VRIJE
UNIVERSITEIT
BRUSSEL

PROJECT INFORMATION RETRIEVAL

GROUP 7

Anisha Sachdeva - 0571873

Arne Van Quickelberghe - 0565405

Introduction

Aspects

Aspect 1	Aspect 2: B	Aspect 3: B	Aspect 4: A

Dataset
The dataset we used are conversations from the European parliament. We found it via the link below.

http://www.statmt.org/europarl/

Before we could use these documents, we first extracted the xml attributes from the documents. During our simulation we took the first 1000 documents and noted a total of 72947 type of words.

Division of work
The work was divided as follows, Anisha did the implementation of aspects 2 and 3 and Arne implemented aspects 1 and 4. We have each completed the report for the aspects we specified.

Aspect 1: Positional Index

1.1 Architecture

Aspect 1 consists of retrieving a positional index from a collection of documents. We have created a positional index component that takes a document as input and processes it into an index. So we are going to process each document from the collection by our positional index component into an index, and then bring the indexes together into one index.

When we look at our positional index component architecture, it consists of four blocks, each with its own task. The exact function of each of these blocks is further explained in chapter 1.2 [1].

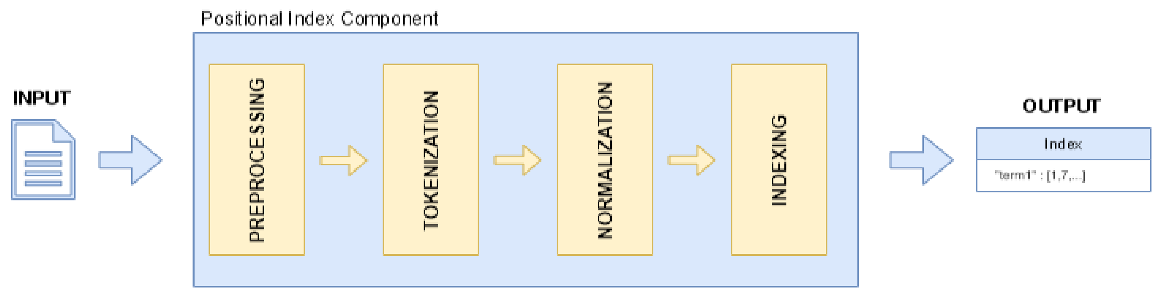


Figure 1: Positional Index Component

1.2 Implementation

To explain the implementation of the inverted index component. We are going with a demo-document through this chapter, with this demo-document we are going to walk through the different blocks in our positional index component to sketch a idea of what happens in each block and what the result from each block would look like.

```
'This is a document from a list of documents, each document contains a list of
words.'
```

Figure 2: Demo-document

1.2.1 Preprocessing

The preprocessing block is the first block of the positional index component, it will remove all punctuation marks from the text and also bring the text to lowercase.

```
# This method will preprocess the text
def preprocess(text):
    text = lowercase(text)
    text = removePunctuation(text)
    return tekst

# This method will convert the text to lowercase
def lowercase(text):
    return text.lower()

# This method will remove all the punctuations
def removePunctuation(text):
    translator = str.maketrans('', '', string.punctuation)
    return text.translate(translator)
```

Code fragment 1: Preprocessing

After preprocessing we obtain the following result:

```
'this is a document from a list of documents each document contains a list of
words'
```

Figure 3: Preprocessing result demo-document

1.2.2 Tokenization

After preprocessing, the text ends up in the tokenization block, the tokenization is done on the basis of spaces and the text is converted into a list of words.

```
# This method will tokenize the text
def tokenize(text):
    return text.split()
```

Code fragment 2: Tokenization

After the tokenization we obtain the following result:

```
['this', 'is', 'a', 'document', 'from', 'a', 'list', 'of', 'documents', 'each',
'document', 'contains', 'a', 'list', 'of', 'words']
```

Figure 4: Tokenization result demo-document

1.2.3 Normalization

The normalization block is going to normalize the list of words. This happens because of two things, first the stop words are removed from the text. This is done using the ntlk library. This library contains a dataset with 'all' English stop words such that they can be filtered out of the list of words. After that, stemming is applied to the remaining list of words. After applying stemming, we therefore obtain a list of terms.

```
# This method will normalize the words
def normalize(words):
    words = removeStopWords(words)
    words = applyStemming(words)
    return words

# This method will remove all the english stop words in from a list of words
def removeStopWords(words):
    # Retrieve a collection of stopwords from the ntlk library
    stop_words = set(stopwords.words('english'))
    # Return each word which is not a stopword
    return [word for word in words if not word in stop_words]

# This method will apply stemming to a list of words
def applyStemming(words):
    # Creating a porterStemmer from the ntlk library
    ps = PorterStemmer()
    # Return a array of terms
    return list(map(lambda word: ps.stem(word), words))
```

Code fragment 3: Normalization

After normalization we obtain the following result:

```
['document', 'list', 'document', 'document', 'contain', 'list', 'word']
```

Figure 5: Normalization result demo-document

[* For this part to work with the ntlk library in python it is important that the following items are installed via the terminal:
- pip install nltk (the ntlk library)
- python -m nltk.downloader stopwords (the dataset with stop words for the ntlk library)]

1.2.4 Normalization

The last block is the indexing block. This will receive the list of terms as input and will build an index based on the list of terms. The index keeps track of the term and the position where the term occurs.

```
# This method will index a collection of words
def index(words):
    indx = {}
    for position, word in enumerate(words):
        if not word in indx:
            indx[word] = [position]
        else:
            indx[word].append(position)
    return indx
```

Code fragment 4: Indexing

After indexing we obtain the following result:

```
{'document': [0, 2, 3], 'list': [1, 5], 'contain': [4], 'word': [6]}
```

Figure 6: Indexing result demo-document

1.3 Evaluation

In general we are quite satisfied with the functionality of the positional index component, we tried to take into account some basic optimizations such as the removal of stop words and stemming.

When we look at the performance of our implementation we notice that the normalization block in our positional index component weighs very heavily on the total performance. When we looked at this further, we saw that the stemming part takes a lot of time. This would therefore offer the greatest possible scope for improvements in the future.

