Project of NYTimes Article Search Engine

Abstract

In the project of New York Times Article Search Engine, the steps are as follows: grab data from NYTimes Article API and store the necessary snippet of the metadata into Elasticsearch database; in part two, create a web page using Django framework to present the result of searching and calculate the similarity among the result of these articles with data visualization .

Introduction

With the development of network technology, the industry of computer science is increasingly dependent on the `database management system` (DBMS). The DBMS is expected to allow users create new databases and specify their schemas, using a specialized data-definition language, support the storage of large amounts of data, allowing efficient access to the data for queries**[1]**. The system also provides various functions that allow entry, storage and retrieval of large quantities of information and provides ways to manage how that information is organized.

The aim of this project is to apply what was learned from the course of data management, such as knowledge of `Relational Database`, `NoSQL`, and `Resilient Distributed Dataset`, to practice. Following such directions, we are able to design a search engine for New York Times article query and provide the interface for users to interact with the database. In the storage and search section, I use `Elasticsearch` instead of NoSQL, since the former one is the expert at searching and provides faster speed for query.

Part I: Procurement of Metadata into Local Storage

NYTimes API

The NYTimes Archive API, used in the Search Engine, returns an array of NYT articles for a given month, going back to 1851. As described in the developer page, the user simply pass the API the year and month, and it returns a JSON object with all articles for that month.

To make sure there is enough articles stored in the database, articles of the first three months this year are selected for the further filter, and then merged as one integrated dataframe `result`.

```
In [3]: # Download NYT Archive News From Jan to March in 2019

        url = "https://api.nytimes.com/svc/archive/v1/2019/1.json?api-key=0sOlXxZ5NPwPFmM4ooDWu3Gu7Pdvhcub"
        url2 = "https://api.nytimes.com/svc/archive/v1/2019/2.json?api-key=0sOlXxZ5NPwPFmM4ooDWu3Gu7Pdvhcub"
        url3 = "https://api.nytimes.com/svc/archive/v1/2019/3.json?api-key=0sOlXxZ5NPwPFmM4ooDWu3Gu7Pdvhcub"
        response = requests.get(url)
        response2 = requests.get(url2)
        response3 = requests.get(url3)
        docs = json.loads(response.content.decode('utf-8'))['response']['docs']
        docs2 = json.loads(response2.content.decode('utf-8'))['response']['docs']
        docs3 = json.loads(response3.content.decode('utf-8'))['response']['docs']
        docs_df = pd.DataFrame(docs)
        docs2_df = pd.DataFrame(docs2)
        docs3_df = pd.DataFrame(docs3)
```

```
In [111]: ## Merge three dataframes to one df and display the first 5 rows of df

          frames = [docs_df, docs2_df, docs3_df]
          result = pd.concat(frames, sort = False)
          result.index = range(len(result))
          result.head()
```

Out[111]:

	_id	blog	byline	document_type	headline	keywords	lead_paragraph	multimedia	news_desk	print_page	...	slideshov
0	5c2d52db3a125f5075c029ae	[]	{'original': 'By NEIL GENZLINGER', 'person': [...	article	{'main': 'Daryl Dragon, of the Captain and Ten...	[{'name': 'persons', 'value': 'Dragon, Daryl (...	He and his wife, Toni Tennille, were one of th...	[{'rank': 0, 'subtype': 'xlarge', 'caption': N...	Obits	10	...	
1	5c2d38833a125f5075c02974	[]	{'original': 'By CHRISTINA ANDERSON', 'person'...	article	{'main': 'Where Doulas Calm Nerves and Bridge ...	[{'name': 'glocations', 'value': 'Sweden', 'ra...	In Sweden, midwives deliver babies. But doula ...	[{'rank': 0, 'subtype': 'xlarge', 'caption': N...	Foreign	6	...	

Head of the Merged Dataframe

Modules and Packages used

In the project, modules, such as `numpy`, `string`, `collections`, `re`, `json`, `requests`, `seaborn`, `matplotlib`, `base64`, and `io`, are imported from the python standard library. Packages of `pandas` (for constructing the dataframe), `nltk` (for Lemmatization), and `elasticsearch` (for storage of metadata and search) are installed in addition.

Filter rules of the section and topic

For convenience, the most popular section (which contains most topics in quantity) "U.S." is chosen, and only columns of the article id, headline, keywords, web_url are kept as attributes of a table.

