Information Retrieval Project

Likollari Kelvin, Prendi Gerald

December 16, 2022

An advanced Web Crawler used to crawl data from 3 different motorcycle web pages.

1. Introduction

For the Information Retrieval project, an innovative search engine has been created. This search engine is designed to display motorbike information. Three distinct, but motorcycle-related websites have been crawled in order to accommodate users' expectations for motorbike search when utilizing the search engine.

2. How to start the crawler

A Makefile, containing all the instructions needed to start the application has been written in order to prevent any undesired complications. To activate the app, simply run **make run build**. This will run all the commands present inside the Makefile. Alternatively, in case anything goes wrong, the commands needed for running the backend of our application are stated below:

```
python3 -m venv .venv

source .venv/bin/activate

pip install -r requirements.txt

solr create -c motorcycles__

cd src

scrapy crawl Motorcycle -O motorcycles.json

cd solr/api

python3 schema.py

post -c motorcycles__ motorcycles.json

cd ..
```

```
FLASK_APP=src flask run
```

Unlike the backend, our front end is significantly easier to be run. The needed-to-be-run command is:

```
npm run dev
```

Given the intricacy of all these instructions (particularly for the backend), a bash script file containing the commands required to start the application has been prepared to make your life simpler.

3. Frameworks and Project Structure

The project has been separated into 2 parts:

(1) The Backend

(2) The Frontend

Effectively, as the name states, the backend of our application contains the structure of our project and the logic describing the way our application parts are connected. The official language used for the backend is Python. For the User Interface, VueJS was used. VueJS is an open-source model–view–ViewModel front-end JavaScript framework, used mainly for building user interfaces and single-page applications. Effectively, VueJS is a JavaScript framework, aimed at personalizing user interfaces.

4. Application Backend

The backend of our application may be the most important, but it is also the most difficult to understand. In this section, the full project functionality, which is contained in the backend, will be thoroughly discussed.

(a) As previously said, our aim was to crawl an assigned (as well as two of our choosing) motorbike website. The crawling files for those websites are located in the SPIDERS directory. Once done with the scraping, the content of the webpages browsed, is put in the JSON files. The scraped content of each motorcycle is stored in the JSON file, together with its details (name, brand, manufacturer, year, etc).

(b) **Schema files**: Schema files have been used as they help in describing the structure of our incoming data file.

Our application consists of 3 schema files, one storing entries in JSON format, one in XML format, and the other written in python. Our schema files were developed in such a way that their format may be utilized to specify how the crawled data is translated by Solr. Specifically, the JSON one takes into consideration the categories stored in the motorcycles.json file (name, manufacturer, year, etc). Those entries have specified the language in which the material was crawled, as well as other details such as whether it is saved, indexed, or even required.

When the Solr installation was finished, the first file to be configured was schema.xml. Effectively, our schema.xml file defines firstly the name for the field, which is required,

alongside the type, whether the field should be indexed (i.e. searchable or sortable), whether the field should be retrievable, etc. Finally, the python schema file should be self-explanatory, since it has been utilized and set up to assist in the configuration of the JSON schema one. The Schema API can be used to change our schema content, utilizing the ManagedIndexSchemaFactory class. The Schema API allows us to have read and write access to the Solr schema for all collections. Fields, dynamic fields, field types, and copyField rules may be added, removed, or replaced.

(c) **Config.json**: The Config Solr API enables altering many parts of your XML files via REST-like API calls. Using this API, all modified configurations are recorded in the config.json file.
The code of the config.json file is depicted below:

```
{
"config": {
    "add-searchComponent": {
        "name": "suggest",
        "class": "solr.SuggestComponent",
        "suggester": {
            "name": "mySuggester",
            "lookupImpl": "BlendedInfixLookupFactory",
            "dictionaryImpl": "DocumentDictionaryFactory",
            "field": "__suggest__",
            "indexPath": "suggester_infix_dir",
            "suggestAnalyzerFieldType": "text_en",
            "buildOnStartup": "true",
            "buildOnCommit": "true",
            "highlight": "off"
        }
    },
    "add-requestHandler": {
        "name": "/suggest",
        "class": "solr.SearchHandler",
        "defaults": {
            "suggest": "true",
            "suggest.count": "10"
        },
        "components": [
            "suggest"
        ]
    }
}
}
```

As seen from the code above, this JSON file has been used to configure the suggestions shown to the user typing the query. It uses the SuggestComponent class from the Solr API, used in order to provide automatic suggestions for query terms. It is a powerful feature aimed at auto-recommendations in an IR system. Solr's SuggestComponent class is also used for spell-checking.

