Classic Cars (911 Club) report

Batyrev Georgy / Rim Hun

Developed as part of the group project for *Information Retrieval SA 2023*, part of BSc INF at Università della Svizzera Italiana (USI) in Lugano, Faculty of Informatics, Switzerland

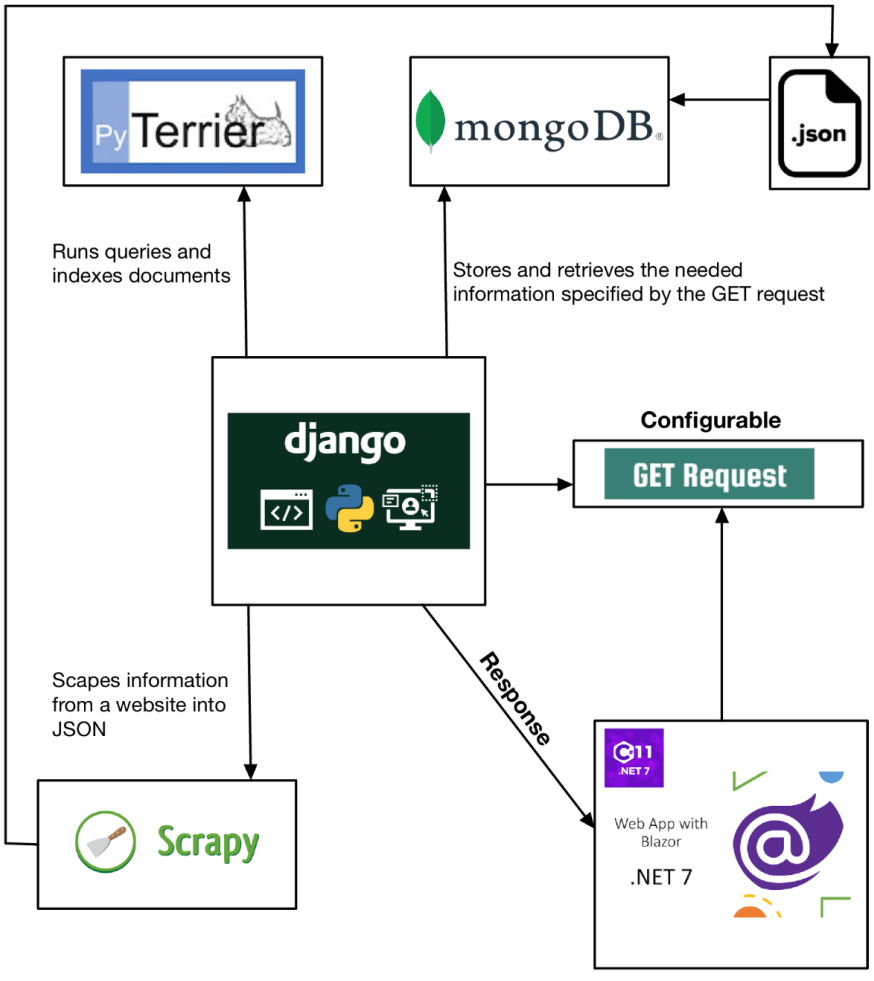
Abstract

This report outlines the development of a vintage cars-for-sale search engine, detailing the technology stack, crawling, indexing, clustering, backend functionality, and frontend presentation. It also includes user evaluation using the System Usability Scale (SUS). The report provides a comprehensive overview of the search engine's technical aspects and development process.

Simple Featured: **Filtering** and **Results Presentation** Complex: **Results Clustering**

1. Tech Stack

In our system, various components function autonomously. It is crucial to comprehend the distinct elements that contribute to the functioning of our system.



1.1 Scrapy

Scrapy is an open-source Python framework for crawling and scraping websites. Our backend invokes the Scrapy process for crawling/scraping a set of vehicles in auction or for sale, categories or search queries. The scraped data output is then directed into a JasonDB.json file, ready to be read and indexed by PyTerrier.