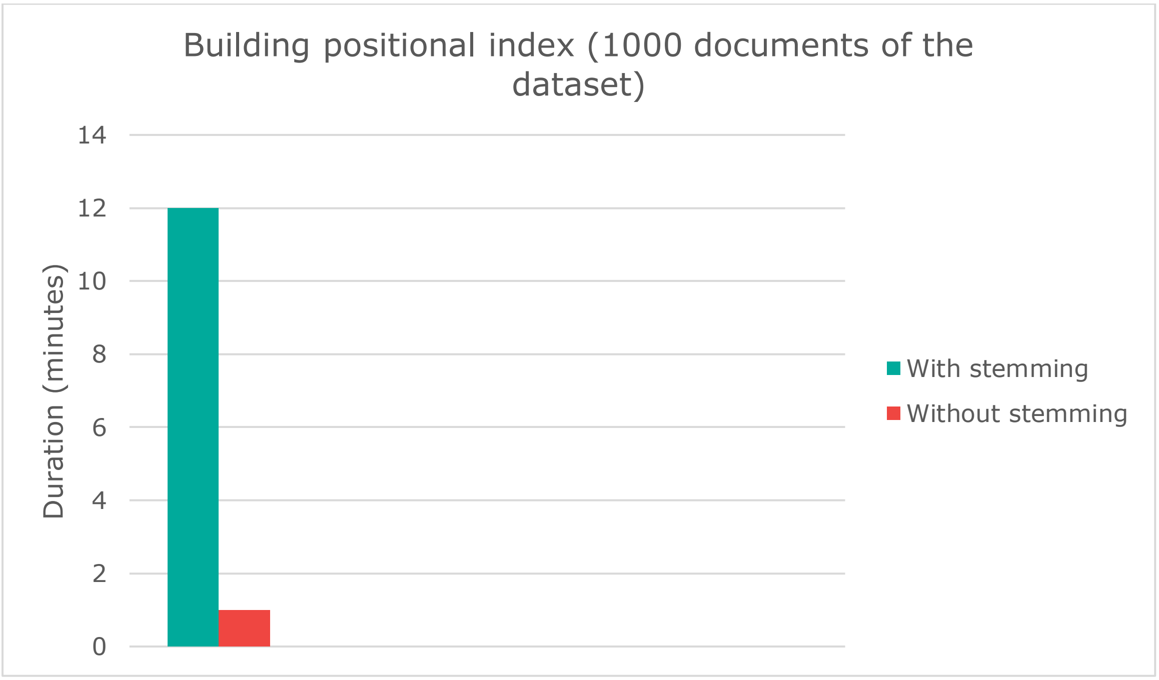


Figure 7: Positional index performance 1000 documents dataset

Another thing is that we process file by file. However, if we were in a situation where we have collections of documents with very large documents, it would be more interesting to do stream processing. Such that interim results can also be given and that we would not process file by file.

Aspect 2: Querying

2.1 Architecture

Once, we extracted the positional index with some pre-processing and normalization done the next aspect to be implemented was querying from the positional index created.
While looking for some particular information, users tend to search for keywords that may be contained in some documents making it easier to sort and read the documents rather than reading each and every document to retrieve that specific information. The more is the closeness between the query terms, the more is the possibility that the documents containing them would be relevant to the concept around the user query.
Based on this concept, we implemented proximity search querying i.e. basis the two input strings and the expected number of words between them, a user can search for all the documents satisfying the given query. We included the following two types of query search:

1) The strings given by the end user should be within 'n' number of words, while the order of the string entered by the user should be considered.
2) The strings given by the end user should be near 'n' number of words, independent of the order they appear.

2.2 Implementation

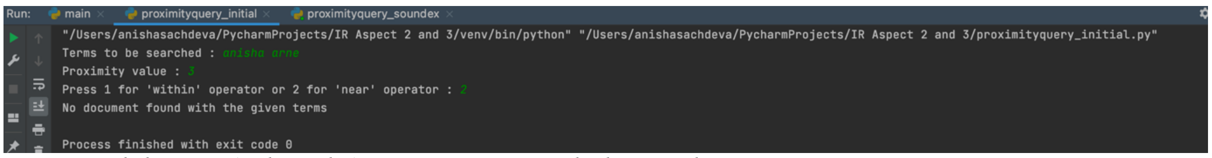
Following is the approach which we developed to implement the proximity search.

1. User enter the terms to be searched for along with the proximity (within or near) value.
2. If term 1 or term 2 are not present in any of the documents, a message will be displayed indicating that no document was found with the required terms.
3. Next, user is given the option to select if he wants to search the terms using the 'within' operator or the 'near' operator.
   Option 1: 'Within' operator
   Option 2: 'Near' operator
4. If user selects option 1,
   set the value of near = 0
   else, near = -proximity
5. Now, find the list of intersecting documents and using the following condition
   near <= position[second term] – position[first term] <= proximity
   find and print the list of documents as per the option chosen between 'within' or 'near' operator.
6. If the terms are present in atleast one of the documents but either they do not have any intersecting document or if they have an intersecting document, they are not within or near the proximity of specified words, a message indicating the same would be displayed.

[Note:

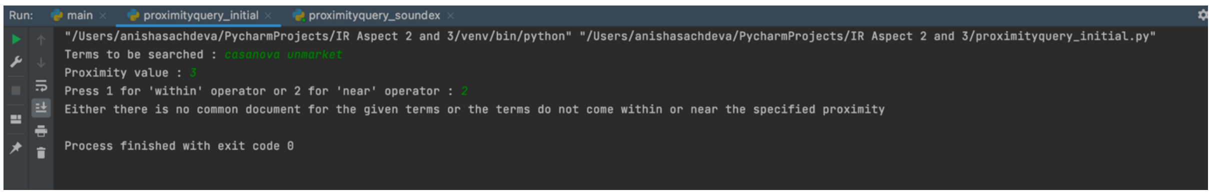
- For our implementation of proximity queries, we will consider two-word phrases.
- For the initial implementation, we will be considering the positional index extracted from aspect 1 (with stemming included and stop words removed).]

Below are the examples from the document dataset that we considered representing the three possible outcomes that we discussed above.



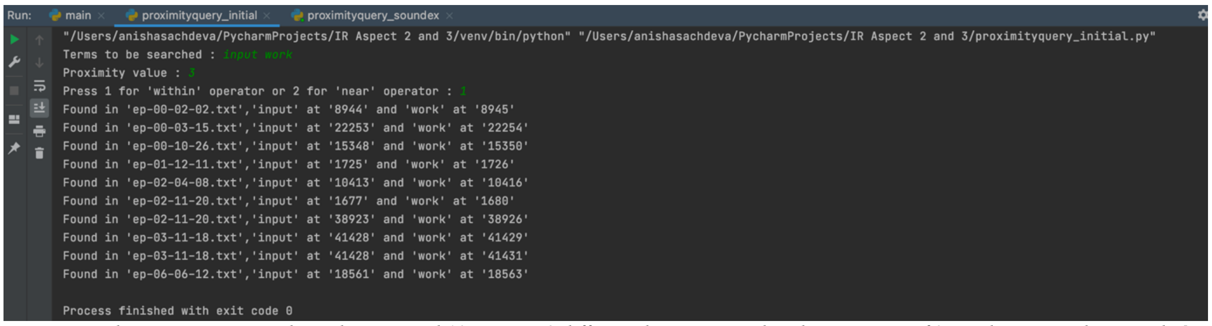
```
"/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/venv/bin/python" "/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/proximityquery_initial.py"
Terms to be searched : anisha arne
Proximity value : 3
Press 1 for 'within' operator or 2 for 'near' operator : 2
No document found with the given terms

Process finished with exit code 0
```

Figure 8: *Both the terms 'Anisha' and 'Arne' were not present in the document dataset*



