CE306 – Information Retrieval

Assignment 1

1804162

Contents

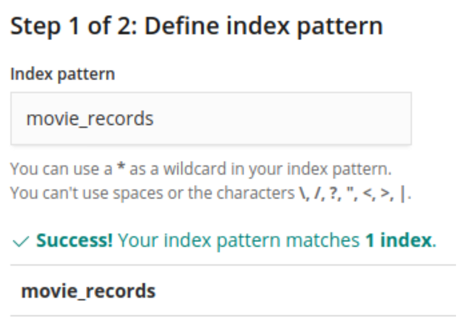
How to Run the System (Requirements and Instructions)

Requirements:

- The code has been developed on a Linux system, so may not work on Windows or Mac devices.
- The user will need to have the Elasticsearch search engine on their system, as well as the Java 8 Runtime Environment. These should be installed to the same directory. The Instructions assume the user is using Elasticsearch version 6.5.1.
- The Python Elasticsearch client must be installed in order to run the Python code. This can be installed from the command line using the following command:
  pip install elasticsearch
- A copy of the "wiki_movie_plots_deduped.csv" file from Kaggle will be required in the same directory as the Python script. This has been provided, however the link to this file can be found here: https://www.kaggle.com/jrobischon/wikipedia-movie-plots?select=wiki_movie_plots_deduped.csv
- The user should also have Kibana installed to the same directory as Elasticsearch and the JRE if they are wishing to use it (NOTE: The version of Kibana MUST match the version of Elasticsearch).

Instructions:

1. Run the Elasticsearch search engine from the command line by navigating to the directory that it and the JRE are located, and using the following command:
   ```
   JAVA_HOME=/jre1.8.0_111 PATH=$JAVA_HOME/bin:$PATH elasticsearch-6.5.1/bin/elasticsearch -d
   ```
2. Run the Python script. This can be done from the terminal using the command:
   ```
   python CE306_Assignment1_Code_1804162.py
   ```
   This script will attempt to establish a connection to the Elasticsearch search engine by pinging the search engine. A connection between the Python client and the Elasticsearch search engine can be established once the search engine is running.
3. Follow the on-screen prompts within the terminal.
4. If the user is wishing to use Kibana, then that can start Kibana by using the following command:
   ```
   JAVA_HOME=/jre1.8.0_111 PATH=$JAVA_HOME/bin:$PATH kibana-6.5.1-linux-x86_64/bin/kibana &
   ```
5. Once Kibana is running, open your browser and enter the address:
   http://localhost:5601/
6. The user will have to create an Index Pattern in order to search the indexed data. Do so by entering '**movie_records**' when prompted, as seen below.

   Step 1 of 2: Define index pattern

   Index pattern

   movie_records

   You can use a * as a wildcard in your index pattern.
   You can't use spaces or the characters \, /, ?, ", <, >, |.

   ✓ Success! Your index pattern matches 1 index.

   movie_records

   After this select '**Next step**', and then select '**Create index pattern**'. You can now navigate to the Discover section of Kibana to enter search queries. By default, Kibana will search the entire index. However, you can add fields to search in on the left in order to restrict the search.

Indexing

For the purpose of this assignment, I have sampled the first 1000 documents from the CSV file listed within the Requirements. This sample size can be changed within the script by simply changing the following line to the desired amount of documents:

```
if (index <= 1000):
```

As the first row of the CSV file will be the headers for the fields within the index, '**index**' will be initialized to '**0**' so that the value in the above code is purely relating to the documents.

The script first tries to open the CSV file within python. If successful, a reader will load the elements of the file into a Python dictionary. Any existing index will be deleted (there shouldn't be an existing index when the script runs for the first time), and then a new index is created.

The line of code below deletes any existing index using the same name:

```
es.indices.delete(index=index_name, ignore=[400, 404])
```

In the above code, 2 exceptions are ignored. The first relating to an exception indicating that the index already exists; and the second relating to an exception given if it tries to delete an index that does not exist.

The following code is then used to create the new index:

```
es.indices.create(index=index_name, body={
    'settings': {
        'index': {
            'number_of_shards': 1
        }
    }
}, ignore=400)
```

Specifying the **'number_of_shards**' means that search queries will not be split up within Elasticsearch. Although this can be less efficient on larger indexes, only 1000 documents are being indexed, so this is not an issue. Doing this also means that search results and scoring is replicable if done within a single shard.