1.2 PyTerrier

PyTerrier is a Python library designed for working with the Terrier (Originally Java) information retrieval platform. Terrier is an open-source search engine that supports experimentation and research in information retrieval. Specifically, application to our project includes features like distributed indexing, replication and load-balanced querying, and automatic recovery. It is responsible for indexing and querying the collection. It retrieves the lost of vehicle objects most pertinent to the user's search query and filters.

1.3 Django + MongoDB

For our backend we chose Django. It handles user queries, and result clustering and interacts with all the other components of our stack. It also handles the scraping and indexing process, as it periodically provides the PyTerrier and Scrapy processes with the search/{filters} to scrape and index. The MongoDB database serves the purpose of storing, deserializing and remembering vehicle information. It performs additional computation on the user's query such as pagination, query reranking, sorting, "more like this' ' similarity feature and input satisfaction. It also adopts some security measures like using a rate limiter to protect from DDOS attacks.

1.4 .NET + C#

Blazor, based on C#, played a crucial role in our project's frontend. It allowed us to create dynamic and interactive web applications using C# and .NET instead of traditional web technologies like JavaScript. With Blazor, we built responsive user interfaces, integrated client-side logic, and facilitated real-time updates. This technology choice provided a seamless and consistent development experience, as we could leverage our C# expertise across both frontend and backend, ensuring efficient communication and data flow between the user interface and server.

2. Retrieving and indexing documents

This section is dedicated to exploring the fundamental aspects of any search engine: the retrieval and indexing of documents. We will provide a detailed explanation of these two critical processes.

2.1 Crawling process

We have crawled 3 different websites which advertise classic vintage cars (“classiccars.com”, “classiccarsforsale.com”, “ERclassic.com”). All the websites are crawled and scraped using the “Spider” library of Python language. There were a few challenges whilst crawling the websites.

The first challenge was finding the right websites. For safety reasons, many websites forbid the access of crawlers and return “403 HTTP Status code (Forbidden access to the resource)”, and even if it is allowed, it restricts the number of requests in a timeframe. The second challenge was the crawling process itself. All of these websites have a dedicated page for individual cars where concise information is displayed and the search tab itself has very minimal information. This meant, that to collect the description the crawler had to go into the dedicated page and come back and continue crawling others. To avoid this issue, we divided the crawling into 2 separate parts. First, we crawl the search tab for all relevant vehicles and link to their dedicated page and save it in a JSON file. Then the link to the dedicated pages are converted into the starting_url in a different spider from the JSON and each of the dedicated pages are read. Next challenge was scraping all the information in our supposed structure of a car on a sale. Our struct of a car had: brand, model, year of production, price, and description of the car. In addition, we added the link to the dedicated page and the URL of that vehicle’s image provided by the website for a better user experience. Some of the websites didn’t specify all the information we wanted, or at least not in a format we could scrape easily. For example, “classiccarsforsale.com” had all the information as the title of the dedicated page, and the title was inconsistent depending on the vehicle. Hence, we had to get the price and year of production from the search tab and get the brand and model of the vehicle from the URL to the dedicated page. There were a few more minor problems such as choosing the efficient starting URL, collecting all information in the correct format etc.

In total, we had 2708 vehicle information from “classiccars.com”, 3569 vehicle information from “classiccarsforsale.com”, and 249 vehicle information from “erclassics.com”. When we implemented the user interface and carried out the “alpha testing” one crucial fact was discovered, it turns out “classiccarsforsale.com” itself is a website scraped from “classiccars.com” and a lot of the vehicle information gathered from it had a dedicated page redirecting to “classiccars.com” page, and cars not present in the “classiccars.com” had its unique dedicated page (assuming some cars are posted on both sites and they merged the database and some cars are only on one of the sites so it has a dedicated page for them).

We mainly chose websites hosted on the UK domain, and there are multiple reasons behind this decision. The UK has a long history for the second-hand car market, especially for vintage cars, hence, it was a good pool to choose from. Second, there wasn’t a language barrier which simplified the assessment of the information we gathered much easier. Lastly, it eliminates the necessity to unite the currency of transactions and improves the user experience. At the end, the results are saved in 3 different JSON files so all three are merged into one file called “jsonDB.json” in the JSONS folder using the “*generateTargetFile()*” function created in the “indexerScript.py” file in utilities folder.