In this process, the filtered dataframe of `newdata` turns out to have duplicates with the same id, so the repeated tuple should be removed and 1540 entries remain.

In [5]:
```
## Filter the section of NYT News with U.S. label

newdata = result.loc[result['section_name'].str.contains('U.S.'), ['_id','headline', 'keywords', 'web_url']]
newdata = newdata.drop_duplicates('_id','first',inplace=False)
newdata.index = range(len(newdata))
newdata.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1540 entries, 0 to 1539
Data columns (total 4 columns):
_id         1540 non-null object
headline    1540 non-null object
keywords    1540 non-null object
web_url     1540 non-null object
dtypes: object(4)
memory usage: 48.2+ KB
```

In [53]:
```
## The duplicate found in the filtered dataframe

nds = newdata.loc[newdata['_id'].str.contains('5c37f60b3a125f5075c03d57'), ['_id','headline', 'keywords', 'web_url']]
nds
```

Out[53]:

	_id	headline	keywords	web_url
347	5c37f60b3a125f5075c03d57	{'main': 'At White House, Empty Desks and Unpa...	[{'name': 'subject', 'value': 'United States P..	https://www.nytimes.com/2019/01/10/us/politics...
351	5c37f60b3a125f5075c03d57	{'main': 'At White House, Empty Desks and Unpa...	[{'name': 'subject', 'value': 'United States P..	https://www.nytimes.com/2019/01/10/us/politics...

Filter of the Section and duplicates

Furthermore, the frequencies of keywords stored in the NYT article metadata are counted in descending order and the top ranked topic turn out to be "United States Politics and Government". The corresponding process of filter is planned to be done during storing in Elasticsearch.

Data cleansing and Lemmatization

Before being stored in the Elasticsearch, the article body of the original `newdata` should be cleaned from html syntax, while the cleaned body needs to be separated into word by word and be stored as a word list after `lemmatization` and remove of `stopwords` for the sake of further similarity comparison among the query results.

Besides, keywords need to be extracted from the metadata provided by NYT API, then are to be divided word by word, and finally integrated as a keyword list for one particular article. The corresponding methods of extract_articleBody, extract_keywords, and get_words are presented as follows.

```
## Function of extracting article body from uncleaned html file

def extract_articleBody(response):

    article = response.text[response.text.find('<section
name="articleBody"'):response.text.find('<div class="bottom-of-article">')]
```

```
    left = -1
    right = -1
    s = ''
    for i in range(len(article)):
        if article[i] == '<':
            left = i
            if left != right+1:
                s += article[right+1:left]
        elif article[i] == '>':
            right = i
    if right != len(article)-1:
        s += article[right+1:len(article)]

    return s
```

```
## Function of extracting keywords from the list

def extract_keywords(kw_str):
    kw_list = []
    for i in range(kw_str.count('value')):
        kw_str = kw_str[(kw_str.find('value')+1):len(kw_str)]
        kw_list.append(kw_str[(kw_str.find(':')+3):(kw_str.find("',"))])
    return kw_list
```

```
## Use Regular Expression to remove capital words and punctuation mark

import nltk
nltk.download('stopwords')
nltk.download('wordnet')
from nltk.corpus import stopwords
cachedStopWords = stopwords.words('english')
from nltk.stem import WordNetLemmatizer
wnl = WordNetLemmatizer()

def get_words(text):
    words_box = []
    text = text.translate(str.maketrans('','',string.punctuation))
    text = text.lower()
    words_box.extend(re.split(r"[“”‘–\&\s]\s*", text))
    words_box = [word for word in words_box if word not in cachedStopWords and
word != '']
    words_box = [wnl.lemmatize(w) for w in words_box]
    return words_box
```

Storage of articles in Elasticsearch

For this part, the search engine called Elasticsearch based on the Lucene is introduced at first. It provides a distributed, full-text search engine with an HTTP web interface and schema-free JSON documents. As described by the developer, Elasticsearch allows the user to store, search, and analyze big volumes of data quickly and in near real time. It can also be used as NoSQL Database, especially for the aim of document storage. However, unlike a relational design with proper normalization, when the name of attribute is changed, we should design our mappings and store our documents such that it's optimized for search and retrieval**[3]**.