(d) **Pipeline**: We have built a pipeline for this project. This pipeline is used mostly to receive the user input and transform it into a Solr query request, which shall lead to also being able to alter the Solr result too.

(e) **Bike**: In the Bike.py, Bike2.py, and Bike3.py you shall find the defined fields for our items (in this case, our motorbikes).

(f) **App.py**: This file is the fundamental "brain" of how our application backend functions. It is essentially used to start the Solr server. It's also how the Flask framework is started. When a query is placed in Flask, App.py is used to handle incoming requests and attempts to query the Solr backend. Furthermore, it is utilized to automatically update the suggestions every 60 minutes, to establish the Solr API, and to start the crawling operation in general. Finally, it contains certain configured routes, which are mostly used for querying and recommendations.

(g) **Flask**: In the above subsection, Flask framework was mentioned. Flask may be used in conjunction with Solr to allow the user to execute a query search and receive the (typically) desired results.

(h) **Deployment**: Our application has also been deployed. We wanted to make it easier and smoother for our testers to utilize our application, so we used Docker. To save our testers time from entering many commands in order to execute our application, we created a Dockerfile that, when run, will automatically install all the assets required for the program's startup. A docker-compose.yml file has also been created, which when executed, may start both the backend and front end of our application on distinct ports. Last but not least, a Makefile has been created to help with testing our program. The Makefile provides the commands needed to start the program and install the relevant dependencies, when applicable.

(i) **Automatic Scraping**: Automatic scraping is an additional functionality for this project that has been set up and deployed. As the headline indicates, a scheduler has been set up which re-crawls all three websites in 60-minute intervals. This implies that if there are any new additions/deletions or modifications to the objects (motorcycles) stored on any of those three websites, they will be reflected in our collection (JSON file) as well.

(j) **Solr cloud deployment**: In addition to the aforementioned functionalities, we have also cloud-deployed Solr. It has three replicas, which are primarily intended to increase searching capacity and improve fail over by giving additional copies of the data. For reference, a replica is a tangible representation of a shard.

5. Application Frontend

The front end should be simpler to comprehend. As previously stated, VueJS was utilized to create the User Interface. Our front end has been broken into several components, each focused at a different part of the program. The search bar, the page's footer, the auto-completion component shown while the user is entering a query, the list of results after the user has finished formulating his question and wishes to receive results, and so on are common components. All of those components have been suitably styled and are included in the main.vue file, App.vue, which contains (includes) all of the User Interface components. App.vue is the main user interface file that contains all of the components and views.
Our application is very Google-like, meaning it has adopted design ideas from the original Google Search engine. It contains an image in the middle, an input search bar where the

desired query shall be typed, and a search button. We aim for a user-friendly and at the same time simple user interface to make the querying experience more friendly to the user.

In addition, a directory containing the application views has been created. These views, unlike the components directory, are dynamic and are changed based on the current application state (i.e. if the user is formulating a query). Vite, last but not least, was utilized. Vite comes with several templates and supports various prominent front-end frameworks, including VueJS. Vite is a build tool that includes a development server that bundles production code. To begin the User Interface, use the aforementioned command:

```
npm run dev
```

, we are essentially running Vite, on port 5173. Vite provides many functionalities such as running an application with a developer server and has additional features such as a live server that reloads the page when a code modification has been performed. Moreover provides detailed descriptions in case of compiler errors.

Additionally, in the front end, you will find a **utils** directory containing various Javascript files. Those files contain some typical getters and setters for our application, written to make our lives easier.

Moreover, a router directory has been set up, containing a file describing the routes of our application and the components inserted, based on the URL path.

Furthermore, in the store directory, there exists a javascript file containing, in the beginning, the accessors and mutators necessary for our application. The interesting part though is at the end, where actions regarding our application are contained. They are divided into two categories, querying, and suggestions. After fetching the respective path, the results are stored in a JSON file.

Last but not least, a Dockerfile has been written in order to make the process of running the application on different computers easier.

