Information Retrieval

Instructions for running your system (Engineering a Complete System)

- This assignment saw me utilise a number of programs such as Elasticsearch, Python, Jupyter Notebook and Ubuntu. There were a few ways in navigating this task and although tricky I hope I can provide a detailed view of the code used to operate.
- Instructions on how to control each component I will hope to outline in this report but running the system can be founded in searching for the keyword of the article in the search area and it returns the most relevant document of all 1000 imported.
- The following code is a good overview of how the code runs by showing indexing, tokenizing, normalizing and selecting keywords with these helper functions. Although this code is at the top, the pre-processing follows afterwards as we load data to Elasticsearch.

```
import pandas as pd
import json
from elasticsearch import Elasticsearch
from sklearn.feature_extraction.text import CountVectorizer
import re
from nltk.stem import LancasterStemmer

def sort_coo(coo_matrix):
    tuples = zip(coo_matrix.col, coo_matrix.data)
    return sorted(tuples, key=lambda x: (x[1], x[0]), reverse=True)

def extract_topn_from_vector(feature_names, sorted_items, topn=30):
    """get the feature names and tf-idf score of top n items"""

    sorted_items = sorted_items[:topn]
    score_vals = []
    feature_vals = []

    for idx, score in sorted_items:

        score_vals.append(round(score, 3))
        feature_vals.append(feature_names[idx])

    results= {}
    for idx in range(len(feature_vals)):
        results[feature_vals[idx]]=score_vals[idx]

    return results
```

Indexing

- Downloading the metadata.csv file of the 181,000 scholarly articles proved to be particularly tricky to begin with as the file was so big. Downloading and navigating Elasticsearch was difficult in the beginning which led me to the use of python for much of the coding.

  Nonetheless, once the JSON files were imported in, I utilised the first 1000 documents as instructed to carry out a concise and smaller initial experiment.

- The following line of code shows the way I loaded in the data:

```
elastic_search = Elasticsearch()
load = pd.read_csv ('metadata.csv')
for num in range(0,999):
    try:
        article_body =
json.load(open(files.iloc[num]['pdf_json_files']))['body_text']
        string = str(json.dumps(article_body))
        all_text = ""
        for k in string.split("\n"):
            all_text += re.sub(r"[^a-zA-Z0-9]+", ' ', k)
```

- This code in which we load the data into the Elasticsearch and then apply the indexing and tokenization with the string function and the formulae at the end.

Sentence Splitting, Tokenization and Normalization

- To complete a list of words for tokenization we sort them into relevant words as per the TF-IDF vector that we will see later and then we create a 'keywords' vector which considers the 10 most relevant number of words.

```
#sort words into relevance

        sorted_items=sort_coo(tf_idf_vector.tocoo())

#select number of words to take into account, in this case the top 10

        keywords=extract_topn_from_vector(feature_names,sorted_items,10)
```

- Next is a part of the tokenization in which we apply functions, so we have the most important words on top. The next part is an important function and code in which we select the most important words and apply it.

```
# function to flip the list so most important is on top
def sort_coo(coo_matrix):
    tuples = zip(coo_matrix.col, coo_matrix.data)
    return sorted(tuples, key=lambda x: (x[1], x[0]), reverse=True)

# function to help select the most important words
def extract_topn_from_vector(feature_names, sorted_items, topn=30):
    """get the feature names and tf-idf score of top n items"""

```

Selecting Keywords

- Our indexing process extends to transforming tokens by identifying the key words and in this this way I managed to clean the data so as to *identify* the relevant words and sort them so they can be useful in our querying and indexing.
    - The first steps we take as can be seen in the code below is to load in the list of stop words that we want to use. Words that do not hold any weight or are not useful to us are dropped in this code.