In Elasticsearch, the index corresponds to the database in relational database, the type name vs. the table, the document vs. the row, the field vs. the column, and the mapping vs. the schema. Therefore, the index of "nyt_articles" is firstly constructed as the database.

In [8]:
```
from elasticsearch import Elasticsearch

es = Elasticsearch()
result = es.indices.delete(index='nyt_articles', ignore=[400, 404])
result = es.indices.create(index='nyt_articles', ignore=400)
print(result)
```

```
{'acknowledged': True, 'shards_acknowledged': True, 'index': 'nyt_articles'}
```

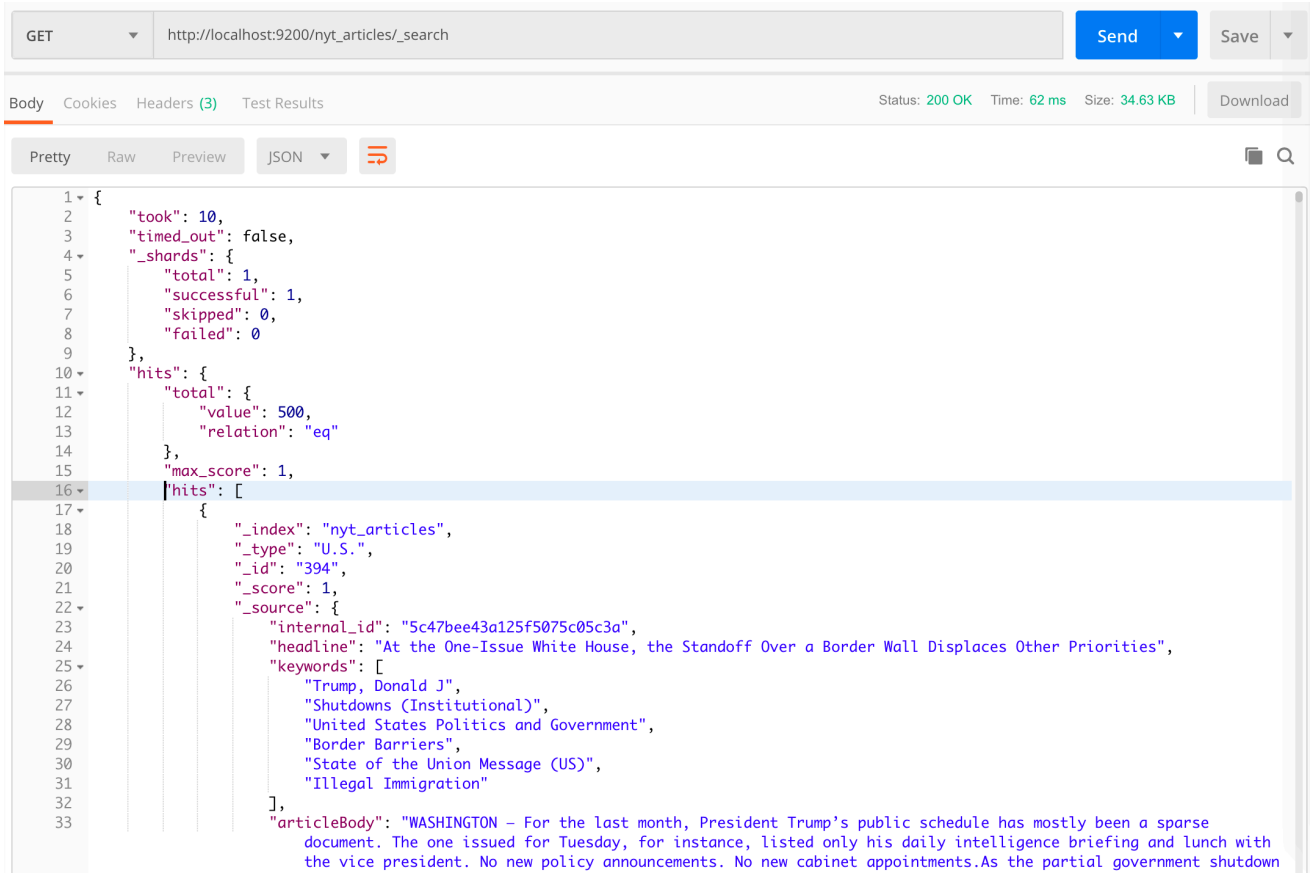
Set up of Elasticsearch and create a index of "nyt_articles"