6. User Evaluation

After finalizing our mini search engine implementation and testing the system on our own machines, we needed to acquire other testers with other points of view and perspectives to test, assess, and comment on our application. Even though the majority of people ignore it, user assessment is one of the most crucial components of product publishing. Typically, people/companies deprioritize user assessment in order to focus on what they consider to be more critical characteristics of their product. Skipping user evaluation might have serious repercussions and for this reason, we decided to thoroughly hire testers our application. Four people thoroughly tested our system. The users ranged in age from 18 to 34 years.

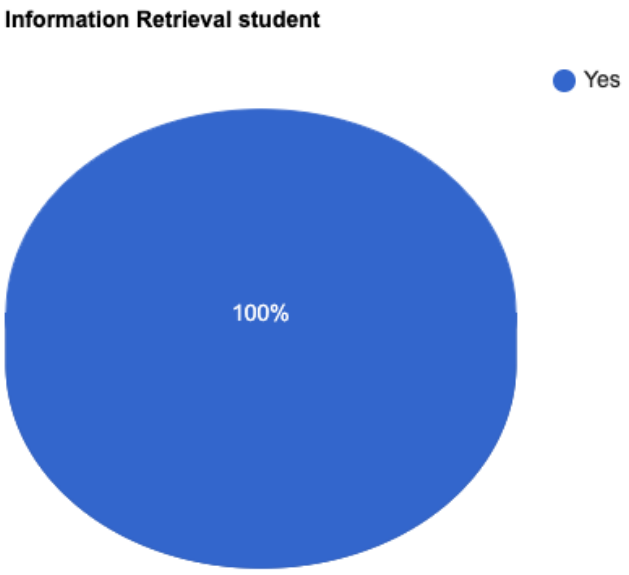

Figure 1: User Evaluation Target Audience

Identifying our target audience is the first stage in User Evaluation. The preceding pie chart shows that our program was completely and carefully examined by Information Retrieval students.

The following is some general comments about our application:

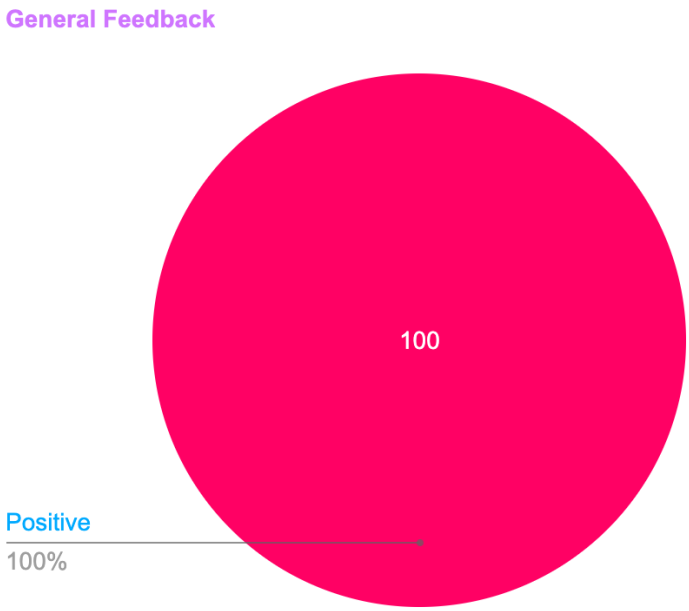

Figure 2: General Search Engine Feedback

As can be seen above, the great majority of our users were really satisfied with how our application worked.

Regarding the complexity of our application, despite the fact that the User Interface we used was basic and straightforward, with a simple search bar in the middle, an input box, and a search button (Google-style), one of our users considered our IR system a bit confusing. This piqued our interest, and it appeared that users who found our system a bit sophisticated would utilize search engines with a lot of information on their main page, such as Yahoo.