2.2 Handling user queries from Blazer using Django

Our frontend is constructed using Blazer (further explanation in section 3) and backend is constructed using Django. Django is a free and open-sourced web development framework dedicated to the Python language. The reason for choosing a Python-based framework for the backend of our web application other than its solid performance is because our indexing and clustering algorithm is written using python and it reduces the overhead related to integrating functions written in different languages.

Our Django backend accepts one type of API endpoint from the Blazer application, (frontend) which is "/api/vehicles/", which is defined in the "url.py" script in the backend folder. The rest of the information needed to handle the user request is passed as search parameters of the URL. In the search parameter various parameters are starting from vehicle information "brand", "year" (=year of production), "min_price", "max_price" to general parameters like "currentPage" which indicates the page number for the "Paginator" function which returns corresponding vehicle objects depending on "itemsPerPage" parameter which sets how many vehicles should be displayed. Lastly, user query strings can be also passed as search parameters.

All of the information mentioned is passed to "views.py" script which creates a class called "VehicleListView" which extends "ListAPIView" of "rest_framework" which is a toolkit for building Web APIs. When initializing an instance of the "VehicleListView" class, we are passing the "index" and "jsons" as initial parameter of the class. As it can be guessed from the name, "jsons" is the list of JSON objects where each object is the JSON object version of the vehicle information. One thing to be clarified is that the variable name "index" is named wrong. Technically "index" is "retrieval transformation" (according to PyTerrier Documentation) generated with the "index" of our pages and "weighting model", (ranking function) which in our case is the "BM25" relevance-based ranking function. The reason, "index" and "jsons" are passed as initial parameters for "VehicleListView" class is to avoid building the "dictionary" every time there is a new search request from the user. The "retrieval transformation" object built from "dictionary" remains constant as long as the content of DB remains constant, hence, rebuilding it every time is a waste of time, especially because generation of "index" is a very time-consuming process.

From the search parameters and initial class parameters, VehicleListView's "get_queryset()" function filters the list of "Vehicle" objects present in our DB (MongoDB). The function starts from indexing (searches) according to the search query string from the user given through the search bar and clusters the result (which will be further explained in sections 2.4 and 2.5) from the indexed and clustered result, the remaining list of vehicles are filtered by search parameters given by the user (Such as price, brand and year). Then the "Paginator()" function returns 'n'-th 'm' number of vehicles (where n = currentPage, m = itemsPerPage) from the filtered list, which is then returned to the front-end of the application to display it to the user.

2.3 MongoDB

We chose MongoDB as our Database to store all the vehicle information due to its simplicity and our familiarity from the past experience. The information in the MongoDB are imported through MongoDB GUI application from the "jsonDB.json" file which contains parsed scraped information about vehicles (Please refer to README.MD for guide on how to import the json to DB). Vehicle information is parsed according to the model set in the backend (Please look at models.py file in the backend folder).

2.4 Indexing process

As briefly mentioned before, the indexing process is scripted using the “PyTerrier” library which is an extension of the JAVA’s “Terrier” library. The term used for indexing is the “text” field of the Vehicle object. Text field is the concatenation of all the vehicle information, in addition to the description of the vehicle information provided in the websites. In the “generateIndex()” function of the “indexerScript.py” file, the text fields along with the document number (“docno”) is used to generate a document frequency index. Then that index is used to create a retrieval transformation object which can be used to generate a ranking of the documents according to the query phrase given by the user. In our retrieval transformation object, we are using “BM25” (Best Matching 25) relevance ranking model because it is an improved version of TF-IDF model which utilizes pivoted document length normalization method (it rewards shorter documents and penalizes long documents). In the same script, “getQueryResult()” function is also provided, which takes in query phrases, DB as a list, and the retrieval transformation model to return a DataFrame (pandas) of objects ranked according to the query. As mentioned before, the generation of the retrieval transformation model is only done once in the beginning of the backend initialization and recycled for higher efficiency but the getQueryResult function is called every time there is a search request from the user.