After that, we need to process the article body, keywords, and the word list while storing the necessary data into Elasticsearch. Meanwhile, the cleaning-up and extraction of article body from the NYT server using the web_url should move on to the data of next row satisfying the requirement of keywords `United States Politics and Government` in case the process of procurement meets failure on some certain row.

```
## Store id, headline, article body, keywords, and url of 500 articles into ES

count = 0
for i in range(len(newdata)):
    kw_str = str(newdata['keywords'][i])
    if kw_str.find('United States Politics and Government'):
        if len(newdata['web_url'][i]) == 0:
            continue

        ## Extract the keywords
        kw_list = extract_keywords(kw_str)

        ## Extract the article body
        response_text = requests.get(newdata['web_url'][i])
        text = extract_articleBody(response_text)
```

```
        ## Insert into the ES
        count += 1
        data = {'internal_id': newdata.at[i,'_id'], 'headline':
newdata['headline'][i]['main'], 'keywords': kw_list, 'articleBody': text,
'web_url': newdata.at[i, 'web_url'], 'words': get_words(text)}
        nyt_es = es.create(index='nyt_articles',doc_type='U.S.',id=count,
body=data)

    if count == 500:
        break
```

Using `Postman`, a Chrome App for interacting with HTTP APIs, it presents with a friendly GUI for constructing requests and reading responses. The screenshot of the stored JSON filed is as follows.



Part of the JSON object as the Database

Challenges of Part I

1. Since there might faces some difficulties obtaining or processing the article body, the procurement of article body using `web_url` should have done with storing in a list of article body before the data of certain columns of metadata is integrated into one `data` JSON object. The most obvious examples are listed such as failure to acquire the body due to the type of video news or visualized news, and insufficient RAM caused by using inefficient extracting syntax in the function while extracting the actual article body.

2. Also, due to unfamiliarity of regular expression grammar, the imperfect syntax used in the function of `extract_articleBody` resulted in incompleteness of the article body, though it would not affect the efficiency of query in the second part except for the presentation of the highlight of the article snippet.

Part II: Keyword Search and Output of Similarity

In this part, in order to construct a `GUI` (Graphical User Interface) of the search engine, I prefer to use Django as the Python-based web framework interacting with Elasticsearch. Therefore, I will follow such steps from the guideline of *ElasticSearch with Django the easy way* **[4]**:

> 1. Setting up ElasticSearch on our local machine and ensuring it works properly
> 2. Setting up a new Django project
> 3. Bulk indexing of data that is already in the database
> 4. Indexing of every new instance that a user saves to the database

The design of query in Elasticsearch

Thanks to strong search function in Elasticsearch, the query using Python can be easily achieved by Python Elasticsearch Client. As long as the

```
from datetime import datetime
from elasticsearch import Elasticsearch
es = Elasticsearch()

doc = {
    'author': 'kimchy',
    'text': 'Elasticsearch: cool. bonsai cool.',
    'timestamp': datetime.now(),
}
res = es.index(index="test-index", doc_type='tweet', id=1, body=doc)
print(res['result'])

res = es.get(index="test-index", doc_type='tweet', id=1)
print(res['_source'])

es.indices.refresh(index="test-index")

res = es.search(index="test-index", body={"query": {"match_all": {}}})
print("Got %d Hits:" % res['hits']['total']['value'])
for hit in res['hits']['hits']:
    print("%(timestamp)s %(author)s: %(text)s" % hit["_source"])
```

Example Usage

The following code of class `ElasticObj` is designed to function as collect query words in `collect.py` as a part of Django project. In the function of `QueryData`, body is filled with the query word passing from the front end, the limited range of `articleBody`, and the highlight syntax for presenting the result in html file.

```
class ElasticObj(object):
    def __init__(self,ip):
        self.client=Elasticsearch(
            ip
        )

    def QuertData(self,str):
        response=self.client.search(
            index= "nyt_articles",
            body={
                "query":{
                    "match":{
                        "articleBody": str
                    }},
                "size":100,
                "highlight":{
                    "pre_tags":['<span class="keyWord">'],
                    "post_tags":['</span>'],
                    "fields":{
                        "articleBody":{}
                    }
                }
            }
        )
        return response
```

As the result of Elasticsearch, the process of the query could be near real time, then Elasticsearch passes the JSON object of the `hits` (search result) to `search.py`, and finally the result would display by the html page within the framework of Django.