- In the next step we transform 'allH text' by applying the same counter vector to our data and thus we have returned object with only the most desirable words.

```
#retrieve stop words that are common on most scholarly articles and particularly this one
        with open("stopwords.txt", 'r', encoding="utf-8") as f:
            stopwords = f.readlines()
            stop_set = set(m.strip() for m in stopwords)
            stopwords=frozenset(stop_set)


#get rid of stop words
        cv=CountVectorizer(stop_words=stopwords)


#fit counter vector to get most used words
        word_count_vector=cv.fit_transform([all_text])
        feature_names=cv.get_feature_names()


        tfidf_transformer=TfidfTransformer(smooth_idf=True,use_idf=True)
        tfidf_transformer.fit(word_count_vector)
        tf_idf_vector=tfidf_transformer.transform(word_count_vector)

```

IDF =Log[(# Number of documents) / (Number of documents containing the word)] and

TF = (Number of repetitions of word in a document) / (# of words in a document)

- Quite simply, now that we have a refined bag of words, we must employ the statistical model TF-IDF as noted above so can provide context to it. TF-IDF helps us understand how useful a word is to a sentence, to a document and helps ignore words that are misspelled.

Stemming or Morphological Analysis

- For the stemming process we had to extract the base form of the words by removing the affixes from them. This simply helped us reduce the size of our index and increase the retrieval accuracy of our search.
- The main issue in this step was finding the right stemming algorithm of the many that I could have used. Ultimately, I opted for the Lancaster stemming algorithm, LancasterStemmer.

```
#algorithim to stem the most common words, then place them into an array

        lancaster=LancasterStemmer()

        final_body_words = []
        for k in keywords:
            final_body_words.append(lancaster.stem(k))

#hello
```

- With the for function, I placed the key words in the Lancaster stem, with the key words being a number of number of words to consider, which in my case was the top 10.

Searching

- We begin in this section by making the dictionary into a json info and uploading to Elasticsearch.

```
74          json_info = {"Title":load.loc[num]['title'],"Date":load.loc[num]
            ['publish_time'],"journal":load.loc[num]['journal'],"Authors":load.loc[num]
            ['authors'],"Url":load.loc[num]['url'],"Words":final_body_words}
75
76          elastic_search.index(index="documents",id=num ,body=json_info)
77      except:
78          continue
79
80  print("Input query to look for, words, author or date of publication")
81  input_user = input("Word:")
```

- Next was creating a sort of menu in which we could print the requests of the input user and have it return details for the searched 'word'. I programmed the search to return details such as 'title', 'date', 'author', 'URL', and so on.

```
80  print("Input query to look for, words, author or date of publication")
81  input_user = input("Word:")
82  try:
83      search = elastic_search.search(index="documents", body={"query": {"match": {"Words": input_user}}})
        ['hits']['hits'][0]['_source']
84      print("Details of search: " + "\n" + search['Title'] + "\n" + search['Date'] + "\n" +
        search['journal'] + "\n" + search['Authors']+ "\n" + search['Url'])
85  except:
86      try:
87          search = elastic_search.search(index="documents", body={"query": {"match": {"Author":
            input_user}}})['hits']['hits'][0]['_source']
88          print("Details of search: " + "\n" + search['Title'] + "\n" + search['Date'] + "\n" +
            search['journal'] + "\n" + search['Authors']+ "\n" + search['Url'])
```

- Similar to this code returning out hits of our index, if it did not, I simply programmed a print function to return "Not Found".

Discussion

- Considering I was extremely clueless to begin with, the coding became a little bit easier as things went on and I continued focusing on the lecture materials which were of extreme help to me. A few other websites were extremely helpful, and I based most of my code off which can be found in the references below.

  Ultimately, in the future I would perhaps build the code and the search engine a different way, a more methodological way in which the functionality is a bit clearer, and the pathway is a bit clearer. Although my first time, it does seem to be a bit muddled up and the line of thought is hard to understand at times. Nonetheless I tried to cover all the bases and hope the search engine operates perfectly as it did for me.

References

- https://medium.com/@igorkopanev/a-brief-explanation-of-the-inverted-index-f082993f8605

- https://medium.com/analytics-vidhya/tf-idf-term-frequency-technique-easiest-explanation-for-text-classification-in-nlp-with-code-8ca3912e58c3

- https://kb.objectrocket.com/elasticsearch/how-to-use-the-search-api-for-the-python-elasticsearch-client-265

- https://medium.com/@igorkopanev/a-brief-explanation-of-the-inverted-index-f082993f8605