Project 4 – PRIMARY CHOICE

💡 **Notion Tip:** Use this template to organize a project kickoff and build out the necessary context. Share this page with other team members by clicking the `Share` button on the top right of this page. Learn more about sharing & permissions here.

Table of contents

Video has Uploaded to Youtube

https://youtu.be/UqAJGebtyto

Author:

Huayi TANG 21101676 Sorbonne M2 RES-INSTA

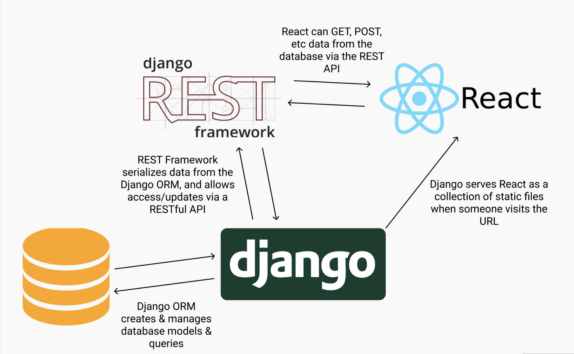
MOHAMED AZIZ ABDERRAHMEN 3802009 Sorbonne M2 RES-INSTA

Astrid PASSOT 21205135 Sorbonne M2 RES-INSTA

▼ General architecture of our project

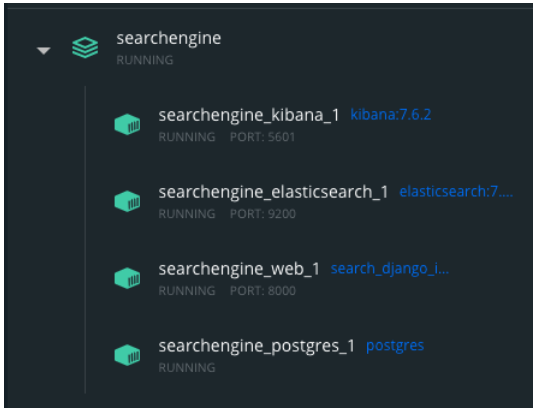
The application can be accessed from any web browser, whether on mobile or not.
Our project is divided into two main entities, the client and the server. Our client is built with React and we use Bootstrap for layout and interfacing. The client is started on port 3000. For the server, we are using a development with Django 4.1.5 and with the help of django DRF, the development methodology we are using is “front and back-end separation”. This also allows us to easily create mobile apps for any future projects, as they can still reuse the backend API.

Our application follows an MVC architecture. Views are managed by the React client, the controller is in the django server, models are serialized objects.



Before the development, we agree on the interaction form and data format of the data interface. Then we realize parallel development of front-end and back-end, in which the front-end can mock test alone after development, while the back-end can also use Httpie interface self-testing, after a module is finished at each part, we begin the functional co-tuning and verification of the format.

And in order to have Environment consistency during the development , we are using Docker Compose, it helps ensure the portability, scalability, isolation , we choose docker compose also considering the deployment reason , it could set up and tear down the entire stack of services, making it easier to test and deploy updates.




```
version: '3'
volumes:
  pgdata: {}
  esdata: {}
services:
  web:
    build:
      context: .
      dockerfile: ./docker/web/Dockerfile
    image: search_django_image # Name of the image
    volumes:
      - .:/app
    depends_on:
      - postgres
      - elasticsearch
    env_file: .env
    ports:
      - "8000:8000"
    command: /start

  postgres:
    image: postgres
    env_file: .env
    volumes:
      - pgdata:/var/lib/postgresql/data

  elasticsearch:
    image: elasticsearch:7.6.2
    volumes:
      - esdata:/usr/share/elasticsearch/data
    environment:
      - discovery.type=single-node
      - ES_JAVA_OPTS=-Xms2g -Xmx2g
    ports:
      - "9200:9200"

  kibana:
    image: kibana:7.6.2
```



```
    depends_on:
      - elasticsearch
    ports:
      - "5601:5601"
```