Definition of Similarity

The similarity could be defined as easily as the common words in both articles, or as hard as understanding text meanings using the technique of NLP. Within my understanding, the definition of similarity between two articles could be set the frequencies of common words excluding stop words, since the section and topic have been limited in a certain range. Therefore, it might be less likely to meet with a circumstance of similar results from different topics.

According to the guideline of my definition, I take an approach of common words with higher absolute value corresponding to much more similarity at first. However, the result of similarity is not satisfactory as expected due to different lengths of articles in the query result.

```
In [118]: def rank_similarity(query_result):
              N = len(query_result['hits']['hits'])
              if N == 0:
                  return None
              elif N > 10:
                  N = 10
              D = [[0 for i in range(N)] for j in range(N)]
              for i in range(N):
                  s1 = set(query_result['hits']['hits'][i]['_source']['words'])
                  for j in range(i+1,N):
                      s2 = set(query_result['hits']['hits'][j]['_source']['words'])
                      D[i][j] = len(s1 & s2)
              return D

          D=rank_similarity(query_result)
          max_similarity = max(max(row) for row in D)
          D_dataframe = numpy.matrix(D)
          i, j = numpy.where(D_dataframe == max_similarity)
          s1 = set(query_result['hits']['hits'][int(i)]['_source']['words'])
          s2 = set(query_result['hits']['hits'][int(j)]['_source']['words'])
          print(s1 & s2)
          print('Article ' + str(i+1) + ' and article ' + str(j+1) + ' are most similar: ' + str(len(s1&s2)) + ' words')

          {'appearance', 'doesn't', 'bill', 'going', 'south', 'one', 'hillary', 'language', 'pull', 'young', 'every', 'began',
          'involved', 'used', 'question', 'monday', 'cnn', 'crime', 'away', 'medium', 'need', 'secure', 'taken', 'social', 'wo
          uld', 'attempt', 'senior', 'clinton', 'u', 'campaign', 'fund', 'someone', 'within', 'published', 'donald', 'sense',
          'background', 'didn't', 'drew', 'come', 'change', 'since', 'senator', 'sander', 'person', 'state', 'saying', 'curren
          t', 'political', 'influence', 'presidential', 'member', 'early', 'claim', 'three', 'voter', 'woman', 'opponent', 'se
          cretary', 'case', 'country', 'ok', 'senate', 'take', 'around', 'course', 'others', 'white', 'law', 'gun', 'think', '
          may', 'public', 'worker', 'ultimately', 'could', 'center', 'drawn', 'policy', 'ever', 'many', 'attention', 'increase
          ', 'made', 'back', 'day', 'action', 'make', 'base', 'entire', 'former', 'connected', 'trump', 'role', 'really', 'dis
          trict', 'worked', 'run', 'year', 'time', 'referring', 'president', 'vice', 'came', 'race', 'court', 'jr', 'sure', 'h
          elped', 'statement', '30', 'democrat', 'part', 'organization', 'help', 'trying', 'number', 'we've', 'portion', 'cand
          idate', 'week', 'last', 'well', 'home', 'work', 'several', 'see', 'represent', 'mr', 'internal', 'meeting', 'cut', '
          want', 'seen', 'speaking', 'justice', 'position', 'program', 'like', 'needed', 'announced', 'view', 'start', 'added'
          , 'police', 'across', 'american', 'decision', 'point', 'never', 'night', 'another', 'first', 'interest', 'criminal',
          'stuff', 'interview', 'new', 'way', 'important', 'right', 'leader', 'seems', 'effort', 'running', 'becoming', 'alrea
          dy', 'don't', 'two', 'long', 'that's', 'got', 'founder', 'support', 'press', 'type', 'asked', 'politically', 'house'
          , 'party', 'among', 'started', 'beat', 'democratic', 'also', 'official', 'people', 'rather', 'prosecutor', 'york', '
          ago', 'show', 'said', 'i'm', 'believe', 'fellow', 'know', 'say', 'showed', 'career', 'witness', 'it's', 'speech'}
          Article [4] and article [7] are most similar: 203 words
```

First Approach of Similarity

After the first trial, I attempt to use the result of double the intersection divided by the sum of lengths of two articles to express the weight of common words for both articles. The relative quantity turns out to be better than the first one.