(It is important to note scripts for both indexing and clustering (next section) are implemented in the “utilities” folder in “classic_cars” folder)

2.5 Clustering (Complex Feature)

We implemented the result clustering feature using the “sklearn” and “nltk” library. According to the clustering example given in the iCorsi, the cleaning of the text is done right before the clustering. However, we are cleaning the text before putting it in the database for higher accuracy. Hence, the first action performed for clustering in our case was stemming of the text using the stemmer from “nltk” library. There was a suggestion to implement the stemming and save it in the DB without stemming every time but we weren’t 100% sure if there wouldn’t be any impact on the indexing process, hence, it is performed during the clustering process. Then the documents containing the stemmed words are vectorized using the “Vectorizer” function of the “sklearn” library so we can apply the k-means algorithm to create *‘k’* unique set of clusters from the large cluster of documents provided through the indexing. Number of clusters is decided according to the number of brands present in the large cluster.

It is possible to either cluster and index or index and cluster, and we took the approach of later for the implementation of the later model as we believed it would provide a better user experience as this approach will consume both less time and memory, which was very suitable for our case where we have a massive collection of almost 7,000 documents in the database.

However, the problem with this approach was the ranking of the displayed documents. When the documents get clustered, the formerly sorted (ranked) DataFrame from the indexing process is lost, and if we sort it back it loses the point of clustering. Hence, to integrate the two features together, we grouped the vehicles according to the cluster they belonged to obtain their mean (average) indexing score and sorted them as clusters. (Vehicles in cluster with higher average indexing scores were placed higher than vehicles in the cluster with lower average indexing scores) Then vehicles were sorted amongst themselves in the cluster in the order of higher indexing score. All of these maneuvers were implemented very easily thanks to the pandas dataframe.

Overall, we believe we have integrated the result clustering and indexing process very well with a good balance of performance both in time complexity and memory usage.

3. Presenting documents to users: frontend

3.0 Implementation insights

User Interface: Built using Blazor .NET, the front-end presents an interactive and responsive user interface. This interface includes components such as forms, buttons, and data displays, allowing users to interact with the application intuitively. Using the inherent to the framework concept, reflection, single vehicle card is modeled by the class inside the Services folder:

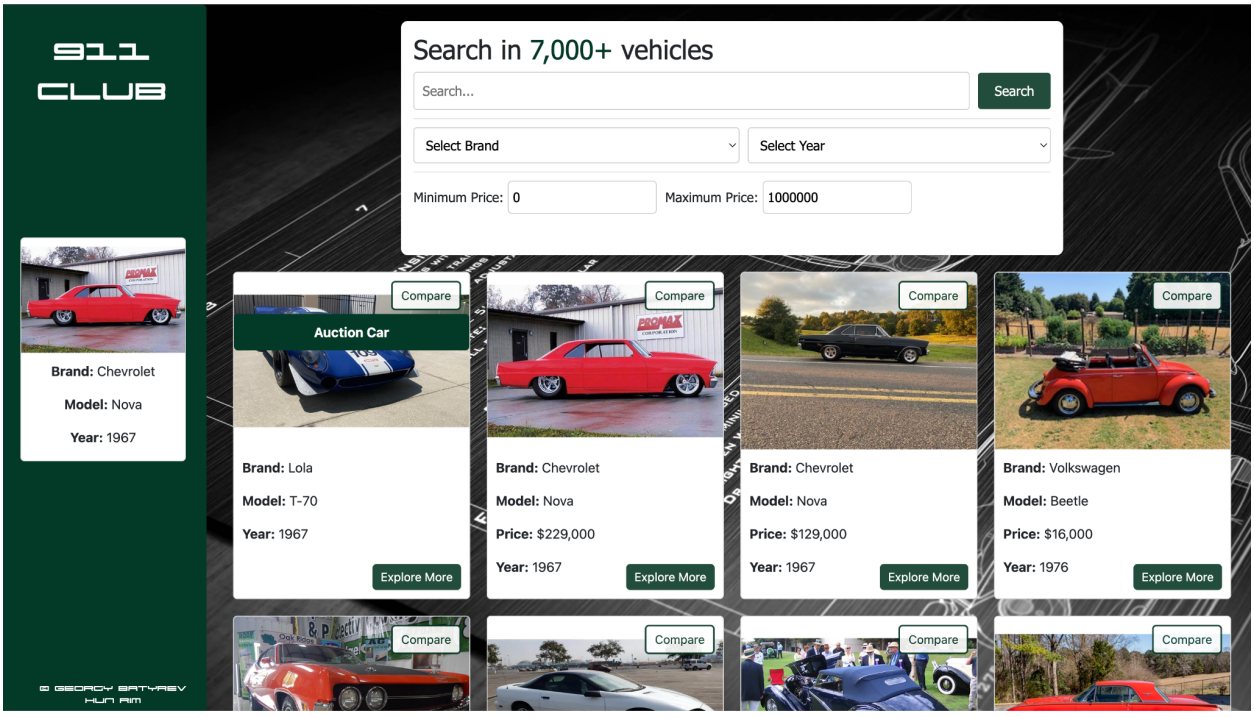
```
public class Vehicle {
        public string Image_url { get; set; }
        public string Brand { get; set; }
        public string Model { get; set; }
        public int Price { get; set; }
        public string Year { get; set; }
        public string Text { get; set; }
        public string Detail_url{ get;}}
```