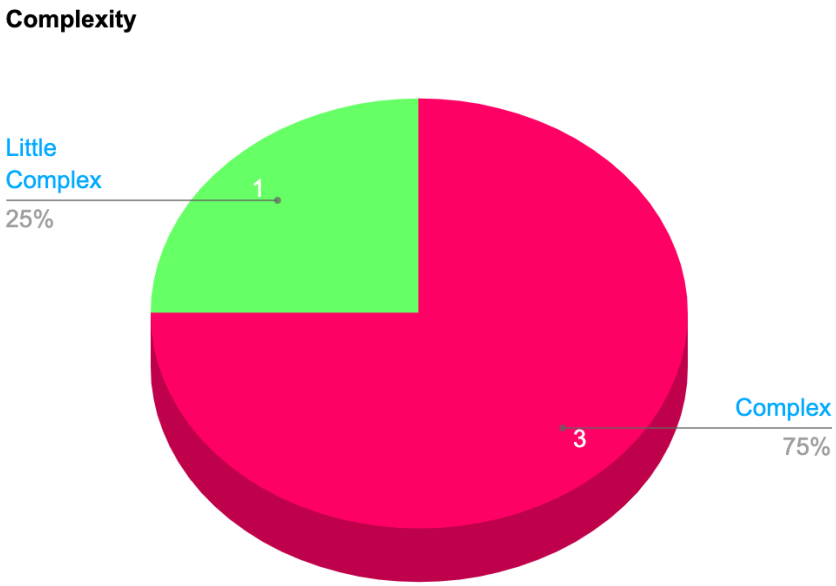

Figure 3: Application Complexity

Fortunately, none of our testers would require external assistance to use/test our program, which was a relief because when we created this application, our primary aim was simplicity, i.e. not requiring other resources to utilize it.

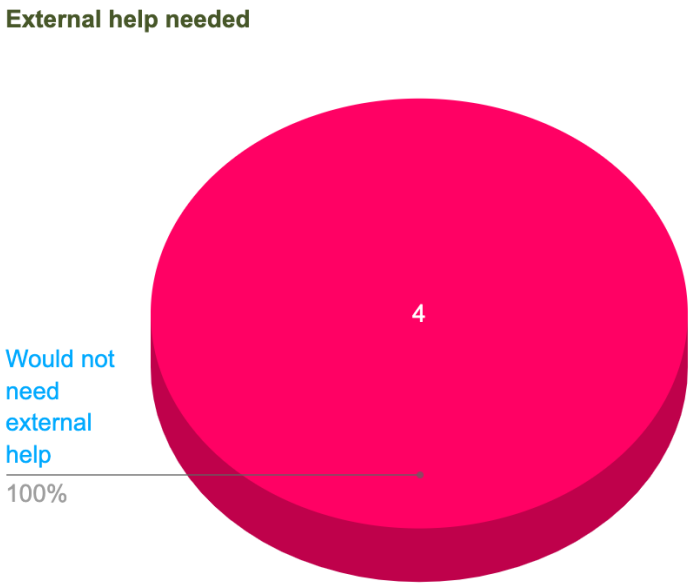

Figure 4: External Help

When building this system, we prioritized consistency. We wanted the user to be able to see all of the results that were relevant to their query. Unfortunately, we did not entirely meet our users' expectations in this regard, as seen by their ratings: one out of two believed the system was a bit erratic/inconsistent, while the other discovered no errors at all.

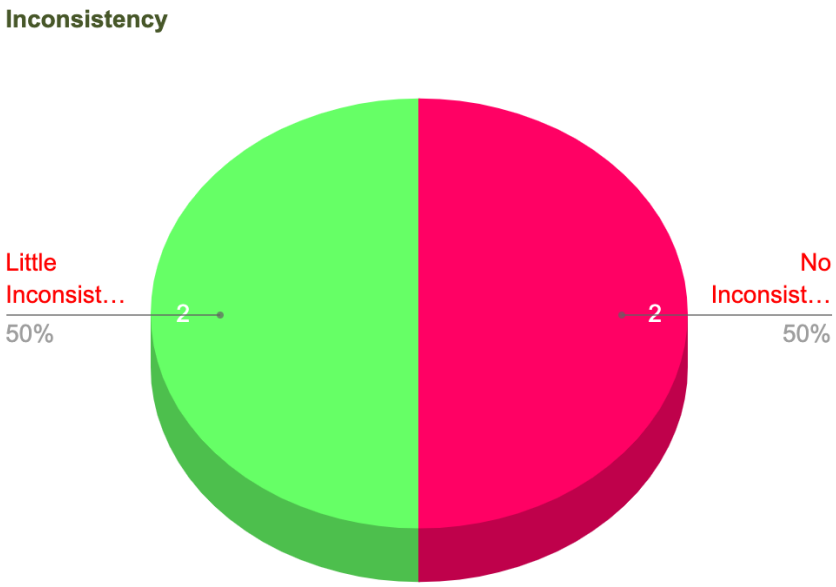

Figure 5: System inconsistency

Our customers were convinced that they would achieve their goals by utilizing our system.