As for the common words, the function of intersection between two sets of word boxes could be easily achieved. The same is the function of counting intersection length. As a side note, the word list of every articles is stored in Elasticsearch after processing with lemmatization, which prevents from possible invalid and repeated words.

```
## Similarity of two articles in the result
# Pick top 10 articles
N = len(cleareSponse)
if N > 10:
    N = 10
D = [[0 for i in range(N)] for j in range(N)]

# Find the intersection of the word lists of two articles
for i in range(N):
    s1 = set(cleareSponse[i]['source']['words'])
    for j in range(i+1,N):
        s2 = set(cleareSponse[j]['source']['words'])
        D[i][j] = len(s1 & s2) * 2 / (len(s1) + len(s2))
        D[j][i] = D[i][j]
```

```
# Form the relational matrix into a dataframe
# Use matplotlib to plot the heatmap of 10*10 relations
d = pd.DataFrame(D)
cols = [i for i in range(1,11)]
sns.set(font_scale=2)
fig, ax = plt.subplots(figsize=(12, 9))
ax = plt.axes()
mask = np.zeros_like(d, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True
ax.set_title('Similarity Between 2 Articles in 10 Results').set_size(23)
hm = sns.heatmap(d, cbar=True,annot=True, mask=mask, square=True, ax = ax,
fmt='.2f', annot_kws={'size': 20}, yticklabels=cols, xticklabels=cols)
```

Implementation of Front End and Back End

Within my immature understanding of the relation between the back end and the front end, the presentation of the results of By means of the Django framework, the strings of `hits`, `result_count`, `last_seconds`, `page_nums`, `message`, and `my_html` extracted from `collect.py` and `search.py` could pass to the front end, and then the query result and graph generated by python `matplotlib` library could be displayed on the web page using html language.

```
## Pass the query words from the request by collect.py to the search function
def search(request):
    request.encoding = 'utf-8'
    es = ElasticObj(["127.0.0.1:9200"])
    if 'q' in request.GET:
        response = es.QuertData(request.GET.get('q'))
        message=request.GET.get('q')
        clearesponse=[]
        count = 0
        for hit in response['hits']['hits']:
            cleares={}
            cleares['source'] = hit['_source']
            cleares['source']['headline'] = cleares['source']['headline']
            if "articleBody" in hit['highlight']:
                cleares['source']["articleBody"]=''.join(hit["highlight"]
["articleBody"])
            else:
                cleares['source']['articleBody'] = ' '.join(cleares['source']
['articleBody'])
            cleares['index']=hit['_index']
            cleares['type'] = hit['_type']
            cleares['id'] = hit['_id']
            cleares['score'] = hit['_score']
```

```
            cleares['source']['web_url'] = cleares['source']['web_url']
            cleares['source']['words'] = cleares['source']['words']
            clearesponse.append(cleares)
            count += 1
            if count >= 10:
                break
        result_count=response['hits']['total']['value']#
        last_seconds = response['took']/1000.0#
        page_nums=int(result_count/10)+1  #
    else:
        message = 'The keyword is empty!'
```

```
# Output the plot to the front end
img = io.BytesIO()
fig.savefig(img,format='png',bbox_inches='tight')
img.seek(0)
encoded=base64.b64encode(img.getvalue())
my_html = '<img src=" {}">'.format(encoded.decode('utf-8'))

## Return strings of the result for rendering
return render(request,"result.html",{"all_hits":clearesponse,
                                     "result_count":result_count,
                                     "last_seconds":last_seconds,
                                     "page_nums":page_nums,
                                     "message":message,
                                     "my_html":my_html})
```

