CE706 - Information Retrieval 2021

Assignment 1

2003930

Instructions for running your system (Engineering a Complete System)

Download metadata.csv from:

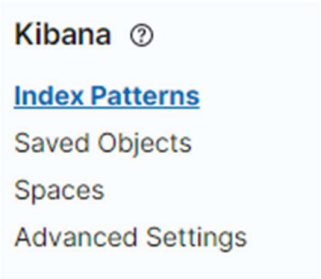
https://www.kaggle.com/allen-institute-for-ai/CORD-19-research-challenge

Once this has been downloaded put it in the same folder as "assignment.py". I didn't upload the csv file to Faser due to the size of it.

To run and setup this program first open cmd/terminal and locate to the folder where both the file "assignment.py" and "metadata.csv" are located. Before running the file ensure that both Elasticsearch and Kibana are running. The program assumes that Elasticsearch is running on localhost port 9200. If this is not the case, then please change this in the python file. The python file takes two arguments. The first is the field you wish to search in and the second is the parameter you wish to search for. For example, if I wanted to search for the term covid in the title I would run the program using "python assignment.py title covid". Run the python file entering your search requirements to index the documents. If the documents have all been indexed, then it will display "Indexed 1000 items".

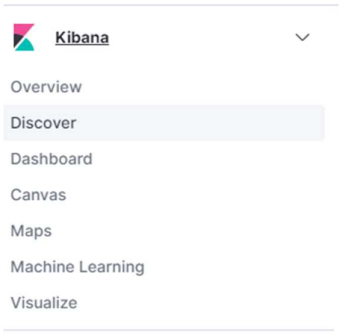
To view these documents in Kibana You need to create an index pattern from the mapping the python file provides. To do this in Kibana go to the management tab and select "stack management".

Then under Kibana select the index patterns tab.

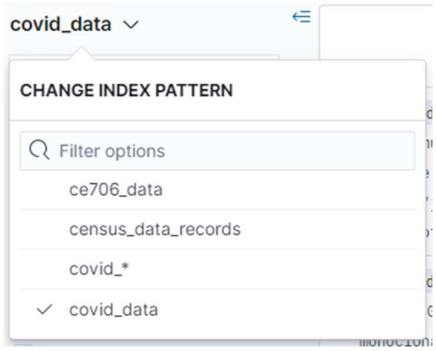


We then need to create an index pattern. To do this press the blue create index pattern button on the top right of the page. In the index pattern name text field enter “covid_data” as shown below.

Then click through the rest of the options to create the index pattern. Once it has been created go to the discover section in Kibana



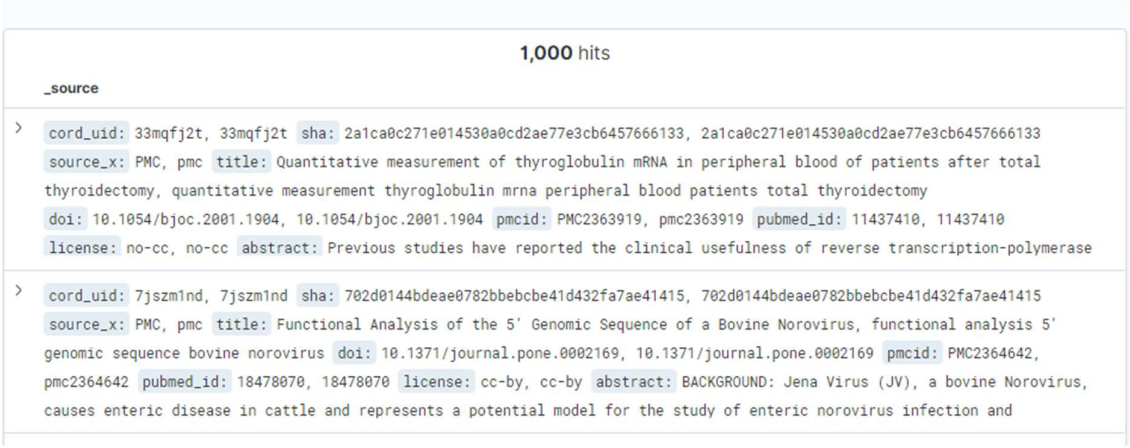
Once here change the index pattern to the one labelled “covid_data”. This will contain the 1000 items which have been indexed.



