support for OR query in the code, we have defined a function **t1_or_t2** that takes two sorted posting lists as input, and returns their union. We use two pointers **ptr1** and **ptr2** that iterate over the two posting lists, comparing the elements at the pointers and adding the smaller one to the output list until we reach the end of both lists.
   We also keep track of the number of comparisons made during the merging process. If any of the input posting lists are empty, we simply return the other posting list as the output. Finally, we return the output list and the number of comparisons made. Overall, this function efficiently merges two sorted posting lists by avoiding duplicate entries, and returns the result as a sorted list.

4. OR NOT query: To provide the support for OR NOT query in the code, we have defined two functions **not_t1(posting1**) and **t1_or_not_t2(posting1, posting2)**.
   The **not_t1** function takes a posting list **posting1**, and uses the **t1_and_not_t2** function to return the documents that are in the **docIDs** list, but not in posting1. It then returns the resulting list and the number of comparisons made by **t1_and_not_t2**.
   The **t1_or_not_t2** function takes two posting lists, **posting1** and **posting2**, and returns the documents that are in **posting1** or not in **posting2**. It first checks if posting1 is empty, in which case it calls **not_t1(posting2)** to return the documents that are not in posting2. If **posting1** is not empty, it uses **not_t1(posting2)** to get the documents that are not in posting2, and then uses **t1_or_t2(posting1, not_posting2)** to return the documents that are in either posting1 or not in posting2. It then returns the resulting list and the number of comparisons made by both **not_t1** and **t1_or_t2.**

(iii) Query Preprocessing and Query Matching.

Query and operations input is taken from the user. Then, preprocessing is done on the query to generate query tokens.

The below steps of preprocessing were performed for preprocessing the query text:

1. First we converted the text to lowercase.
2. Fixed the contractions as well
3. Performed tokenization on the text with the help of nltk library

4. We then removed all the stopwords from our tokens.
5. Punctuations were removed from the tokens
6. Blank spaces were removed from the tokens

After this, query matching is done using the query tokens and operations to find the relevant documents.

Input:

```
1
Investigation of experimental wing
AND,AND
```

Output:

```
Query 1: investigation AND experimental AND wing
Number of documents retrieved for query 1: 19
Names of documents retrieved for query 1: cranfield0001.txt,
cranfield0030.txt, cranfield0078.txt, cranfield0189.txt,
cranfield0202.txt, cranfield0222.txt, cranfield0225.txt,
cranfield0442.txt, cranfield0497.txt, cranfield0636.txt,
cranfield0694.txt, cranfield0712.txt, cranfield1062.txt,
cranfield1074.txt, cranfield1075.txt, cranfield1092.txt,
cranfield1337.txt, cranfield1338.txt, cranfield1341.txt
Number of comparisons required for query 1: 679
```

Q3 Phrase Queries

(i) Bigram Inverted Index

We have implemented a helper function to generate bigrams for a set of tokens in a given document, and we have used this function to generate the inverted index for all documents in a given folder. We have used a defaultdict to initialize the document frequency and postings dictionaries, where the postings dictionary returns an empty list for any key that does not exist. We then iterate over all documents in the given folder and parse each document to extract the set of tokens, which we convert to a list of tokens using ast.literal_eval. We then use the helper function to generate bigrams from the list of tokens, and update the postings and document frequency accordingly. Finally, we sort the index in alphabetical order by terms and then sort the individual posting lists corresponding to each term in the index.

Boolean AND support for Bigram Inverted Index

We have defined a function **t1_and_t2** that takes two posting lists as input, representing the documents in which two given terms appear, and returns their intersection along with the number of comparisons made.

The function first checks if either of the two input posting lists is empty. If so, it returns an empty list and the number of comparisons made is 0. If both the posting lists are non-empty, we initialize two pointers, one for each posting list.

We then start iterating over both the posting lists simultaneously until we have compared all elements in one of them. At each iteration, we compare the document ID at the current position

of both the pointers. If the document ID is the same in both the posting lists, we add the document ID to the output list and increment both the pointers. Otherwise, we increment the pointer corresponding to the smaller document ID.

Finally, we return the intersection of the two posting lists and the number of comparisons made during the execution of the function.

(ii) Positional Index

We have implemented a helper function named **pos_helper()** that takes in a list of tokens and a document ID as input. This function is used to create a positional index for the given set of tokens.

The function first checks if a token is already present in the **term_doc_pos** dictionary. If the token is present, then it adds the current document ID and its corresponding positions of the token in the document to the dictionary. If the token is not present, then it creates a new entry in the dictionary with the token as the key and a nested dictionary as its value. The nested dictionary stores the document ID and its corresponding positions of the token in the document.

After adding all the tokens and their positions to the **term_doc_pos** dictionary, we create another dictionary called **term_doc_freq**, which stores the number of documents in which a token occurs. We loop through all the keys in **term_doc_pos**, and count the number of documents for each token and store it in **term_doc_freq**.

Finally, we sort both the dictionaries **term_doc_pos** and **term_doc_freq** in alphabetical order with respect to the terms. The internal dictionaries of **term_doc_pos** are also sorted in increasing order of document IDs, and the list of positions is sorted in increasing order as well.

Boolean AND support for Positional Index

We have defined three functions for positional search using an index created in previous steps. The first function **helper2** checks whether the given document contains a given token at a given position. This function returns **True** if the document contains the token at the given position, and **False** otherwise.

The second function **validDocID** takes a document ID, a list of tokens, and a starting position as input. It then iterates over the tokens and checks if the given document contains each token at the corresponding position. The function returns **True** if all the tokens are found in the document at the given positions, and **False** otherwise.

The third function **pos_search** is the main function that takes a list of tokens as input and returns the list of document IDs that contain all the tokens in the list at the desired positions. The function first checks if the first token in the list is present in the positional index. If not, the function returns an empty list. Next, it gets all the document IDs and the corresponding positions for the first token from the index. Then, for each document ID, the function checks if the document contains all the tokens at the desired positions. If a document contains all the tokens, it is added to the list of results.

Overall, these functions work together to perform positional search using an index created from a set of documents.

(iii) Query Preprocessing and Query Matching

Query input is taken from the user. Then, preprocessing is done on the query to generate query tokens.

The below steps of preprocessing were performed for preprocessing the query text::

1. First we converted the text to lowercase.
2. Fixed the contractions as well
3. Performed tokenization on the text with the help of nltk library
4. We then removed all the stopwords from our tokens.
5. Punctuations were removed from the tokens
6. Blank spaces were removed from the tokens

After this, bigram tokens are created using the query tokens. Then, bigram tokens are used for query matching using bigram inverted index and query tokens are used for query matching using positional index. Then, relevant documents using both indexes are shown as output.

(iv) Comparing the two indexes

Input:

```
1
Surface pressure distribution
```

Output:

```
Number of documents retrieved for query 1 using bigram inverted
index: 10
Names of documents retrieved for query 1 using bigram inverted index:
cranfield0025.txt, cranfield0101.txt, cranfield0173.txt,
cranfield0193.txt, cranfield0209.txt, cranfield0310.txt,
cranfield1231.txt, cranfield1246.txt, cranfield1248.txt,
cranfield1262.txt
Number of documents retrieved for query 1 using positional index: 9
Names of documents retrieved for query 1 using positional index:
cranfield0025.txt, cranfield0101.txt, cranfield0173.txt,
cranfield0193.txt, cranfield0310.txt, cranfield1231.txt,
cranfield1246.txt, cranfield1248.txt, cranfield1262.txt
```

From the above output, we can see that bigram inverted index gave a false positive in file “cranfield0209.txt”