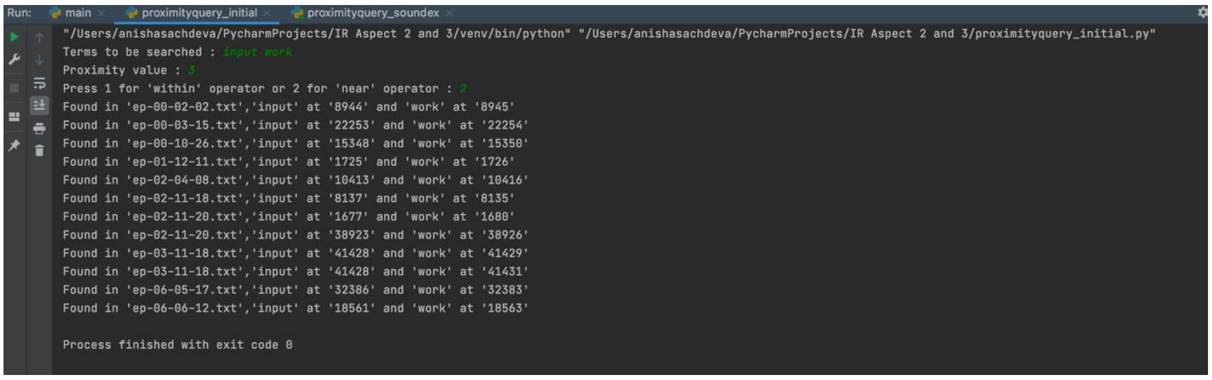
```
"/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/venv/bin/python" "/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/proximityquery_initial.py"
Terms to be searched : casanova unmarket
Proximity value : 3
Press 1 for 'within' operator or 2 for 'near' operator : 2
Either there is no common document for the given terms or the terms do not come within or near the specified proximity

Process finished with exit code 0
```

Figure 9: Both the terms, 'casanova' and 'unmarket' were present in atleast one of the documents but either they did not have any intersecting document or if they had an intersecting document, they were not within the specified proximity of 3 words



```
"/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/venv/bin/python" "/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/proximityquery_initial.py"
Terms to be searched : input work
Proximity value : 3
Press 1 for 'within' operator or 2 for 'near' operator : 1
Found in 'ep-00-02-02.txt','input' at '8944' and 'work' at '8945'
Found in 'ep-00-03-15.txt','input' at '22253' and 'work' at '22254'
Found in 'ep-00-10-26.txt','input' at '15348' and 'work' at '15350'
Found in 'ep-01-12-11.txt','input' at '1725' and 'work' at '1726'
Found in 'ep-02-04-08.txt','input' at '10413' and 'work' at '10416'
Found in 'ep-02-11-20.txt','input' at '1677' and 'work' at '1680'
Found in 'ep-02-11-20.txt','input' at '38923' and 'work' at '38926'
Found in 'ep-03-11-18.txt','input' at '41428' and 'work' at '41429'
Found in 'ep-03-11-18.txt','input' at '41428' and 'work' at '41431'
Found in 'ep-06-06-12.txt','input' at '18561' and 'work' at '18563'

Process finished with exit code 0
```

Figure 10: The terms, 'input' and 'work' appeared 10 times in 8 different documents within the proximity of 3 words in accordance with the order of 'input' preceding 'work'



```
"/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/venv/bin/python" "/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/proximityquery_initial.py"
Terms to be searched : input work
Proximity value : 3
Press 1 for 'within' operator or 2 for 'near' operator : 2
Found in 'ep-00-02-02.txt','input' at '8944' and 'work' at '8945'
Found in 'ep-00-03-15.txt','input' at '22253' and 'work' at '22254'
Found in 'ep-00-10-26.txt','input' at '15348' and 'work' at '15350'
Found in 'ep-01-12-11.txt','input' at '1725' and 'work' at '1726'
Found in 'ep-02-04-08.txt','input' at '10413' and 'work' at '10416'
Found in 'ep-02-11-18.txt','input' at '8137' and 'work' at '8135'
Found in 'ep-02-11-20.txt','input' at '1677' and 'work' at '1680'
Found in 'ep-02-11-20.txt','input' at '38923' and 'work' at '38926'
Found in 'ep-03-11-18.txt','input' at '41428' and 'work' at '41429'
Found in 'ep-03-11-18.txt','input' at '41428' and 'work' at '41431'
Found in 'ep-06-05-17.txt','input' at '32386' and 'work' at '32383'
Found in 'ep-06-06-12.txt','input' at '18561' and 'work' at '18563'

Process finished with exit code 0
```

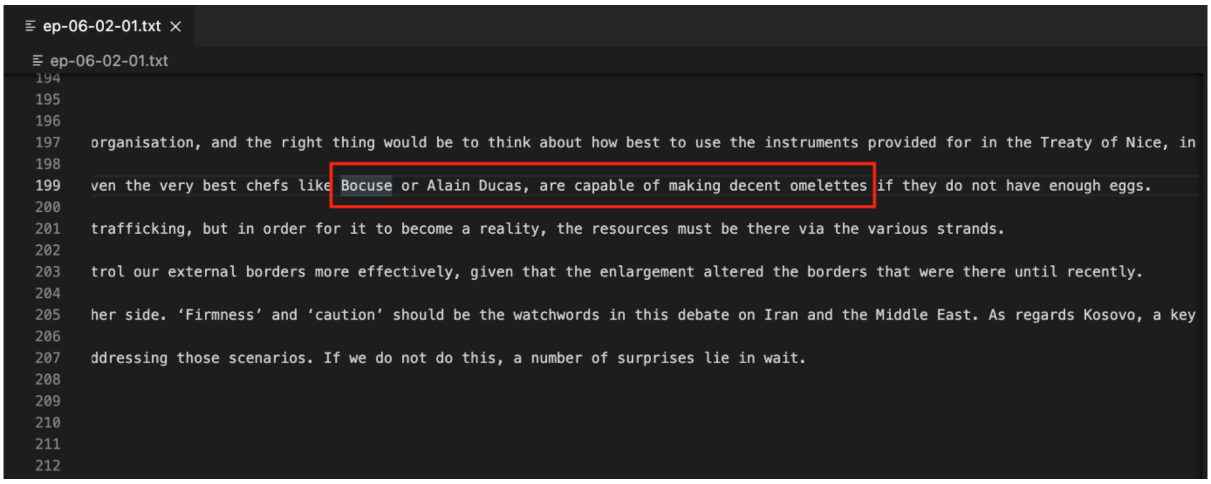
Figure 11: The terms, 'input' and 'work' appeared 12 times in 10 different documents within the proximity of 3 words irrespective of the order of appearance

2.3 Evaluation

We had chosen the first 1000 documents from the European parliament corpus and developed it's positional index (in aspect 1). The positional index created contained preprocessing steps like removal of punctuation marks, lowering of alphabet cases and normalization techniques like stemming. However, we observed that while querying this index, there were a lot of false positives present. Let us consider a simple example to understand what is a false positive: a document containing a sentence "The lion is the king of the jungle". Here if we were to search for lion and jungle within 4 words, the document containing only this sentence should not be in the output/ result. However, if we would remove the stop word "the" from the sentence, the proximity between lion and jungle would indeed become less than 4 words and the document would appear in the output.

Another issue encountered was due to stemming. While implementing stemming rules, many morphological variants generated were not the real words and it was difficult to search for the exact words while querying. Many queries resulted is not able to find the words because either some special character was added to the original words or some were stemmed to words which did not make any sense.

