COLLEGIUM DA VINCI

Wydział Nauk Stosowanych

Kierunek: INFORMATYKA
studia pierwszego stopnia

Bartosz Pietrzak

Short And Sweet: Comprehensive Approach for Summing up Application's Reviews using NLP

Do rzeczy: Zwięzłe podejście do podsumowywania recenzji aplikacji przy użyciu NLP

Praca inżynierska

Praca wykonana pod kierunkiem
dr. Tomasza Tyksińskiego

Poznań, 2024

1. Introduction

As the digitalisation of today's society progresses, more and more data is flooding the shared space of the internet. As a result, data has become one of the most important resources a business or an individual can obtain. Based on information retrieved from it, solutions to problems can be created, and business decisions can be made. Because of the value of data, companies have developed systems and IT positions like data analysts to extract valuable insights from the data. Amazon gathers data on user preferences by recording what they have viewed or purchased; Google’s YouTube platform records users' viewing habits by constantly monitoring videos played to improve their recommendation system.

Unfortunately, much of the user-created data is unstructured, limiting knowledge that can be extracted without prior preprocessing. In short, unstructured data does not convey useful information at first glance. A good example of unstructured data would be a review. It is only possible to know whether a review was positive or negative or the reason for writing it once it is read. A star or thumbs-up rating is one solution for managing a pure review's unstructured nature. A simple sentiment analysis can be performed thanks to the implementation of a rating. However, the reason for said rating is still concealed behind the veil of unstructured data. Additional requirements, such as mandatory review titling or tagging, would be needed to understand the reasons behind a rating. However, this could significantly reduce the number of users willing to give a rating.

However, AI (Artificial Intelligence) has become more widespread for automated data structuring. For example, the market analysis chapter will broadly examine OpenAI’s Data Analysis tool or service provided by MonkeyLearn Inc. Automated data exploratory analysis and different AI tools.

Despite the renaissance of AI-based developer tools like GitHub’s Copilot or OpenAI’s ChatGPT, tools that convey the sentiment of a given application's user base to assist business intelligence are still rare.

This thesis will present a tool that will help publishers of given applications on the Google Play Store understand their product's main flaws and strengths in the eyes of their user base. Tags will structure reviews. Tag extraction and review tagging will be

performed by a machine-learning topic modelling technique called BERTopic. Additionally, the general sentiment of the reviews will be analysed using the DistilBERT transformer model from HuggingFace, a popular machine-learning framework. All this information will be presented in a bar and pie plot format to simplify the user interface.

2. The current state of knowledge

This chapter will present the problem domain, the rationale for partaking in such a topic, examples of existing solutions and a general overview of the technologies used in the project.

2.1 Vocabulary

Before any examples of the applications are shown, an introduction to technical terminology is in order:

- AI is short for artificial intelligence. Artificial intelligence is a part of the computer science field. It is meant to create systems which resemble human intelligence in different settings and tasks.
- Machine Learning, or ML, is a subset of Artificial intelligence. The field creates mathematical models that can improve output quality over time as they are trained on their training data.
- GitHub is an open-source code hosting platform.
- Open source refers to open-source software with publicly available source code.

2.2 Market analysis

Machine learning and, in broader terms, AI are essential fields of computer science, contributing many valuable tools as they progress towards finer solutions. These advanced tools eventually become available to the general public, including developers; however, despite their indisputable usefulness, many of these tools still need to be considered by their user base. There might be many reasons for the approval and disapproval of a given AI-based solution. Nevertheless, three general reasons stand out as the most decisive:

- ease of setup,
- readability of the results,
- quality of the results.

User interface is the most conspicuous reason for user interactivity with a given tool. A more readable and pleasant interface leads to more clarity.

Related but distinguishable is the ease of setup. Tools that require extensive prior installation might become unusable solely based on the difficulty of their activation. Thus, it is essential for a given application to minimise the extent to which manual configuration is required. The approachable configuration has a second merit. If the results yielded by the application are satisfactory to the user, they might be more satisfied if the setup process is not convoluted. There is no need to explain that the result granted by the given tool is its trademark. Many AI tools have lost users due to the results they generate.

Additionally, as much as results are of the utmost importance, many users – especially technologically inclined – pay much attention to whether or not a given tool is open-sourced.

Below are examples of AI-centred applications and tools that follow previously described user-friendly features and some that do not.

MonkeyLearn

Service created by MonkeyLearn Inc. [18], is a tool that provides text analysis solutions via a pleasant and readable interface.