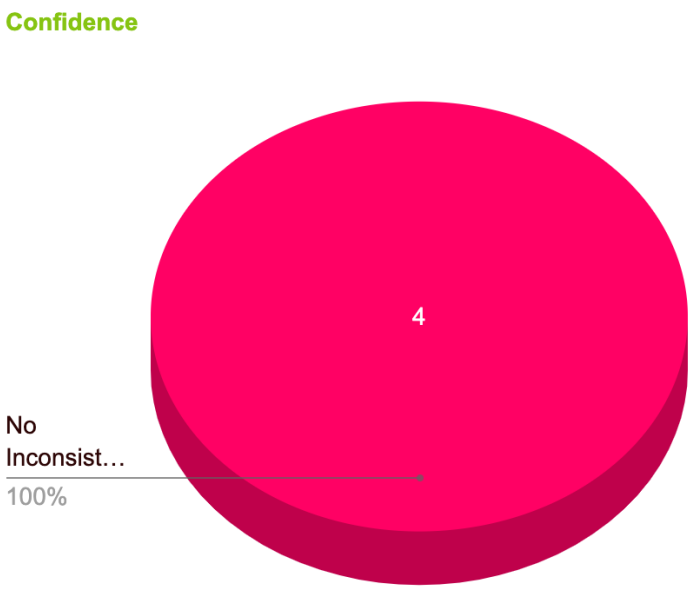

Figure 6: User Confidence

In terms of the efficacy of the results, the great majority of our users discovered the motorcycle they were looking for.

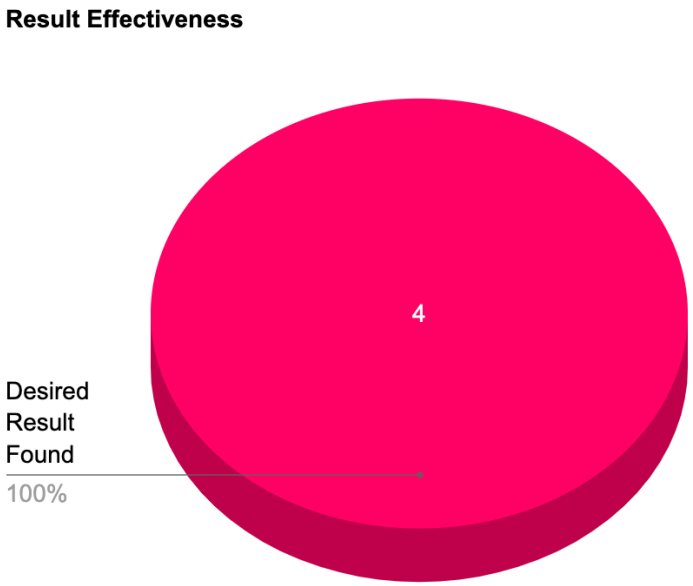

Figure 7: Result Effectiveness

We requested users to assess our search engine at the end of the user evaluation session, taking into consideration the aforementioned pie charts and their overall experience while testing our crawler. The outcomes were mostly positive, as seen below:

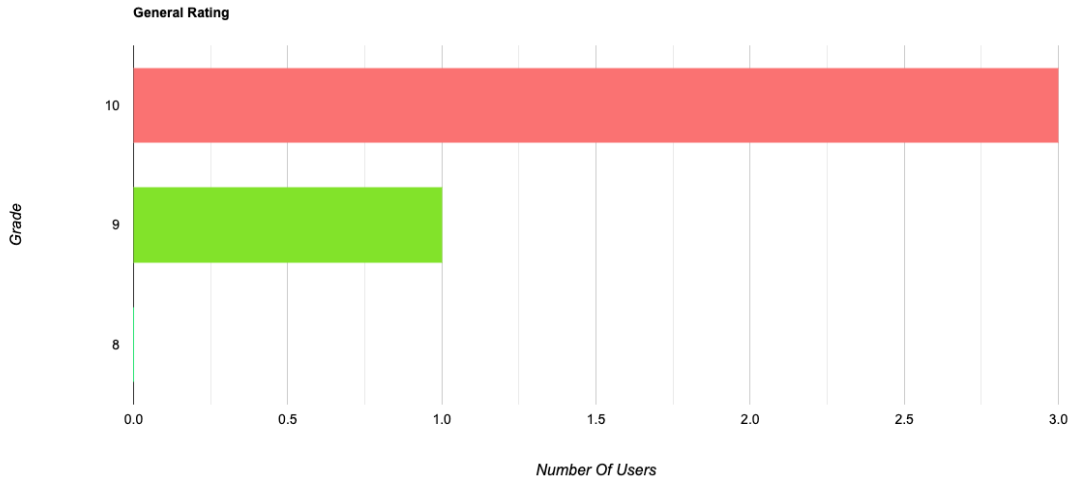

Figure 8: System Rating

As an extra step, we asked users what might have been done better while building this