We have implemented the 2 explicit features defined by the topic: a basic keyword search based on the content of the texts, as well as an advanced search that can use Regular Expressions (Regex) for the keyword search. For the advanced search, we have adapted our DAAR1 project and reimplement it in python : `clone of the egrep command that allows searching by Regex.`

We also implemented the 2 implicit features that is ranking and recommendation, and for the Algo we used pls refer to the `Problem statement and our solution also algorithm explaination` section

In matter off fact apply the KMP and Ahoullman in such a huge dataset is so costly, so we choose to use the elasticsearch as our searching service in order to interact with our front end , but we still got all the Personal Algo function implemented and API defined, and it works fine, **except some of the seach like regex search would take like more than 3 hours to get the results ,** and some times it brings webserver container down, but if you want , thanks to django DRF you could still test our backend through testing the API with base url that we prescribed in section `API CONVENTION`

▼ API CONVENTION AND JSON OBJECT FORMAT

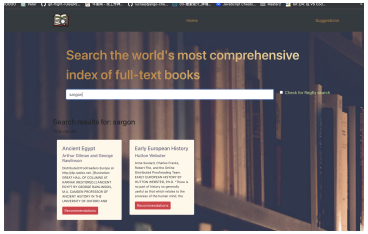
Feature	base URL elastic search(GET Method by defaullt)	base URL School Algo(GET Method by defaullt) DONT ALLOW ANY SPACES	Features
project BASE API	`http://localhost:8000/api`	`http://127.0.0.1:8000/schoolsearch/`	BASE URL
Search by keyword:	`GET /search?q=<keyword>`	`GET /search?q=<keyword>`	`Explicit feature "Search" :` allow you to search for documents by keyword. The `q` query parameter should contain the keyword to search for. The response should be a list of documents whose index table contains the keyword.

Feature	base URL elastic search(GET Method by defaullt)	base URL School Algo(GET Method by defaullt) DONT ALLOW ANY SPACES	Features
Search by RegEx:	`GET /search?regex=<regular_expression>`	`GET /search?regex=<regular_expression>`	`– Explicit feature “Advanced search” :` search for documents using a regular expression. The `regex` query parameter should contain the regular expression to search for. The response should be a list of documents that contain a string matching the regular expression in their index table.
Recommendations based on book_name Reommand top 5 Book that content similar to the given_book_name book's content	`GET / recommendations?book_name=<book name>`	`GET /school_ recommendations?book_name=<book name>`	`– Emplicit feature of recommendation :` Suggestion of documents with a content similar to the Indiacted Book Name. return the top 5 books objects that contents similar to the indicated name book.

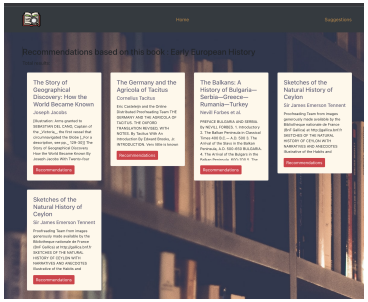
```
{
            "title": "Domestic Animals",
            "author": "Richard Lamb Allen",
            "link": "http://www.gutenberg.org/ebooks/34175",
            "bookshelf": "Animals-Domestic",
            "text": "file was produced from imaxxxxxxx +16000 words"
},
```

▼ Visual Display

Our Mainpage(check the box to launch the regex search)



And click on the recommendation buttuon to get the top 5 similar books



▼ Problem statement and our solution also algorithm explanation:

❓ General Problem Overview

The objective of the project is to create a search engine for a library in the form of a web application. It must be able to efficiently search for books among a large number of available documents, here 1664 books of at least 10 000 characters. We also made sure to implement an ordering criterion for the results, here Jaccrad Similarity.