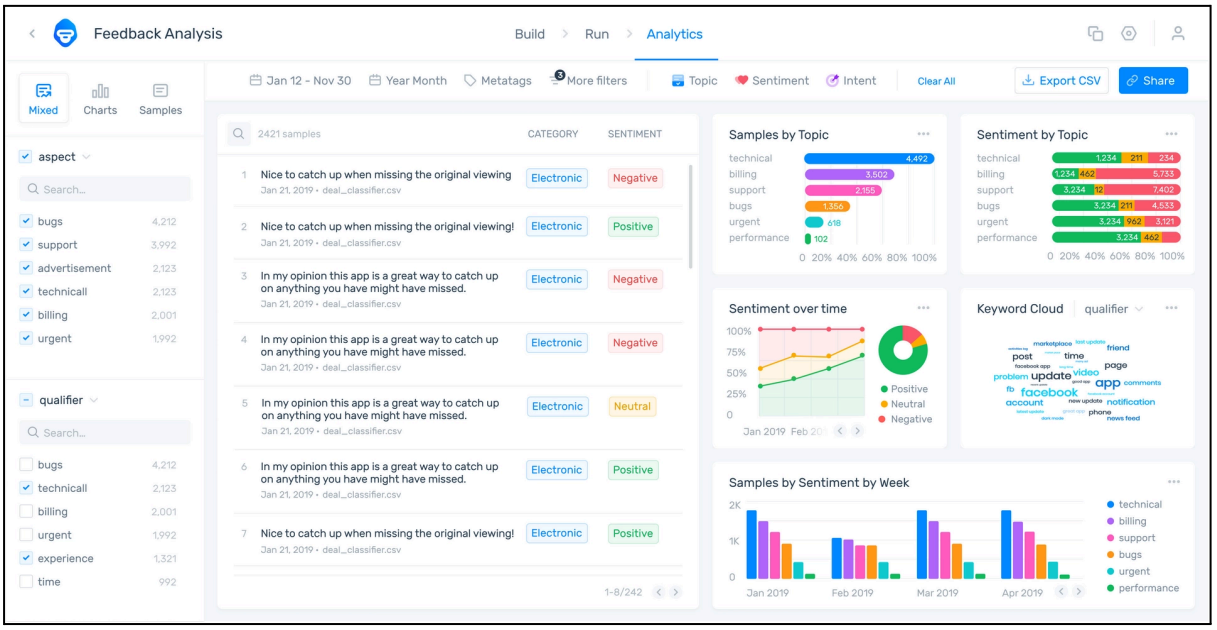

Figure 2.1 Interface of MonkeyLearn's service (source: [18])

The service works on textual data the user passes as a .csv file. The application then analyses the data and extracts insights such as topics, sentiment, and urgency. It

supplies a range of tools that allow users to build their pipeline for insight extraction. It is a robust and intuitive solution for businesses that receive unstructured data and want to derive value from it.

However, there are some caveats. Mainly, the service is paid, making it inaccessible to some users. Secondly, MonkeyLearn Inc. needs to provide the source code for their project. An open-source solution would have allowed developers to enhance their service and add additional features. The most considerable merit of this application is its intuitive and informative interface. This thesis project will try to mirror the intuitive nature of plots implemented in MonkeyLearn's application; simultaneously, its source code will be available to the public, allowing for free use of the project.

DALL-E web service

DALL-E [7] is a web service released by OpenAI, a research organisation in San Francisco. In 2021, generative AI was not a famous phrase in technical or non-technical communities. DALL-E was one of the first generative machine learning models that received significant attention in technical and non-technical media.

The purpose of the service is to generate images based on text prompts written by the user. While text prompts might follow a standard way of explaining the desired outcome to the machine, it is generally better to structure the prompt in an already established conventional way. For example, repeating an image's most desired object or style several times in the description. Depending on the quality of the prompt, the user may receive a satisfactory image.

The ease of use is self-evident. The user controls only the text prompt and optional image input. The simple interface allows newcomers to focus on and experiment with different structures of prompts instead of being overwhelmed with many settings for image generation, as is the case with an open-source project for a different generative model called "Stable Diffusion web UI" [1].