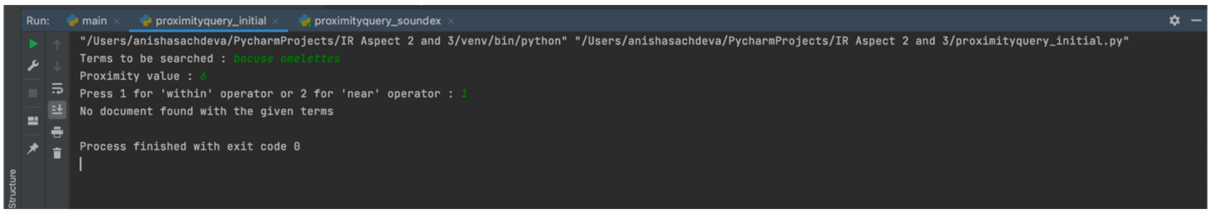
Following is an example from the set of documents we considered, which resulted in false positives explaining both the scenarios which we discussed above.
We randomly selected a document ‘ep-06-02-01.txt’ and chose the terms ‘Bocuse’ and ‘omelettes’ to examine. We noted from the text document that the difference between both the terms were of 9 words.



```
ep-06-02-01.txt

organisation, and the right thing would be to think about how best to use the instruments provided for in the Treaty of Nice, in

ven the very best chefs like Bocuse or Alain Ducas, are capable of making decent omelettes if they do not have enough eggs.

trafficking, but in order for it to become a reality, the resources must be there via the various strands.

trol our external borders more effectively, given that the enlargement altered the borders that were there until recently.

her side. ‘Firmness’ and ‘caution’ should be the watchwords in this debate on Iran and the Middle East. As regards Kosovo, a key

ddressing those scenarios. If we do not do this, a number of surprises lie in wait.
```

Figure 12: Screenshot of randomly chose text document

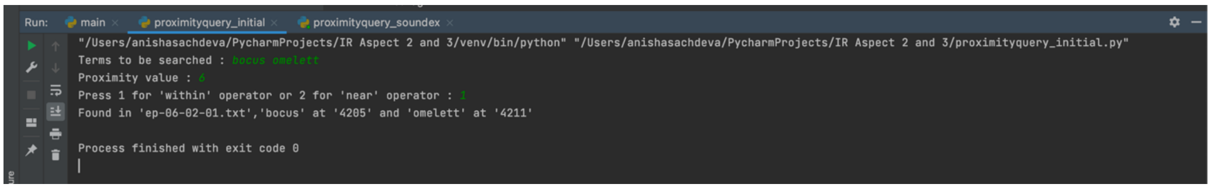
While searching for the correctly spelled terms, the system was unable to look for the terms in the positional index due to stemming and falsely resulted in giving the output as ‘No document found with the given terms’



```
"/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/venv/bin/python" "/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/proximityquery_initial.py"
Terms to be searched : bocuse omelettes
Proximity value : 6
Press 1 for 'within' operator or 2 for 'near' operator : 1
No document found with the given terms

Process finished with exit code 0
```

Figure 13: False result due to stemming

Next, we searched for the terms as they appeared after stemming in the positional index. ‘Bocus’ and ‘omelett’ were searched with proximity set as 6. The system resulted in the false positive output by referring to the document ‘ep-06-02-01.txt’. Due to the removal of stop words ‘or’, ‘are’ and ‘of’, the system incorrectly measured the difference between the terms as 6 instead of 9.

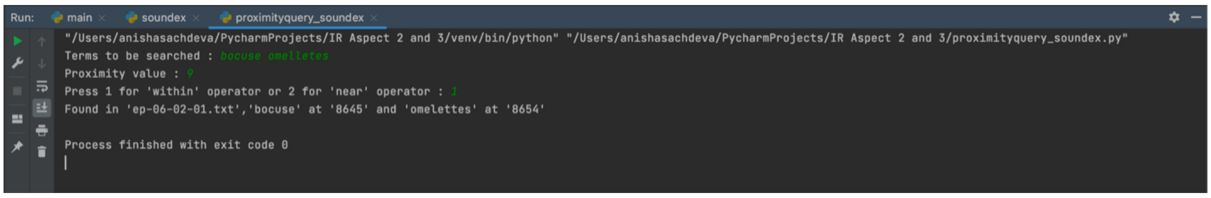


```
"/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/venv/bin/python" "/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/proximityquery_initial.py"
Terms to be searched : bocus omelett
Proximity value : 6
Press 1 for 'within' operator or 2 for 'near' operator : 1
Found in 'ep-06-02-01.txt','bocus' at '4205' and 'omelett' at '4211'

Process finished with exit code 0
```

Figure 14: False positive due to removal of stop words

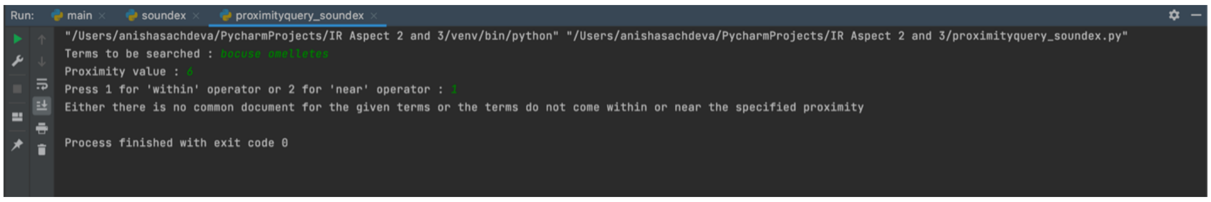
One way to improve our querying approach would be to implement *soundex* algorithm which would generate the same four character codes for original and stemmed words. However, as we considered only the first 1000 documents for our project and in order to make our approach even better we decided to have the positional index without stemming and with the inclusion of stop words. Also, the list of stop words was containing 900 English stop words and could easily be incorporated in the positional index.

We chose to go for accuracy than simplicity with respect to our project considerations. With the new positional index, we again queried for ‘Bocuse’ and ‘omelettes’ and it indeed resulted accurately. (Refer to section 3.3 of the report for more results)



```
"/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/venv/bin/python" "/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/proximityquery_soundex.py"
Terms to be searched : bocuse omelletes
Proximity value : 9
Press 1 for 'within' operator or 2 for 'near' operator : 1
Found in 'ep-06-02-01.txt','bocuse' at '8645' and 'omelettes' at '8654'

Process finished with exit code 0
```

Figure 15: The system identified both the terms as they were not stemmed in the positional index. With the inclusion of stop words the proximity was accurately calculated as 9 between the specified terms



```
"/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/venv/bin/python" "/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/proximityquery_soundex.py"
Terms to be searched : bocuse omelletes
Proximity value : 6
Press 1 for 'within' operator or 2 for 'near' operator : 1
Either there is no common document for the given terms or the terms do not come within or near the specified proximity

Process finished with exit code 0
```

Figure 16: With the inclusion of stop words, there were no false positives and the proximity was calculated accurately

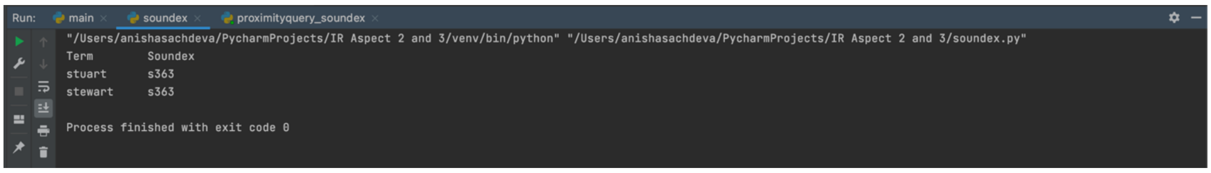
Aspect 3: Normalization

3.1 Architecture

The next aspect to be implemented was for phonetic correction i.e. misspelled English words due to the similar pronunciations. It may happen, that while querying a user might enter a misspelled word as it's pronunciation might be resembling to the target word (that he actually wanted to query). As an example, while searching for people with last name as 'Smith', the user might search for 'Smyth' as they both sound similar, however, the expected output would be incorrect due to the spelling mistake.
To render this problem, we implemented the *soundex* algorithm which would put similar sounds in equivalence class and generates a four-character code for the terms present in the positional index. Also, as mentioned above, *soundex* algorithm can be very helpful in scenarios wherein stemming is implemented.