✅ General Solution Overview

For each of the features, we've offered 2 possible Solutions using different base URL:
1.Using the `django_elasticsearch_dsl` to customize our query to the elasticSeach Container base URL : `http://localhost:8000/api`

2.Write the our own algorithm , retrive all the datas from the PSQL database Container `Book.objects.all()`
base url: `http://localhost:8000/schoolsearch/`
And Using our own algorithm to filter the dataset instead of using query API offered by the Library
Explicit feature “Search” - `KMP`
Explicit feature “Advanced regex search” - `Ahoullman`
Implicit feature of ranking - `By counting occurence while doing the matching`
Explicit feature of recommendation - `Jaccard Similarity`

▼ Data Layer :

one need to collect sufficiently many text documents,The minimum size of the library must be 1664 books. The minimum size of each book must be 10,000 words. `guteberg_download.py`

We've used the python `requests` and `BeautifulSoup` to crawls and retrieves data from the Project Gutenberg (PG) digital library, a collection of over 60,000 free e-books. We uses the PG metadata CSV file `gutenberg_metadata.csv` as the starting point and retrieves the book information, such as author, title, and link, from it. It then uses the book ID to retrieve the text of the books either through the `load_etext` method of the `gutenberg.acquire` library or by scraping the text from the PG website.

The code has a limit of 1664 books to retrieve, which can be adjusted by changing the `DATASET_SIZE` value. After retrieving the book information, the code cleans up the text by removing newlines, carriage returns, tabs, and other funny tokens using the `clean_text` function. The cleaned text is then stored in a dictionary named `data` along with the other information. Finally, the data is stored in a Pandas dataframe , and stores in form of a csv file `gutenberg_data.csv`

5 rows | 5 rows × 6 columns | CSV

	Title	Author	Link	ID	Bookshelf	Text
4	On Snake-Poison: It…	A. Mueller	http://ww…	32947	Animal	[Illustration] ON SNAKE-POISON. ITS ACTION A
3	Birds, Illustrated	Color Photograph…	http://ww…	30221	Animal	FROM: THE PRESIDENT OF THE NATIONAL TEACHERS'
2	Artistic Anatomy of…	Édouard Cuyer	http://ww…	38315	Animal	+----------
1	Deadfalls and Snares	A. R. Harding	http://ww…	34110	Animal	DEADFALLS AND SNARES [Frontispiece: A GOOD DE
0	The Extermination o…	William T. Horna…	http://ww…	17748	Animal	[Illustration: (Inscription) Mr. Theodore Roc

> Down load the full dataset with the following link through our google drive if you want to avoid to run the whole scipts on your machine :)
> https://drive.google.com/file/d/1vZzlSnYT3yakErkdw8yCoA94omwuVqdS/view?usp=sharing

Once the full dataset is in the local desktop, we've populate this data into psql database and index some parts of the data from relational database into Elasticsearch

populate this data into psql database container

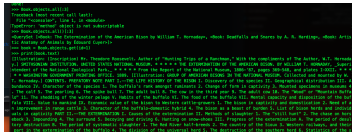
```
#docker-compose run --rm web python manage.py shell# from ta
#Create a model for gutenberg_data.csv#>>> from books.models import Book#>>> Book.populate()
```

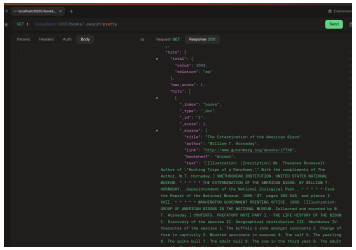
This is in the Django container

index some parts of the data from relational database into Elasticsearch container

```
$ docker-compose run --rm web python manage.py search_index --rebuild
```

Using httpie or `curl -X GET "localhost:9200/books/_search?pretty"`





▼ Explicit feature "Search"

Search documents by keyword. On user input a string S, the application returns a list of books objects whose "text" attributs contains keyword