Index Component: A critical component in Blazor .NET, the Index component, is responsible for creating and sending GET requests to the Django API. This is triggered by user actions like clicking a button search or submitting a filter. Look at *ApplyFilters()* to see the way the GEt request is composed.

3.1 Visual Overview

A search engine is just one component of an Information Retrieval (IR) system. Equally important is the presentation of documents to users, alongside mechanisms that enable them to interact with these documents. This includes conducting searches and navigating through the system in a user-friendly manner. The frontend of the system plays a crucial role in facilitating these interactions.

Image (1), Homepage of the application

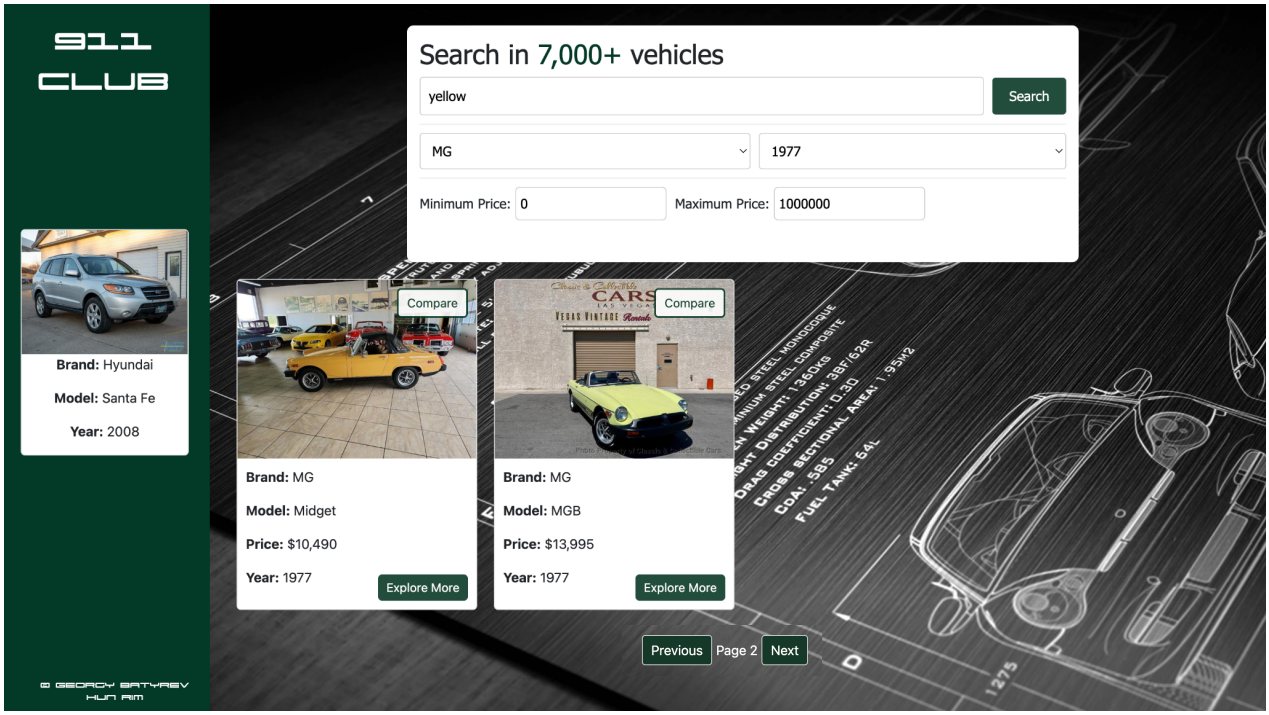


The homepage is designed with a user-friendly interface, featuring an intuitive search bar for query input. For users seeking tailored search experiences, the platform offers filter search options, allowing the refinement of vehicle searches by parameters such as price, brand, and year of manufacture. This filtering feature includes an 'auto-complete' functionality, enhancing ease of use by suggesting options as users begin typing.

The results are shown in a list of cards (see) sorted by the docno* of the documents, which is computed by Pyterrier. Each card contains the most important information about the vehicle. Upon initial loading, the website prominently displays popular vehicle selections. Each listed vehicle is represented as an individual entity, with an “Explore More” link directing users to detailed information on the corresponding website. Notably, vehicles available for auction are uniquely tagged with an “Auction Car” label, indicating their dynamic pricing nature. Additionally, the site integrates a “Compare” feature. This tool enables users to select and save vehicles to a sidebar for convenient comparison against other entries, streamlining the decision-making process. This approach to vehicle browsing and comparison underscores the site's commitment to providing a seamless and efficient user experience.

Within the Django API framework, pagination is efficiently integrated to enhance performance and user experience. This approach limits the display to only 20 entries per page, significantly reducing system overhead and improving the clarity and organization of the information presented. Users can easily navigate through these segmented entries using the tab navigation system located at the bottom of the webpage. This methodical design ensures a streamlined and user-friendly interface for accessing and browsing data.

Image (2), Pagination and filtering



3.2 User evaluation: is the search engine well-built?