3.2 Implementation

We implemented the *soundex* algorithm as explained in the section 3.4 of the reference book "An Introduction to Information Retrieval" [1].
As an example, we generated the codes for 'Stuart' and 'Stewart' and noted that both had the same code 'S363' implying that if a user mistakenly types 'Stuart' in place of 'Stewart' and if the term is searched with the help of the *soundex* code, the result would indeed contain documents with any (or may be both) of the two possibilities.



```
"/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/venv/bin/python" "/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/soundex.py"
Term        Soundex
stuart      s363
stewart     s363

Process finished with exit code 0
```

Figure 17: 'Stuart' and 'Stewart' generated the same soundex code S363

3.3 Evaluation

Next, in order to evaluate the efficiency of *soundex* algorithm we included the *soundex* code for each of the term in our new positional index (which included the stop words and no stemming was implemented).

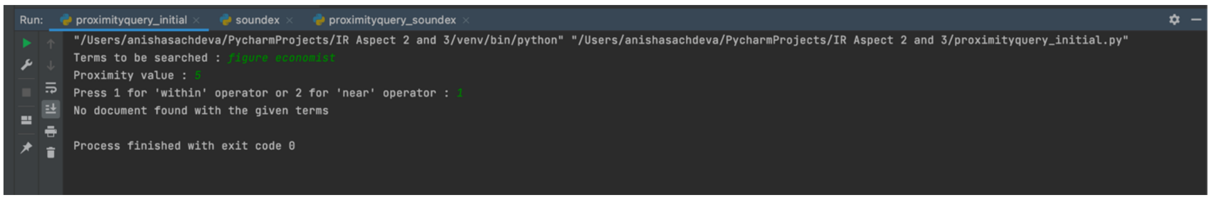
While including *soundex* codes to the positional index, the following four scenarios were acknowledged:

1. The terms consisting only of numeric values were assigned with 'None' as the *soundex* code
2. The terms consisting only of special characters were assigned with 'None' as the *soundex* code
3. The terms consisting of only of English alphabets were assigned with the respective *soundex* code
4. For the terms consisting of alphanumeric or alphabets with special characters, first the numeric values or the special characters were removed and then the *soundex* code for the remaining string containing only alphabets was generated

We observed that implementing *soundex* algorithm helped our system to increase the accuracy while querying using the proximity search for the following scenarios:

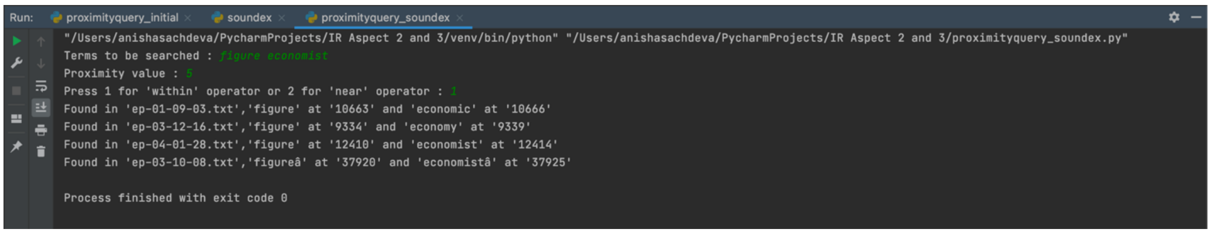
1. In case of querying terms which had some special character(s) attached to it in the documents, those terms could not be identified earlier as the exact query string would

not match. However, implementing *soundex* algorithm, the code generated by the query term could be compared and matched to the *soundex* codes present in the positional index making the proximity search more efficient.
As an example from the dataset, while we searched for the query ‘figure economist’ within the proximity of 5 words, our system without *soundex* implemented could not search for the query. However with the implementation of *soundex*, the system successfully searched for the query with the correct expected output.



```
"/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/venv/bin/python" "/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/proximityquery_initial.py"
Terms to be searched : figure economist
Proximity value : 5
Press 1 for 'within' operator or 2 for 'near' operator : 1
No document found with the given terms

Process finished with exit code 0
```

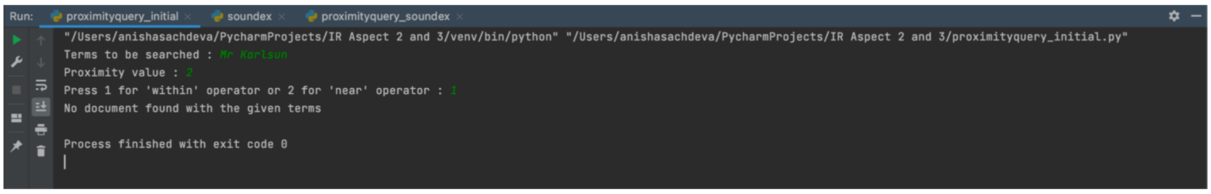
Figure 18: Query could not be executed without soundex implemented



```
"/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/venv/bin/python" "/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/proximityquery_soundex.py"
Terms to be searched : figure economist
Proximity value : 5
Press 1 for 'within' operator or 2 for 'near' operator : 1
Found in 'ep-01-09-03.txt','figure' at '10663' and 'economic' at '10666'
Found in 'ep-03-12-16.txt','figure' at '9334' and 'economy' at '9339'
Found in 'ep-04-01-28.txt','figure' at '12410' and 'economist' at '12414'
Found in 'ep-03-10-08.txt','figureâ' at '37920' and 'economistâ' at '37925'

Process finished with exit code 0
```