Using Elastic Search

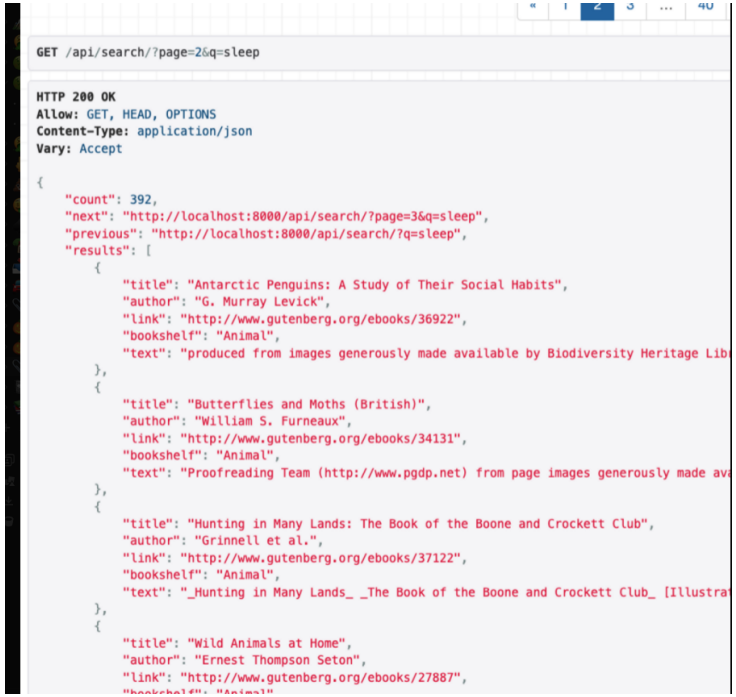
We('ve defined the search API `search_by_keyword` in `search.py` packing our query.

```
tang huayi
def search_by_keyword(keyword):
    s = create_connection_to_index_books()
    # Create the query
    query = Q(
        "multi_match",
        query=keyword,
        fields=["title", "author", "bookshelf", "text"],
        type="cross_fields",
        operator="and"
    )
    # Execute the search
    response = list(s.query(query).scan())
    # Return the results
    return response
```

And then in `views.py` we've defined a `BookListView` and for the coming `request , we get the value of parameter 'q'` as the key word that user input, then we use the `search_by_keyword` API to sent the query to elastic search container and rerturn the results

```
tang huayi
class BookListView(generics.ListAPIView):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
    tang huayi
    def get_queryset(self):
        q = self.request.query_params.get('q')
        if q:
            return search_by_keyword(q)

        regex = self.request.query_params.get('regex')
        if regex:
            return search_by_regex(regex)

        return super().get_queryset()
```

Here is the example of the result



Our Own algo-`books/kmp.py`

Our implementation of the Knuth-Morris-Pratt (KMP) string search algorithm `kmp.py`. The algorithm is used to search for the occurence of a given pattern in a text.

It is an efficient string matching algorithm that uses a pre-processing step to minimize the number of comparisons required to find a pattern in a text.

Our implement consists of two functions:

KMPSearch and computeLPSArray.

The KMPSearch function takes two inputs, the pattern to be searched and the text in which the pattern is to be searched. The function returns the number of times the pattern occurs in the text.

```python
tang huayi *
def KMPSearch(pat, txt) -> int:
    M = len(pat)
    N = len(txt)

    occurence = 0
    # create lps[] that will hold the longest prefix suffi
    lps = [0] * M
    j = 0  # index for pat[]

    # Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps)

    i = 0  # index for txt[]
    while i < N:
        if pat[j] == txt[i]:
            i += 1
            j += 1

        if j == M:
            print("Found pattern at index " + str(i - j))
            j = lps[j - 1]
            occurence += 1


        # mismatch after j matches
        elif i < N and pat[j] != txt[i]:
            # Do not match lps[0..lps[j-1]] characters, th
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1
    return occurence
```