Adhering to best practices, it's imperative to evaluate the efficacy of a search engine after its development. The research discussed below was performed during the beta version of our application. We have followed the following scheme (right).

Evaluation done by: Ivan Linnikov, Maria Batalkina, Max Koldoba.

We have utilised the widely recognized System Usability Scale (SUS) questionnaire. This tool comprises a set of 10 questions designed to assess a system's usability. Each question is rated on a 5-point Likert scale, ranging from 'strongly disagree' to 'strongly agree.' The cumulative score, ranging between 0 to 100, reflects the system's overall usability.

In addition to the standard SUS questions, we included two specific queries tailored to our system:

- "Did I find what I was looking for?"
- "Was it clear how to successfully complete all tasks?"

We engaged three users to interact with the system and complete the SUS questionnaire. During their session, they were instructed to execute various tasks, such as searching for a specific brand of the vehicle, clicking links, and comparing the vehicles. The users demonstrated confidence in navigating the system and accomplished the given tasks. They also independently explored the system, particularly in browsing and locating cars of their interest, without any specific prompts from us.

The SUS questionnaire results were insightful. The average score for each question is derived by dividing by three. The initial SUS scores were reasonably high averaging at around 89.5. This score is commendable given the SUS scale ranges from 0 to 100, however, it was highlighted that the speed of querying needed to be improved. It was our initial model, hence, we were creating an index table every time there was a search query and due to its high processing time, it was slowing the performance of our retrieval. To increase the performance, we moved the indexing to be only performed once when the backend is initialized and reutilized that indexing table after that. Hence, the average time for a search query went from approximately 5 seconds per query to approximately 0.5 seconds per query. (Of course, this number would vary depending on the complexity of the query phrase)

Furthermore, the responses to the two additional questions, specific to our search engine, indicated high user satisfaction. The users were pleased with their search outcomes and completed the tasks effectively. Based on the SUS questionnaire outcomes, it's evident that our system is not only highly usable but also meets user satisfaction benchmarks effectively. **Refer to the specific findings in the appendix.** Importantly, acknowledging the complexity of installation we have asked a fellow informatics student Ivan Linnikov to completely install the ecosystem from scratch. The time it took was: 43 minutes. Keep in mind it was a user with little to know .NET and Django experience.