search engine. We listened to their suggestions and worked hard to improve our crawler while there was still time. Typical suggestions include, but are not limited to:

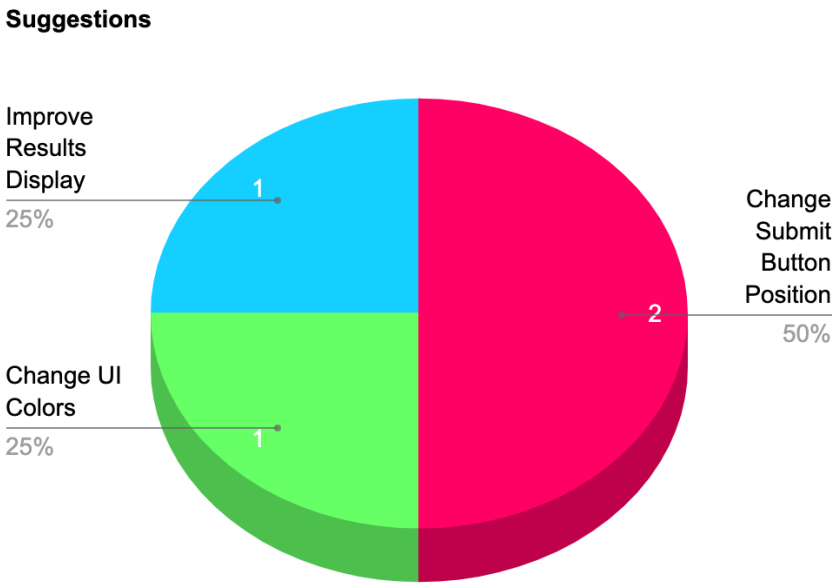

Figure 9: User's Suggestions

Last but not least, we asked users their thoughts about our advanced features. Our advanced feature was **automatic recommendation**. 75% of our testers liked the automatic recommendation feature, while the rest 25% were not really impressed by it.

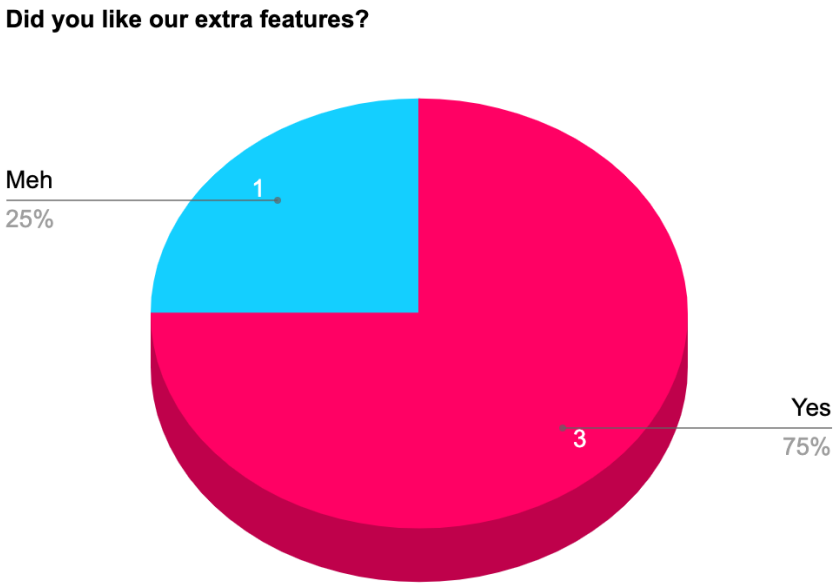

Figure 10: Recommendation Review

Regarding the data, it can be found in the **data** directory. The actual data crawler is stored in two JSON files, called **moto.json** and **motorcycles.json**