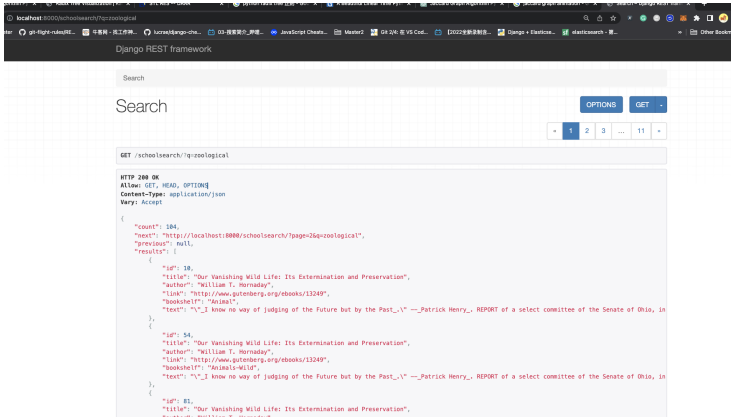
The computeLPSArray function takes three inputs, the pattern, the length of the pattern and the lps array that holds the longest prefix suffix values for the pattern. The function calculates the lps array that is used in the KMPSearch function.


```python
tang huayi *
def computeLPSArray(pat, M, lps):
    len = 0  # length of the previous longest prefix suffix

    lps[0] = 0  # lps[0] is always 0
    i = 1

    # the loop calculates lps[i] for i = 1 to M-1
    while i < M:
        if pat[i] == pat[len]:
            len += 1
            lps[i] = len
            i += 1
        else:
            if len != 0:
                len = lps[len - 1]
            else:
                lps[i] = 0
                i += 1
```

And then in `views.py` we've defined a `SearchView` and for the coming `request, we get the value of parameter 'q'` as the key word that user input, then we use the `KMPSearch` API filter out the book objects from the `Book.objects.all() that contains the keyword we search`

```python
class SearchView(generics.ListAPIView):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
    tang huayi
    def get_queryset(self):
        q = self.request.query_params.get('q')
        if q:
            results = []
            for book in Book.objects.all():
                occurence = KMPSearch(q, book.text)
                if occurence:
                    results.append((book, occurence))
            results.sort(key=lambda x: x[1], reverse=True)
            return [book for book, _ in results]
```

Example result that we use our KMP Algo to luanch the seach



▼ Explicit feature “Advanced search”

Search documents by RegEx. On user input a string RegEx, the application returns : either a list of text documents whose index table contains a string S matching RegEx as regular expression (refer to Lecture 1 of UE DAAR for a formal definition of regular expressions); or a list of text documents containing a string S matching RegEx as regular expression

During the `first bonus project`, we’ve already implement the `aboullman` algorithm by java and our comprehension of this algorithm could be found in our report there, but here we reimplement it using the Python.

Using Elastic Search

We’ve defined the search API `search_by_regex` in `search_py` packing our query.

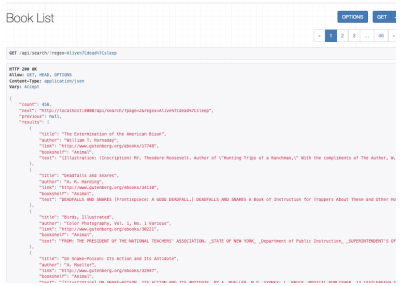
```
tang huayi
def search_by_regex(regex):
    s = create_connection_to_index_books()
    query = Q("query_string", query=regex, default_field='text', analyze_wildcard=True, minimum_should_match=1)
    response = list(s.query(query).scan())
    # Return the results
    return response
```

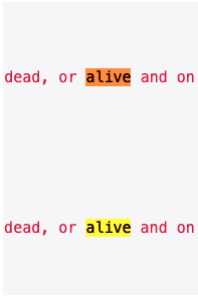

And then in `views.py` we’ve defined a `BookListView` and for the coming `request , we get the value of parameter ‘regex’` as the regex pattern that user input, then we use the `search_by_regex` API to sent the query to elastic search container and rerturn the results

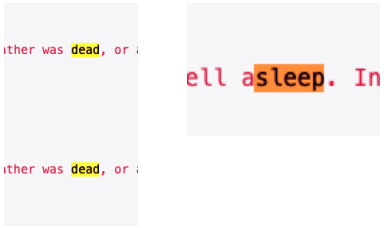
```
class BookListView(generics.ListAPIView):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
    tang huayi
    def get_queryset(self):
        q = self.request.query_params.get('q')
        if q:
            return search_by_keyword(q)

        regex = self.request.query_params.get('regex')
        if regex:
            return search_by_regex(regex)

        return super().get_queryset()
```