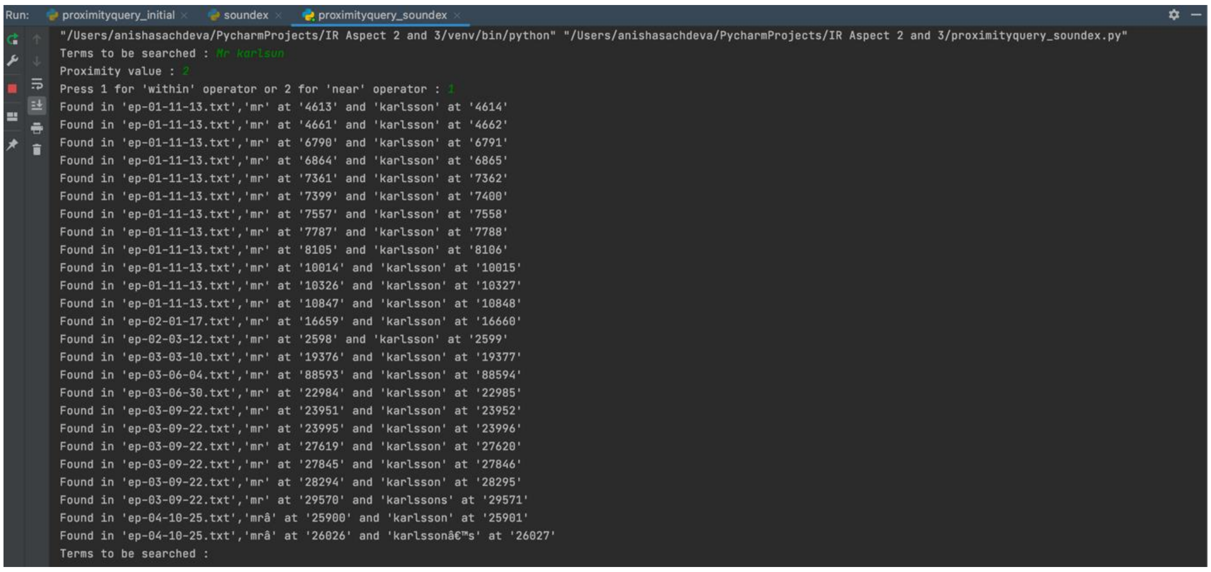
Figure 19: Query successfully executed with soundex implemented

2. While searching for queries, there might be a spelling mistake either while querying or may be the spellings could be incorrect in the text file itself. During these scenarios *soundex* codes can easily be compared for the correct results.
   As an example from the dataset, we executed a query prone to spelling mistake with ‘Mr Karlsson’ being queried as ‘Mr Karlsun’. We observed that the query failed without *soundex* while it succeeded with *soundex*.



```
"/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/venv/bin/python" "/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/proximityquery_initial.py"
Terms to be searched : Mr Karlsun
Proximity value : 2
Press 1 for 'within' operator or 2 for 'near' operator : 1
No document found with the given terms

Process finished with exit code 0
```

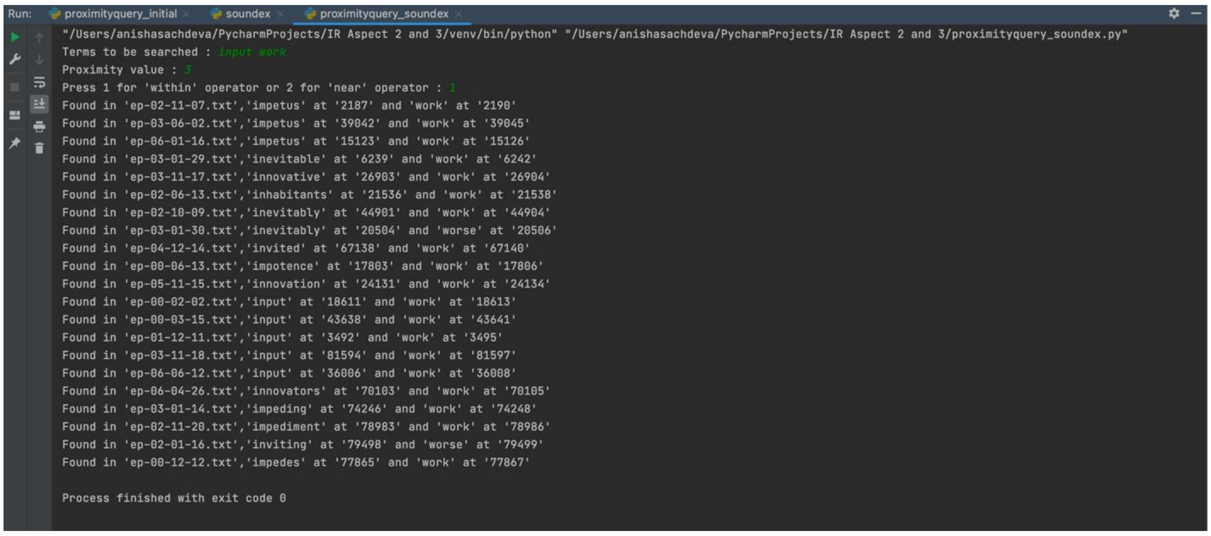
Figure 20: Query failed for ‘Mr Karlsun’ without soundex implemented



```
"/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/venv/bin/python" "/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/proximityquery_soundex.py"
Terms to be searched : Mr karlsun
Proximity value : 2
Press 1 for 'within' operator or 2 for 'near' operator : 1
Found in 'ep-01-11-13.txt','mr' at '4613' and 'karlsson' at '4614'
Found in 'ep-01-11-13.txt','mr' at '4661' and 'karlsson' at '4662'
Found in 'ep-01-11-13.txt','mr' at '6790' and 'karlsson' at '6791'
Found in 'ep-01-11-13.txt','mr' at '6864' and 'karlsson' at '6865'
Found in 'ep-01-11-13.txt','mr' at '7361' and 'karlsson' at '7362'
Found in 'ep-01-11-13.txt','mr' at '7399' and 'karlsson' at '7400'
Found in 'ep-01-11-13.txt','mr' at '7557' and 'karlsson' at '7558'
Found in 'ep-01-11-13.txt','mr' at '7787' and 'karlsson' at '7788'
Found in 'ep-01-11-13.txt','mr' at '8105' and 'karlsson' at '8106'
Found in 'ep-01-11-13.txt','mr' at '10014' and 'karlsson' at '10015'
Found in 'ep-01-11-13.txt','mr' at '10326' and 'karlsson' at '10327'
Found in 'ep-01-11-13.txt','mr' at '10847' and 'karlsson' at '10848'
Found in 'ep-02-01-17.txt','mr' at '16659' and 'karlsson' at '16660'
Found in 'ep-02-03-12.txt','mr' at '2598' and 'karlsson' at '2599'
Found in 'ep-03-03-10.txt','mr' at '19376' and 'karlsson' at '19377'
Found in 'ep-03-06-04.txt','mr' at '88593' and 'karlsson' at '88594'
Found in 'ep-03-06-30.txt','mr' at '22984' and 'karlsson' at '22985'
Found in 'ep-03-09-22.txt','mr' at '23951' and 'karlsson' at '23952'
Found in 'ep-03-09-22.txt','mr' at '23995' and 'karlsson' at '23996'
Found in 'ep-03-09-22.txt','mr' at '27619' and 'karlsson' at '27620'
Found in 'ep-03-09-22.txt','mr' at '27845' and 'karlsson' at '27846'
Found in 'ep-03-09-22.txt','mr' at '28294' and 'karlsson' at '28295'
Found in 'ep-03-09-22.txt','mr' at '29570' and 'karlssons' at '29571'
Found in 'ep-04-10-25.txt','mrâ' at '25900' and 'karlsson' at '25901'
Found in 'ep-04-10-25.txt','mrâ' at '26026' and 'karlssonâ€™s' at '26027'
Terms to be searched :
```