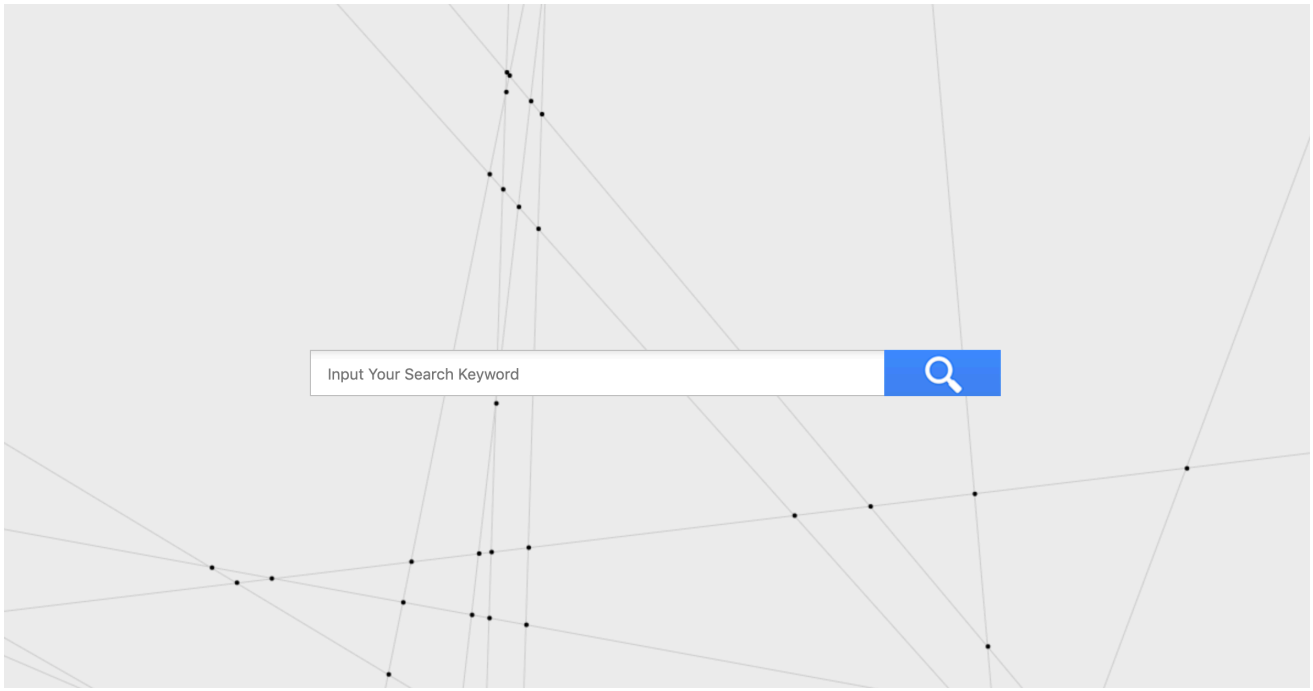
Challenges of Part II

1. The scope of search is too elementary to be limited in the article body. The optimization could be done by extending the query field to keywords, and the project would be better if the priority of search could be set as the rank of first headline, then keywords, and finally article body.
2. The implementation of similarity by my definition is not meticulous enough, and therefore counting the frequency of every word in the intersection should be taken into consideration using the module `Counter`. As a result, the accuracy of the similarity would be more precise. More important, there exists a dilemma in understanding the context meaning. For example, the similarity of the sentences "She is a child" and "She is not a child" is approaching to extremely close by my method. However, two sentences tell stories with opposite ideas. Therefore, more advanced methods in NLP are needed for improvement.