Here is the example of the result

`GET /api/search/?regex=Alive%7Cdead%7Csleep`







Our Own algo- `books/ahoullman.py.` tooks hours for a search among 1664 books

Our own algo supports the regex pattern below

```
. matches any single character
* matches zero or more of the preceding element
+ matches one or more of the preceding element
? matches zero or 1 of the preceding element
| matches the preceding element or following element
() groups a sequence of elements into one element
```

The code you could found in `books/ahoullman.py`, the code is too long i'll not shown it here but here i'd explain the logic of our implementation:

Here we use a Non-deterministic Finite Automata(NFA) technique to make the matcher run in linear time efficiency.

Both NFA construction and string testing on NFA will be linear time. Another benefit of our algorithm is that once a pattern is compiled to a NFA, that NFA can be used to test all future strings for that pattern without recompile the pattern every time.

We keep track of NFA's current states in a set *cur_states*

- Initialise the NFA by putting the *Start* state in *cur_states*
- Remove a character *char* from left of the testing string
- For every state in *cur_states*, first remove it from the set, then check to see whether it can consume the given character *char* or not:
  - If it is a normal state (a State with an enclosed character), check whether *char* is equal to the normal state's char or the normal state is a '.' state (a '.' state can match any character):

    - If yes, the *char* can be consumed by this neighbour state. Add its outgoing neighbours to *cur_states*
    - If no, do nothing
  - If it is a special state (either a Start, Empty or Matching state), go check the special state's outgoing neighbours and repeat the same process until reach normal states, then rewind to previous step. Special states consume nothing.
- Rewind to step 2 and keep doing the same process until either *cur_states* becomes empty or all characters in the testing string have been removed. In the former case, return *False*
- Check whether *cur_states* contains the *Matching* state or from those states in *cur_states* whether they can reach the *Matching* state (via *Special* states). If either case is true, return *True*; otherwise return *False*

But our new implementation, the basic idea is about that offering the API as :

- `if_could_find_pattern(self, p , long_text : str)->bool` to check if the book objects text fields contains the regex we match
- `occurences_that_match_pattern(self, p , long_text : str)->int` to couting the occurence that the regex pattern matches contains in order to get our books objects ranked by "the relevance"

In terms of time efficiency, the NFA scans the test string once. In the worst case, the NFA might be in all available states at each step, but this results in at worst a constant amount of work independent of the length of the string, so arbitrarily large input strings can be processed in linear time **O(n)**. If it were using different patterns to test strings, the overall time efficiency of this NFA approach is **O(m*n)**, where the m and n is the length of pattern and test string respectively.

And then in `views.py` we've defined a `BookSuggestionView` and for the coming `request , we get the value of parameter 'regex'` as the regex pattern that user input, then we use the `search_by_regex` API to sent the query to elastic search container and rerturn the results

```python
class SearchView(generics.ListAPIView):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
    def get_queryset(self):
        q = self.request.query_params.get('q')
        if q:
            results = []
            for book in Book.objects.all():
                occurence = KMPSearch(q, book.text)
                if occurence:
                    results.append((book, occurence))
            results.sort(key=lambda x: x[1], reverse=True)
            return [book for book, _ in results]
        regex = self.request.query_params.get('regex')
        if regex:
            results = []
            for book in Book.objects.all():
                sol = Aboullman()
                occurence = sol.occurences_that_match_pattern(rege
                if occurence:
                    results.append((book, occurence))
            results.sort(key=lambda x: x[1], reverse=True)
            return [book for book, _ in results]

        return super().get_queryset()
```

▼ Implicit feature of ranking

Ordering the presentation of the documents returned by above features. In response to a search or an advanced search, the web/mobile application returns the list of documents ordered by relevance,
according to some mathematical definition of ranking : by decreasing `number of occurrences of the keyword/regEx` in the document.