The data from the reader is then indexed row-by-row, with the first row being set as the headers for the index fields. The code for indexing the documents from the CSV file can be seen below:

```
for row in reader:
    entries = {}
    for i, content in enumerate(row):
        entries[headers[i]] = content
    es.index(index=index_name, doc_type=doctype, body=entries)
```

Sentence Splitting, Tokenization and Normalization

I had originally used the NLTK package in order to tokenize the data, identify and split sentences, and normalize the text. Below is the code for splitting sentences, and the code for splitting word tokens:

Sentence Splitting:

```
new_content = []
tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')
split_sentences = tokenizer.tokenize(content)

for s in split_sentences:
    s = s.lower(); #Normalizes sentence text to lowercase
    temp = ""
    for char in s:
        if char not in string.punctuation:
            temp += char
    new_content.append(temp)
return(new_content)
```

Word Tokenization:

```
new_content = []
word_tokens = word_tokenize(content)
for w in word_tokens:
    w = w.lower()
    if w not in string.punctuation:
        new_content.append(w)
return(new_content)
```

However, this ended up taking a long time for the script to do, so during development I instead decided to use a custom analyser for the purposes of this script; as opposed to using a different library such as NLTK. The reason behind this is that Elasticsearch allows custom analysers to be built and used when mapped to a field within the index; without having to reindex the data, edit/add entries before indexing, or create a second index.

It is also much more efficient, as Elasticsearch uses a default set of analysers when searching documents; so running custom versions of these to fit the documents is generally better. The following code segment is used to create the custom analyser I will be using:

```
"analysis":{
    "analyzer":{
        "custom_analyzer":{
                "type":"custom",
                "tokenizer":"classic",
                "filter":[
                    "lowercase",
                    "english_stop",
                    "english_stemmer"
                ]
        }
    }
}
```

The analyser uses Elasticsearch's '**classic**' tokenizer to tokenize the document text, and also makes use of a filter to normalize the tokens.

The '**lowercase**' filter sets all tokens to lowercase if possible, so they are treated equally regardless of capitalization.

The '**english_stop**' filter can be found within the Selecting Keywords part of the report, and the '**english_stemmer**' filter can be found within the Stemming or Morphology Analysis part of the report.

Selecting Keywords

I had originally done part of this step (stop word removal) using the NLTK package; the code for which can be seen below:

```
new_content = []
stop_words = set(stopwords.words('english')) #Defining list of stop words
word_tokens = word_tokenize(content)

or w in word_tokens:
    if w not in stop_words:
        new_content.append(w)
return(new_content)
```

However, as within the Sentence Splitting, Tokenization and Normalization stage I had decided against using NLTK, and instead opted to use Elasticsearch's analysis features.

As I have decided to code my text pre-processing using Elasticsearch's mapping and analysis features, I have used this in order to create a filter which will remove all English stop words during a search.

The code for the stop word removal part of the script can be seen below:

```
"filter":{
    "english_stop":{
        "type":"stop",
        "stopwords":"_english_"
    }
}
```

I am also using the analysis feature as a more efficient way to implement TF-IDF in order to apply correct weightings to the search results.

The equation for calculating TF-IDF is:

$$\textit{Term Frequency in a Document} * \log\left(\frac{\textit{Total Number of Documents}}{\textit{Number of Documents Containing the Term}}\right)$$

The code for implementing this can be seen in the following code segment:

```
"similarity": {
    "scripted_tfidf": {
        "type": "scripted",
        "script": {
            "source": "double tf = doc.freq;
                       double idf = Math.log(field.docCount/term.docFreq);
                       return query.boost * tf * idf;"
        }

    }
}
```

In relation to the equation, the code variables are as follows:

- '**doc.freq**' = The frequency of the term in a document within a specified field
- '**field.docCount**' = The number of documents that have data for the current field
- '**term.docFreq**' = The number of documents containing the term.

'**query.boost**' simply tells the analyser how weighted the result should be. Before TF-IDF, this is set to '**1**'.

Stemming or Morphology Analysis

As within Selecting Keywords and Sentence Splitting, Tokenization and Normalization, I had originally used NLTK in order to stem index entries. However, that again involves directly editing the content stored within the index; which is not a practical way to do so. The code for the original function can be seen below:

```
word_tokens = word_tokenize(content)
ps = PorterStemmer()
    for w in word_tokens:
        w = ps.stem(w)
```

Because of the above reason, I have again decided to use Elasticsearch's analysis feature in order to create a filter which will stem words to their root forms.

The code for the stemming part of the script can be seen below:

```
"filter":{
    "english_stemmer":{
        "type":"kstem"
    }
}
```

I have decided to use '**kstem**' as my chosen stemmer, as opposed to others like '**porter_stem**' (or '**PorterStemmer**' as had been previously used) as it is less aggressive on how it stems the word. Although in most cases it acts the same, it also includes a small dictionary which it can use to correctly stem irregular words, and differentiate between words with similar roots but different meanings.