Indexing

The dataset used for this experiment was downloaded from https://www.kaggle.com/allen-institute-for-ai/CORD-19-research-challenge . The file used for indexing was ‘metadata.csv’. As per the instructions I indexed 1000 documents. The 1000 I selected was simply just the first 1000 in the document. For each entry indexed a count in a loop increases and when it reaches 1000 it stops indexing documents. Before I was able to index the documents, I had to create a mapping. This required me to form the outline of a schema so that the documents could be inputted into elastic search. This required me to define the data type for each field. For simplicities sake I defined them all as text fields. Most of the fields are strings so this makes sense and allows for easy searching later (there will be no need to convert datatypes). When indexing I had to ensure that the first row of the csv file was identified as the column titles. To index each document, I used the ElasticSearch.index() method. If the system has been set up correctly (as described

above) then there should be 1000 documents indexed as can be viewed in Kibana below.



Sentence Splitting, Tokenization and Normalization

This step required me to transform the indexed text into a normal form. To do this I first decided that I would remove all punctuation and that I would also lower the case of all the text as to not include capitals. To process the text, I created a function called "sentence_tokens_nostopwords" This function first converts the input text to lowercase. It then tokenises text into sentence tokens as required. I done this using NLTK and the sent_tokenize method it provides. It then removes the stop words from each sentence as is required by the next part and then returns these as a list of tokens.

```
def sentence_tokens_nostopwords(s):
    s = s.lower()
    tokens = sent_tokenize(s)
    no_stopwords = []
    for sent in tokens:
        no_stopwords.append(' '.join(w for w in word_tokenize(sent) if w not in stop))
    return no_stopwords
```

This function is called during the stemming part of the program which I will describe later.

Selecting Keywords

The first step to this task was removing the stop words. This can be seen in the "sentence_tokens_nostopwords" function above. The list of stop words I used was from the NLTK.corpus module. Each word of the sentence is only indexed if it does not occur in the stop words list.

```
punctuation = list(string.punctuation)
stop = stopwords.words('english') + punctuation
```

When indexing Elasticsearch automatically uses TF-IDF. So, I ensured this was enabled by setting the similarity function to BM25 which included TF-IDF. I also then calculated it for myself for each document that gets indexed. This can be seen below. First, I ensure that the index is not 0. As this is the headers which we do not want to calculate. Then I define the variable TF_IDF as a dictionary. With the key being the token and the index, it belongs to and the value being the TF-IDT score. To ensure I didn't calculate the same word twice I ensured I only used the unique tokens by using NumPy. The dictionary returned from applying for formula then contains the tokens and their scores which can be used in weighting.

```
if index != 0:
    tokens = tokens_nostopwords(val)
    counter = Counter(tokens)
    token_count = len(tokens)
    for token in np.unique(tokens):
        tf = counter[token] / token_count
        df = freq(token)
        idf = np.log((1000 + 1) / (df + 1))
        tf_idf[index, token] = tf * idf
```

Stemming or Morphological Analysis

When testing different stemming methods, I found that it would be best to instead of stemming apply Lemmatisation on the text. When stemming it may not produce an actual word. In many examples you see that the last letter in words is cut out. This is due to it using the "stem" of a word. For example, if I was to apply stemming to the world "trouble" it would return "troubl" which is not what we want. By applying lemmatization, it looks for the actual word as supposed to just the stem. However, the downside to this is that it increases the runtime slightly. But as we are only indexing 1000 items this is not noticeable. To apply this, I used the WordNetLemmatizer from NLTK in a function which can be seen below.

```
def stemming(s):
    stemmed = [s]
    s_tokens = sentence_tokens_nostopwords(s)
    for word in s_tokens:
        stemmed.append(lemmatiser.lemmatize(word))
    return stemmed
```