Figure 21: Query successfully executed for 'Mr Karlsun' with soundex implemented

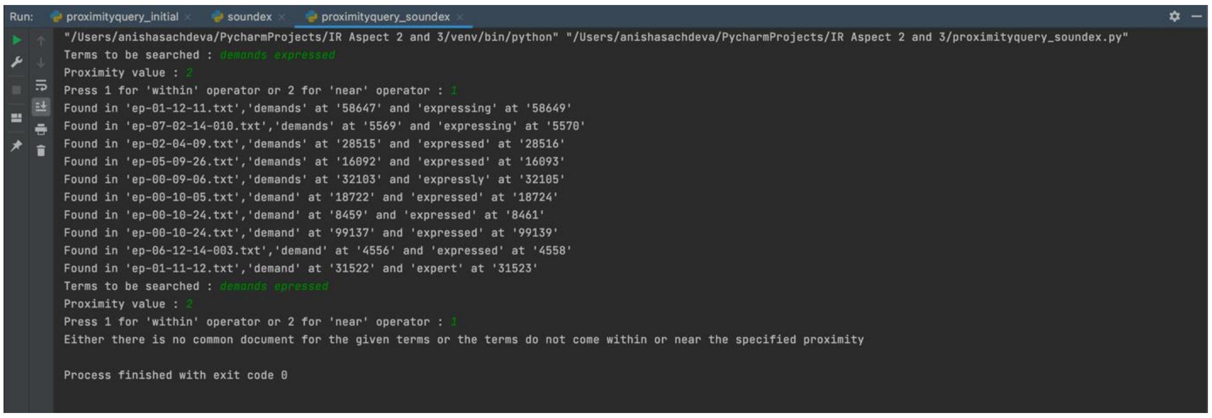
However, by using *soundex* algorithm while it would help to improve the recall by returning a larger number of documents, the precision of the search would go down. If a user would be looking for the most likely matches, he might get disappointed by seeing the large results most of which would be of no-good use.
For example, while searching for 'input work' within the proximity of 3 words the resulting output gives document ids for terms like 'impediment work' which would not be any beneficial to the user.



```
"/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/venv/bin/python" "/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/proximityquery_soundex.py"
Terms to be searched : input work
Proximity value : 3
Press 1 for 'within' operator or 2 for 'near' operator : 1
Found in 'ep-02-11-07.txt','impetus' at '2187' and 'work' at '2190'
Found in 'ep-03-06-02.txt','impetus' at '39042' and 'work' at '39045'
Found in 'ep-06-01-16.txt','impetus' at '15123' and 'work' at '15126'
Found in 'ep-03-01-29.txt','inevitable' at '6239' and 'work' at '6242'
Found in 'ep-03-11-17.txt','innovative' at '26903' and 'work' at '26904'
Found in 'ep-02-06-13.txt','inhabitants' at '21536' and 'work' at '21538'
Found in 'ep-02-10-09.txt','inevitably' at '44901' and 'work' at '44904'
Found in 'ep-03-01-30.txt','inevitably' at '20504' and 'worse' at '20506'
Found in 'ep-04-12-14.txt','invited' at '67138' and 'work' at '67140'
Found in 'ep-00-06-13.txt','impotence' at '17803' and 'work' at '17806'
Found in 'ep-05-11-15.txt','innovation' at '24131' and 'work' at '24134'
Found in 'ep-00-02-02.txt','input' at '18611' and 'work' at '18613'
Found in 'ep-00-03-15.txt','input' at '43638' and 'work' at '43641'
Found in 'ep-01-12-11.txt','input' at '3492' and 'work' at '3495'
Found in 'ep-03-11-18.txt','input' at '81594' and 'work' at '81597'
Found in 'ep-06-06-12.txt','input' at '36006' and 'work' at '36008'
Found in 'ep-06-04-26.txt','innovators' at '70103' and 'work' at '70105'
Found in 'ep-03-01-14.txt','impeding' at '74246' and 'work' at '74248'
Found in 'ep-02-11-20.txt','impediment' at '78983' and 'work' at '78986'
Found in 'ep-02-01-16.txt','inviting' at '79498' and 'worse' at '79499'
Found in 'ep-00-12-12.txt','impedes' at '77865' and 'work' at '77867'

Process finished with exit code 0
```

Figure 22: With large list of documents the precision of search decreases

As *soundex* is a phonetic algorithm, it works good for spelling correction only for those errors wherein some similar kind of sound (alphabet) is missing or is additional. But, it does not work efficiently if a complete different sound goes missing in some term. For example, querying for 'demands epressed' instead of the correct term 'demands expressed', *soundex* is unable to retrieve the (correct) documents as 'x' a prominent different sound goes missing.



```
"/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/venv/bin/python" "/Users/anishasachdeva/PycharmProjects/IR Aspect 2 and 3/proximityquery_soundex.py"
Terms to be searched : demands expressed
Proximity value : 2
Press 1 for 'within' operator or 2 for 'near' operator : 1
Found in 'ep-01-12-11.txt','demands' at '58647' and 'expressing' at '58649'
Found in 'ep-07-02-14-010.txt','demands' at '5569' and 'expressing' at '5570'
Found in 'ep-02-04-09.txt','demands' at '28515' and 'expressed' at '28516'
Found in 'ep-05-09-26.txt','demands' at '16092' and 'expressed' at '16093'
Found in 'ep-00-09-06.txt','demands' at '32103' and 'expressly' at '32105'
Found in 'ep-00-10-05.txt','demand' at '18722' and 'expressed' at '18724'
Found in 'ep-00-10-24.txt','demand' at '8459' and 'expressed' at '8461'
Found in 'ep-00-10-24.txt','demand' at '99137' and 'expressed' at '99139'
Found in 'ep-06-12-14-003.txt','demand' at '4556' and 'expressed' at '4558'
Found in 'ep-01-11-12.txt','demand' at '31522' and 'expert' at '31523'
Terms to be searched : demands epressed
Proximity value : 2
Press 1 for 'within' operator or 2 for 'near' operator : 1
Either there is no common document for the given terms or the terms do not come within or near the specified proximity

Process finished with exit code 0
```

Figure 23: Soundex is not efficient for different explicit sounds

Hence, *soundex* algorithm has its own pros and cons but with respect to our document dataset we implemented *soundex* for better results.

Aspect 4: Wildcards

4.1 Architecture

For aspect 4 we have provided two components that work together. A first component that will build a permuterm index based on a positional index. This takes a positional index as input and forms the rotations for each term and converts them to a permuterm index.

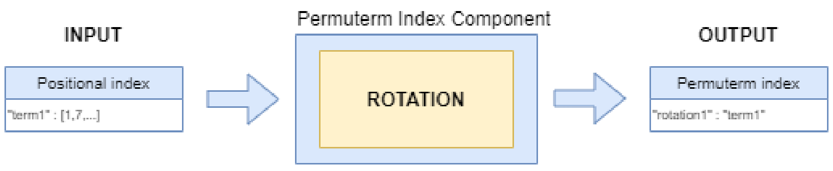


Figure 24: Permuterm Index Component

A second component for running wildcard queries. This will take as input a wildcard query as well as a positional index and a permuterm index. Then it will return as a result the posting list matching the wildcard query. This wildcard query engine consists of 3 blocks, the functionality of each of these blocks is explained in chapter 4.2 [1].