Searching

Searching is performed within the script by using pre-built query bodies, using the user's input and field selection as the items to be searched. The following code is used to search the index:

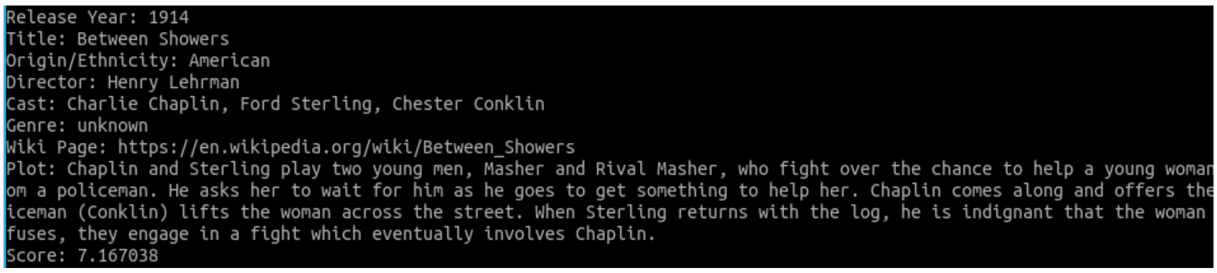
```
res = es.search(index="movie_records", doc_type="movie_record", from_ = 0, size = 3,
                body={
                    "query": {
                        "match": {
                            field : user_input
                        }
                    }
                })
```

The user is first asked to select a field from the document ('**field**'), and then is asked to enter a term to search ('**user_input**'). This code then returns the 3 results with the highest score from the index, and stores them within the variable '**res**'.

The search query could be done manually line-by-line within the terminal, however using pre-built queries and substituting in the user's input makes searching much more modular; allowing the queries to be done with only a simple input from the user.

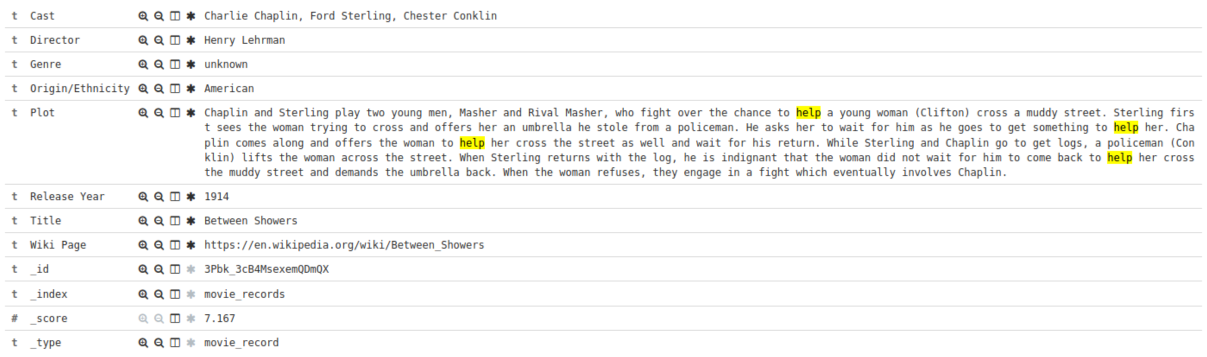
After searching, the system then outputs the total number of documents found from the search, before returning all fields belonging to the top 3 highest scoring documents to the user. The user is then again asked to enter a field to search.

Searching for the term '**help**' within the field '**Plot**' using the script's text-based program returns the following result:



```
Release Year: 1914
Title: Between Showers
Origin/Ethnicity: American
Director: Henry Lehrman
Cast: Charlie Chaplin, Ford Sterling, Chester Conklin
Genre: unknown
Wiki Page: https://en.wikipedia.org/wiki/Between_Showers
Plot: Chaplin and Sterling play two young men, Masher and Rival Masher, who fight over the chance to help a young woman
om a policeman. He asks her to wait for him as he goes to get something to help her. Chaplin comes along and offers the
iceman (Conklin) lifts the woman across the street. When Sterling returns with the log, he is indignant that the woman
fuses, they engage in a fight which eventually involves Chaplin.
Score: 7.167038
```

Kibana can also be used to search these queries. As opposed to entering the field in which to search, the field is instead selected and added to a filter of fields to search in (as described in the Instructions). Searching for the same term in the same field as the above program in Kibana yields the following result:



```
t  Cast              Charlie Chaplin, Ford Sterling, Chester Conklin
t  Director          Henry Lehrman
t  Genre             unknown
t  Origin/Ethnicity  American
t  Plot              Chaplin and Sterling play two young men, Masher and Rival Masher, who fight over the chance to help a young woman (Clifton) cross a muddy street. Sterling firs
                     t sees the woman trying to cross and offers her an umbrella he stole from a policeman. He asks her to wait for him as he goes to get something to help her. Cha
                     plin comes along and offers the woman to help her cross the street as well and wait for his return. While Sterling and Chaplin go to get logs, a policeman (Con
                     klin) lifts the woman across the street. When Sterling returns with the log, he is indignant that the woman did not wait for him to come back to help her cross
                     the muddy street and demands the umbrella back. When the woman refuses, they engage in a fight which eventually involves Chaplin.
t  Release Year      1914
t  Title             Between Showers
t  Wiki Page         https://en.wikipedia.org/wiki/Between_Showers
t  _id               3Pbk_3cB4MsexemQDmQX
t  _index            movie_records
#  _score            7.167
t  _type             movie_record
```