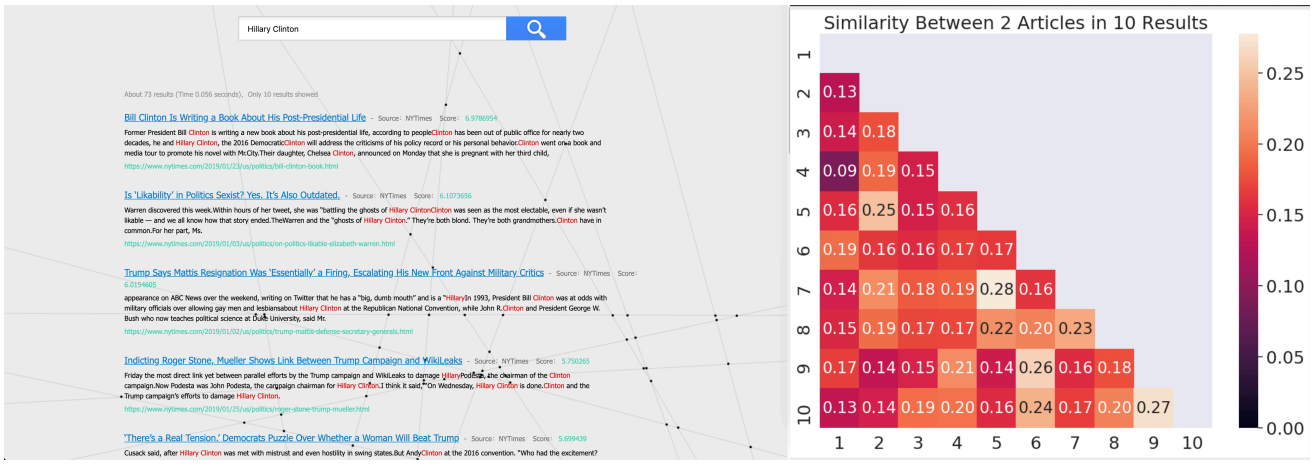
Conclusion and Future Work

Results

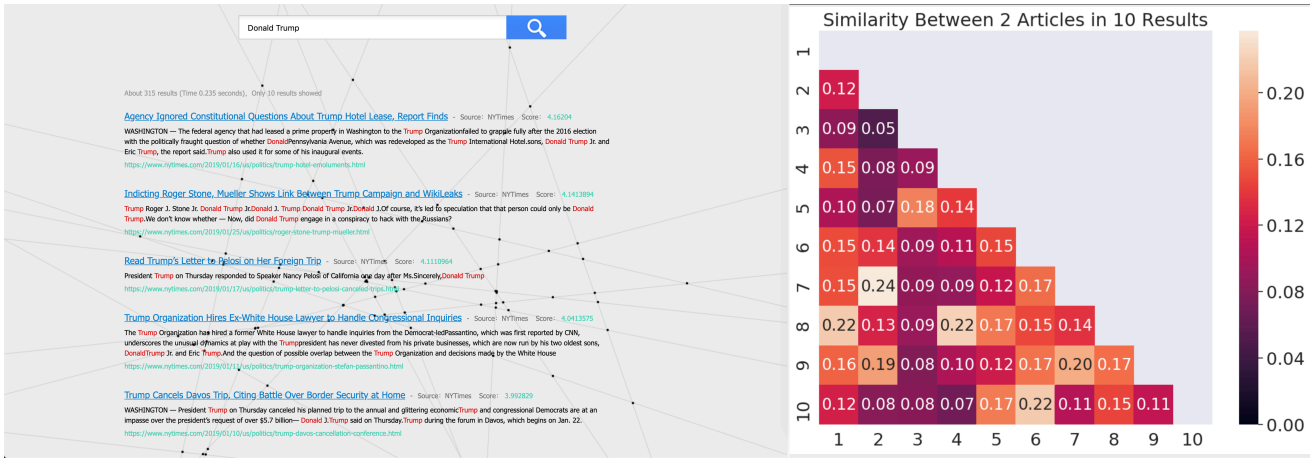
By virtue of designed web framework published online, the initial search form is showed as follows. Multiple words are allowed for input due to Elasticsearch's property.



Search Interface



Query Result of Hillary Clinton



Query Result of Donald Trump

For simplicity of presentation in the report, I converged two functions of search and similarity in one picture. In fact, the similarity graph follows after the search results. The screenshot of search result could be observed by zooming in. The score next to each headline represents the degree of relevance with the query words computing by Elasticsearch.

At the right side of the screenshot, the larger figures, the more relevance between two articles. In the first graph, article 5 and article 7 are most relevant, and article 2 and article 7 are the same in the second graph.

Improvement

Except for the challenges mentioned before, there are three more aspects needed for improvement.

1. The code of both storage and back end could be further simplified. For example, the function of `rank_similarity` should be written in an independent python file so that it could be imported as a module in the `search.py` in the back end.
2. The algorithms of data cleansing, storing in Elasticsearch, and calculation of similarity could be improved in time and space complexity, although the implementation of search in ES is near real time.
3. The user interface could be further beautified. For instance, the way of presentation may mimic the two-column structure of Google.

References

[1] Database. *Wikipedia*. *Terminology and overview*. Available: https://en.wikipedia.org/wiki/Database

[2] Jeffrey D. Ullman, Jennifer Widom. *A First Course in Database Systems*, *3rd* ed. Pearson Group, 2013.

[3] Elasticsearch as a NoSQL Database. *Elastic Blog*, *Relations and Constraints*. Available: https://www.elastic.co/blog/found-elasticsearch-as-nosql

[4] Adam Wattis. ElasticSearch with Django the easy way. *freeCodeCamp*. Available: https://medium.freecodecamp.org/elasticsearch-with-django-the-easy-way-909375bc16cb