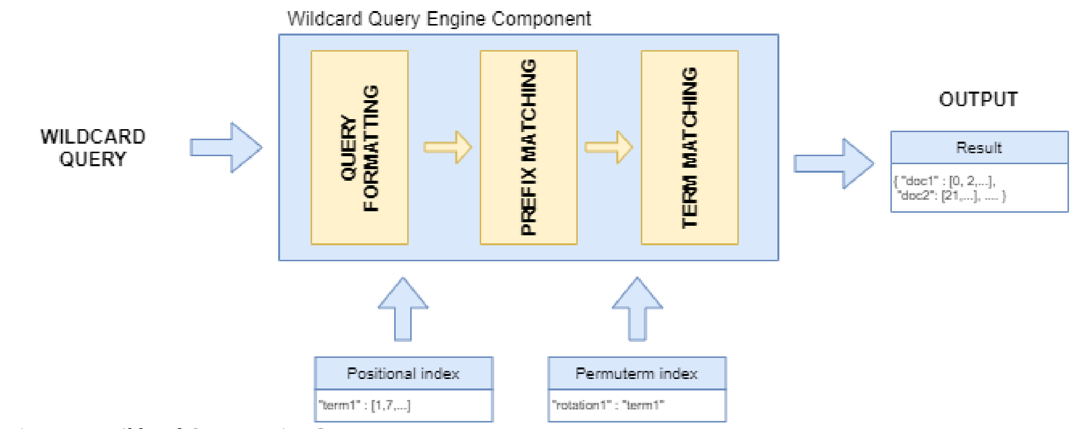


Figure 25: Wildcard Query Engine Component

4.2 Implementation

To explain the implementation of the above components, we use the two demo documents below.

```
“This is a second document, with more information.”
```

Figure 26: doc1.txt

```
“this a test document.”
```

Figure 27: doc2.txt

We also had the positional index made via the positional index component from aspect 1.

```
{'second': {'doc1.txt': [0]}, 'document': {'doc1.txt': [1], 'doc2.txt': [1]},
'inform': {'doc1.txt': [2]}, 'test': {'doc2.txt': [0]}}
```

Figure 28: Positional Index for doc1.txt and doc2.txt

4.2.1 Rotation

In the permuterm index component we use a rotation block. This will form the rotations for each term from the input (positional index), by rotating the string around the $ sign. Next we will add the rotation and the associated term to the permuterm index for each string.

```python
# This method will calculate all the rotations for a given term
def calculateRotations(term):
    rotations = []
    # Check if term is not empty
    if term:
        rotation = term + '$'
        rotations.append(rotation)
        for char in term:
            # Each iteration place the first char at the end of rotation
            rotation = rotation[1:] + char
            rotations.append(rotation)
    return rotations
```

Code fragment 5: Rotation

When we use the positional index from the demo documents as input to the permuterm index component, it will calculate the rotations for each term. We then get the following result:

```
{'second$': 'second', 'econd$s': 'second', 'cond$se': 'second', 'ond$sec':
'second', 'nd$seco': 'second', 'd$secon': 'second', '$second': 'second',
'document$': 'document', 'ocument$d': 'document', 'cument$do': 'document',
'ument$doc': 'document', 'ment$docu': 'document', 'ent$docum': 'document',
'nt$docume': 'document', 't$documen': 'document', '$document': 'document',
'inform$': 'inform', 'nform$i': 'inform', 'form$in': 'inform', 'orm$inf':
'inform', 'rm$info': 'inform', 'm$infor': 'inform', '$inform': 'inform',
'test$': 'test', 'est$t': 'test', 'st$te': 'test', 't$tes': 'test', '$test':
'test'}
```

Figure 29: Permuterm Index for doc1.txt and doc2.txt

4.2.2 Query formatting

The first block of the wildcard query engine component is the query formatting. This will convert a wildcard query into a query that we can later run on the basis of prefix. Depending on the situation, we will convert the query into a prefix query in a different way.

```python
# this method will format a given query to support prefix matching
def formatQueries(query):
    queries = []
    parts = query.split('*')
    if len(parts) == 3:
        #case 1: *X
        if parts[0] == '':
            query = parts[0]
        #case 4: X*Y*Z
        else:
            queries.append(parts[2] + "$" + parts[0])
            queries.append(parts[1])
    #case 1: X*
    elif parts[1] == '':
        queries.append(parts[0])
    #case 2:
    elif parts[0] == '':
        queries.append(parts[1] + '$')
    #case 3: X*Y
    elif parts[0] != '' and parts[1] != '':
        queries.append(parts[1] + "$" + parts[0])
    return queries
```

Code fragment 6: Query formatting

4.2.3 Prefix matching

The second block will match the obtained prefix to the rotations of the permuterm index, and then return the terms for which there is a match.

```
# This method will return all the terms where the prefix matches the rotation
def prefixMatch(prefix, permuterm):
    terms = []
    for term in permuterm.keys():
        if term.startswith(prefix):
            terms.append(permuterm[term])
    return terms
```

Code fragment 7: Prefix matching

When we look at this for the query from 4.2.2 we get the following result:

```
['inform']
```

Figure 30: Result prefix matching for query "inf"'*

Of course we only get "inform" instead of "information" here because we use stemming in the positional index.

4.2.3 Term matching

The third block also the last block will match the obtained terms to the positonal index, and return all postings as a result.

```
# This method will match the term with the positional index
def termMatch(terms, positionalIndex):
    positions = []
    for term in terms:
        positions.append(positionalIndex[term])
    return positions
```

Code fragment 8: Term matching

When we then look at the result for the query from 4.2.2, we get the following result:

```
[{'doc1.txt': [2]}]
```

Figure 31: Result term matching for query "inf" and the demo-documents*

4.3 Running a query

So with the code below we can run a query with our query engine component. As a demo we ran the query w*rk with our dataset.

```
wildcard_query_engine_component.query('w*rk', permutermIndex, positionalIndex)
```

Code fragment 9 Running a query

The query engine will match this query to the permuterm index and can match it to the following words. (This of course happens internally in the query engine)

```
['work', 'woodwork', 'watermark', 'werk', 'winmark']
```

Figure 32: Matched terms

The final result of the query is added as an attachment, named "result_query_wrk.txt".

4.4 Evaluation

In general, we are satisfied with how our permuterm index works. We have divided the different wildcard queries into cases and implemented them accordingly. The way of implementation also ensures that we have separate components that were easy to test and can be combined with each other.

However, when looking at memory usage, our permuterm index is very large. This is because the permuterm will rotate each term and in this way cause the dictionary to expand relatively A possible optimization would be a compressed permuterm index.

Conclusion

Our overall findings from this project are very positive. We have been able to apply the material from the lessons in practice. We have noted several things here.

We noticed that when making the positional index, normalization can be a performance bottleneck. Applying normalization and in particular stemming caused a significant delay, about 10 times slower. Also, when we looked at the optimization of proximity queries, it turned out that our proximity query was more accurate without normalization (without removing stop words and without stemming).

Further, we observed that while we implemented soundex algorithm and then queried our document dataset using proximity search it helped us to increase the accuracy. With the considered amount of text files, though implementing soundex overall increased the accuracy, but we also noted some of its drawbacks like while improving the recall, the precision of search was lowered down.

When we then look at the permuterm aspect we see very clearly that the permuterm index requires a large amount of memory.

We have noticed very clearly during the implementation of our aspects that there is not one correct solution, but that the design of these aspects involves making choices between optimization, memory usage, performance, precision, accuracy and that making these design choices depends on the requirements of the application.

References

[1] Schütze, H., Manning, C. D., & Raghavan, P. (2008). *Introduction to information retrieval*. Cambridge: Cambridge University Press.