Using Elastic Search - the return object is ordering by default search

Our Own algo- `By decreasing` `number of occurrences of the keyword/regEx` `in the document`

Both our KMP algo and Ahoullman algo defines the API that return the occurence number of the hit books, so that after we get the hit documents, it could be oreders my the occurences.

```python
def get_queryset(self):
    q = self.request.query_params.get('q')
    if q:
        results = []
        for book in Book.objects.all():
            occurence = KMPSearch(q, book.text)
            if occurence:
                results.append((book, occurence))
        results.sort(key=lambda x: x[1], reverse=True)
        return [book for book, _ in results]
    regex = self.request.query_params.get('regex')
    if regex:
        results = []
        for book in Book.objects.all():
            sol = Aboullman()
            occurence = sol.occurences_that_match_patt
            if occurence:
                results.append((book, occurence))
        results.sort(key=lambda x: x[1], reverse=True)
        return [book for book, _ in results]

    return super().get_queryset()
```

▼ Explicit feature “recommendation”

Recommand the top5 similar books to the frontend by frontend indicating the books name.

Using Elastic Search

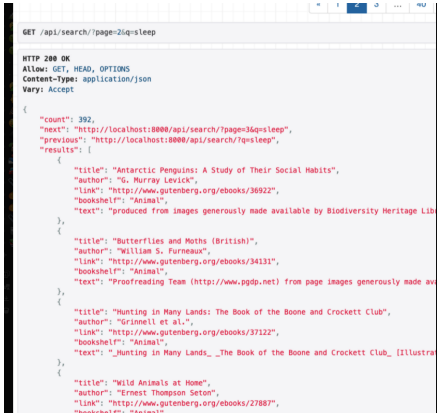
We've defined the search API `search_by_suggestion` in `search.py` packing our query.

```
tang huayi *
def search_by_suggestion(book_title):
    s = create_connection_to_index_books()
    query = Q(
        "match",
        title=book_title
    )
    book = s.query(query).execute()[0]
    print(book.title)
    book_id = book.meta.id
    print(book_id)
    # Create the query
    query = Q(
        "more_like_this",
        fields=["text"],
        like={"_id": book_id},
        min_term_freq=1,
        max_query_terms=12,
    )
    # Execute the search
    response = s.query(query).execute()[0:5]
    # Return the results
    return response
```

And then in `views.py` we've defined a `BookSuggestionView` and for the coming `request , we get the value of parameter ' book_name'` as the `book_name` that user input, then we use the `search_by_suggestion` API to sent the query to elastic search container and rerturn the results


```
tang huayi *
class BookSuggestionView(generics.ListAPIView):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
    tang huayi *
    def get_queryset(self):
        book_name = self.request.query_params.get('book_name'
        if book_name:
            return search_by_suggestion(book_name)
        return super().get_queryset()
```

Here is the example of the result



Our Own algo- `books/jaccard_distance.py`

We defined a class named `JaccardGraph`, which creates a graph data structure using NetworkX library to represent books.

The Strategy Jaccard similarity score is used to measure the similarity between two sets of words. In code, the Jaccard similarity score is calculated between the texts of two books. If the score is greater than 0, it means that the books have some overlapping words, and thus an edge is created between the two books in the graph.

```python
class JaccardGraph:

    def __init__(self, books):
        self.graph = nx.Graph()
        self.load_books(books)
        self.create_edges()

    def jaccard_similarity(self, text1, text2):
        text1_set = set(text1.lower().split())
        text2_set = set(text2.lower().split())
        intersection = text1_set.intersection(text2_set)
        union = text1_set.union(text2_set)
        return len(intersection) / len(union)

    def load_books(self, books):
        self.books = books
        for _, book in self.books.iterrows():
            self.graph.add_node(book['ID'], title=book['Title'],
                                author=book['Author'], link=book['Link'],
                                bookshelf=book['Bookshelf'], text=book['Text'])

    def create_edges(self):
        for book1, book2 in combinations(self.books.itertuples(), 2):
            book1_text = book1[6].lower()
            book2_text = book2[6].lower()
            similarity = self.jaccard_similarity(book1_text, book2_text)
            if similarity > 0:
                self.graph.add_edge(book1[0], book2[0], weight=similarity)
```