The first thing to note here is that this is the function that formats the text which gets indexed. This calls the sentence tokenization function I created above to format the text into a normal form. When indexing I decided I would index the original text along with the tokenized and stemmed versions. As you can see above the raw input is added into the array first. For example, for the title of a document it will index like this.

```
t  title          Bench-to-bedside review: Angiopoietin signalling in critical il
                  lness – a future target?, bench-to-bedside review angiopoietin
                  signalling critical illness – future target
```

Here you can see that the raw text appears first. Then is followed by the sentence tokened and stemmed version. The function above first saves the raw text, then tokenises the sentences and removes the stop words as set out in the "sentence_tokens_nostopwords" function. Then for each word in the sentence it lemmatizes it. To perform the stemming as required by this section.

Searching

There are two ways of searching the indexed items. This can be done either in Kibana or in terminal when running the code. To do this in Kibana I go to the dev tools section. For this example, we are going to search the title field for the value "salmonella". To do this we would create a query as can be seen below. This query returns two values showing the full document for each of these values which is far to long to fit onto one page, but the top can be seen below. As you can see the term salmonella appears in the title.

```
GET /covid_data/_search
{
  "query": {
    "match": {
      "title": "salmonella"
    }
  }
}
```

```
      },
      "title" : [
        "Diversity of Salmonella spp. serovars isolated from the intestines of water buffalo calves
          with gastroenteritis",
        "diversity salmonella spp",
        "serovars isolated intestines water buffalo calves gastroenteritis"
      ],
```

The other method to search is how the one I have created in the python code. As I mentioned in the instructions for running this program the two arguments supplied in the terminal/cmd when running this program are used for searching through the documents. This allows for the user to search through any field just as if they were using Kibana. For this I will use the same search parameters as above. To run the search feature using these parameters we enter the terminal "python assignment.py title salmonella". This will then return several things. The first is the total number of results found. In this case it is 2. It then displays the ID of each item, along with the field being searched for. So, in this case it will show the title field. If we changed to search through the abstract, then this is what would be displayed. It also then shows the TF-IDF score. The higher scored items will be displayed first. The result of this can be seen below. If there are no results for a given search term, then a message will be displayed.

```
Total Results  2

(ID: K_e7-HcBskwqDHgUCr0A)
Search Result: ['Diversity of Salmonella spp. serovars isolated from the intestines of water buffalo calves with gastroenteritis', 'diversity salm
onella spp', 'serovars isolated intestines water buffalo calves gastroenteritis']
TF-IDF Score: 8.091593

(ID: qfe6-HcBskwqDHgUybrp)
Search Result: ['Attenuated Salmonella choleraesuis-mediated RNAi targeted to conserved regions against foot-and-mouth disease virus in guinea pig
s and swine', 'attenuated salmonella choleraesuis-mediated rnai targeted conserved regions foot-and-mouth disease virus guinea pigs swine']
TF-IDF Score: 7.164301
```

To do this I created two functions. The first was to complete the search query and the second was to format the results into a more user-friendly output. The search function takes advantage of the Elasticsearch.search function to search through a given index. The query used is the same that gets used in Kibana.

```
def search(field, parameter):
    field = field.lower()
    parameter = parameter.lower()
    query = {
        "query": {
            "match": {
                field: parameter
            }
        },
        "fields": ['cord_uid', 'title', field]
    }
    result = es.search(index='covid_data', body=query, size=1000)
    # pprint(result)
    return result


def format_search(results, field):
    result_data = [doc for doc in results['hits']['hits']]
    if len(results['hits']['hits']) == 0:
        print("No search results")
    print("Total Results ", len(results["hits"]["hits"]))
    for doc in result_data:
        print("\n(ID: %s)" % (doc['_id']) + "\nSearch Result: %s" % (doc['_source'][field]) + "\nTF-IDF Score: %s" % (
            doc['_score']))
```