4. Installfest + Build Instructions

4.1 Scrappy

- **Version**: As of my last update, Scrapy 2.6.1 is the latest stable version. **Installation**: Python: Ensure Python is installed (preferably Python 3.7 or above). Install Scrapy using pip: *pip install scrapy==2.6.1.*

4.2 Blazor Web App + Django

- **Blazor:** Use the latest stable version of .NET SDK for Blazor. **Installation**: Download and install the .NET SDK from the official Microsoft website. We have used Visual Studio Code.
- **Django:** Django 4.0 is a recent stable version. **Installation**: Ensure Python is installed. Install Django using pip: *pip install django==4.0.*

  Additional packages are required (inside the backend folder):

  ➔ *pip install djongo*
  ➔ *pip install djangorestframework*
  ➔ *pip install django-cors-headers*

4.3 PyTerrier + Pandas + NLTK + skLearn

- **PyTerrier**: Latest stable version as per the PyPI repository. **Installation**: *pip install python-terrier.*
- **Pandas**: A stable version like 1.4.0. **Installation***: pip install pandas==1.4.0.*
- **NLTK** (Natural Language Toolkit): Version 3.6.5 or above. **Installation***: pip install nltk==3.6.5*.
- **Scikit-learn** (sklearn): A stable version like 1.0.2. **Installation***: pip install scikit-learn==1.0.2.*

4.4 MongoDB

- **Version**: As of my last update, MongoDB 5.0 is a recent stable version. **Installation:** *brew install mongodb-community@7.0*
- **Refer to settings.py inside Django as it plays a central role in connecting the application together.**

4.5 After All Installation

- Initializing Backend:
  - Traverse to “backend” folder in the project
  - On terminal:
    - python manage.py makemigrations
    - python manage.py migrate
    - On another terminal tab
      - mkdir ../db
      - Mongod - -dbpath ../db
  - Initialize MongoDB Compass and enter the running DB:
    - In ClassicCars Database, create a *Collection* called “backend_vehicle”
    - In the created collection, import the JSON file “jsonDB.json” from “/project/class_cars/JSONS/jsonDB.json”
  - Finally on backend directory terminal:
    - python manage.py runserver
- Initializing Frontend:
  - Traverse to “/project/classic_cars/Classic_Cars_Project” on VS Code and open “Classic_Cars_Project.csproj” and run the file by clicking “play button” on the right top corner of the screen.

5. Conclusion

In summary, we have effectively developed an internet search engine tailored for car shopping. This system is proficient in web crawling and indexing, enabling it to conduct searches and deliver accurate results to users. The user interface is designed for optimal interaction, allowing users to view and compare the vehicle cards for detailed insights and providing direct links to the auction/retailers' websites. The API endpoints are configurable, robust and efficient, ensuring code reproducibility and further improvements.

Please refer to the video of the working environment that can be found here: https://drive.google.com/file/d/1QmkeCA666RCNpzdbhjsPDvjFcCffMUR1/view?usp=sharing

6. Appendix

Timestamp	I think that I would like to use this system frequently	I found the system unnecessarily complex	I thought the system was easy to use	I think that I would need the support of a technical person to be able to use this system	I found the various functions in this system were well integrated	I thought there was too much inconsistency in this system	I would imagine that most people would learn to use this system very quickly	I found the system very cumbersome to use	I felt very confident using the system	I needed to learn a lot of things before I could get going with this system	Did I find what I was looking for?	as it clear how to successfully complete all tasks?	Any other comments?
11/12/2023 21:00:12	4	4	5	5	5	4	5	5	5	4	-	4	4
11/12/2023 16:10:40	4	5	5	5	5	5	5	5	4	5	-	4	5
10/12/2023 18:39:58	5	5	5	3	4	5	4	5	5	5	-	5	4