We provides two methods for retrieving the top 5 similar books for a given book: `get_top5_similar_books` takes a book ID as input and returns the 5 books with the highest Jaccard similarity score to the input book. `get_top5_similar_books_by_title` takes a book title as input and returns the top 5 similar books based on that title.

```python
def get_top5_similar_books(self, book_id):
    return sorted(self.graph[book_id].items(), key=lambda x: x[1]['weight'], reverse=True)[:5]

def get_top5_similar_books_by_title(self, book_title):
    book_id = self.books[self.books['Title'] == book_title]['ID'].values[0]
    return self.get_top5_similar_books(book_id)
```


▼ Functional Test

> Instead of waiting 6+ hours for 1 test result, we decide to narrow the test dataset down to 10 books, and using jupyterlab to run the algo rythme and check the test result.

```python
import pandas as pd

df = pd.read_csv('../gutenberg_data.csv',nrows=10)

df.head()
```

5 rows × 6 columns

	Title	Author	Link
0	The Extermination of the American Bi…	William T. Hornaday	http://www.gutenber
1	Deadfalls and Snares	A. R. Harding	http://www.gutenber
2	Artistic Anatomy of Animals	Édouard Cuyer	http://www.gutenber
3	Birds, Illustrated	Color Photography, Vol. 1, No. 1 Var…	http://www.gutenber
4	On Snake-Poison: Its Action and Its …	A. Mueller	http://www.gutenber

- Keyword Search

  We decided to use our KMP and Ahoullman algo to the test dataset to conunt the occurence of the key words in each of the 10 book, and compare the result with the result generate by the `count() in python string library`

  What we expected is the result of 'our algo' and 'python count' is exactly the same

  Here we are using the 'animal' as keyword

  Here we could see thats they are exactly the same.

In 24

```python
allo = Ahoullman()
allo_results = np.zeros(df.shape[0])
for i, text in enumerate(df['Text']):
    allo_results[i] = allo.occurences_that_match_pattern(keyword_user,text)

allo_results
```

Out 24

10 rows × 1 columns

	0
0	217.0
1	202.0
2	173.0
3	0.0
4	48.0
5	187.0
6	1.0
7	78.0
8	219.0
9	224.0

```python
import numpy as np
keyword_user = 'animal'

results = np.zeros(df.shape[0])
for i, text in enumerate(df['Text']):
    results[i] = KMPSearch(keyword_user, text)

count_results = df['Text'].str.count(keyword_user).to_numpy()
```

```
count_results
```

```
Found pattern at
```

10 rows

	0
0	217
1	202
2	173
3	0
4	48
5	187
6	1
7	78
8	219
9	224

```
results
```

10 rows

	0
0	217.0
1	202.0
2	173.0
3	0.0
4	48.0
5	187.0
6	1.0
7	78.0
8	219.0
9	224.0

- Regex search

  We decided to use Ahoullman algo to the test dataset to count the occurence of the key words in each of the 10 book, and compare the result with the result generate by the `re.findall(re_pattern, long_text)` in python string library
  What we expected is the result of 'our algo' and '`re.findall(re_pattern, long_text)` is exactly the same

  Here we are using the `buf.alo(e|o)s+` as regex that defined

  Here we could see thats they are exactly the same

```python
import re
regex_pattern = 'buf.alo(e|o)s+'

re_regex_result = len(re.findall(regex_pattern,df['Text'].sum()))
```

```
397
```

```python
allo_regex_result = allo.occurences_that_match_pattern(regex_pattern, df['Text'].sum())
allo_regex_result
```

```
397
```

▼ Conclusion

This project allowed us to delve into the intricacies of search engines, including sorting criteria and the significance of efficient search algorithms.

Our attempts to implement our own search algorithms revealed a challenging aspect that took much longer than anticipated, and our use of Elasticsearch showcased its speed and efficiency, driving home the importance of optimized algorithms. We also utilized Docker in the project, further demonstrating our understanding of its applications.

In future iterations, we aim to improve the performance of our application through pre-processing crawled content and constructing a revert index based on extracted keywords.