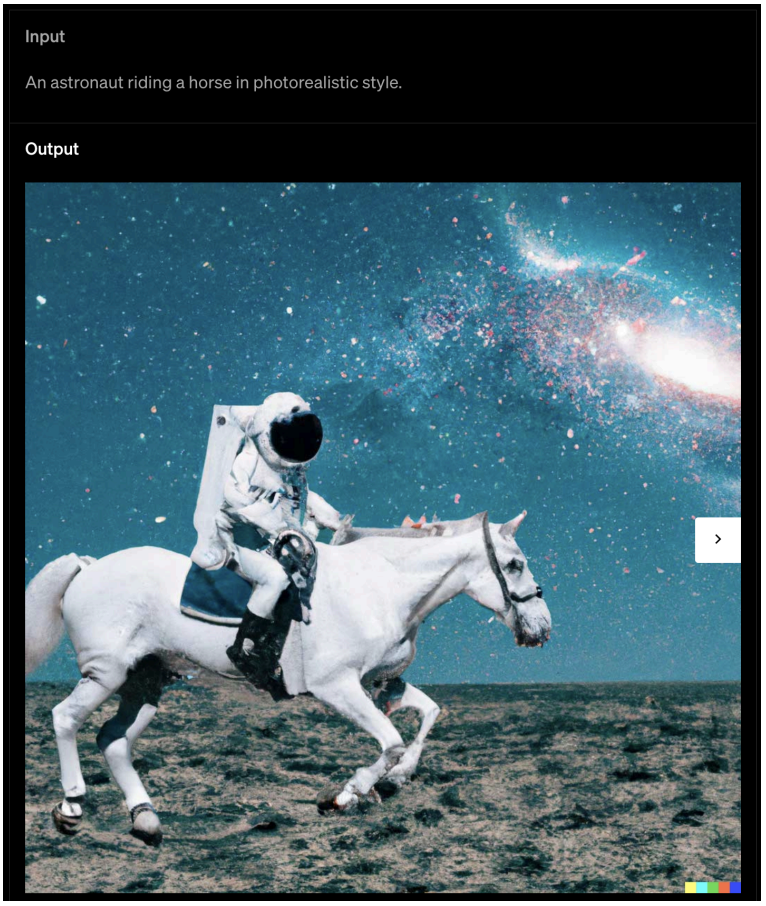

Figure 2.2 Screenshot of DALL-E's interface (source: [7])

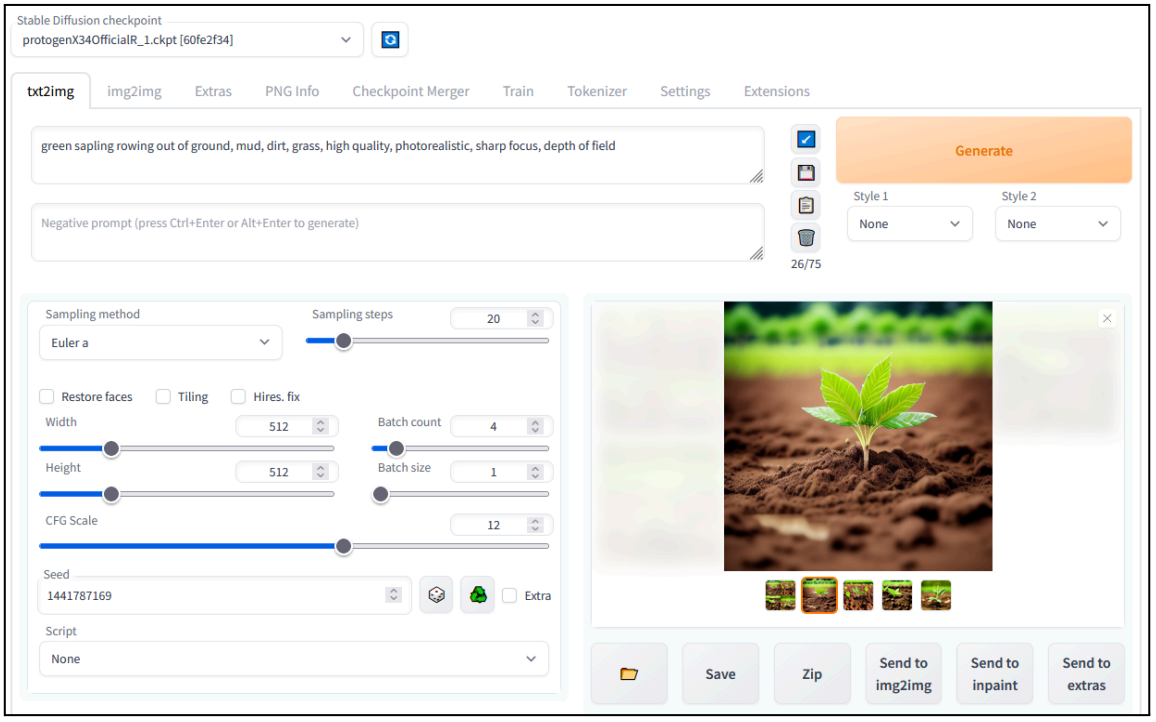

Figure 2.3 Picture of Stable Diffusion web UI project's interface (source: [1])

Nonetheless, as previously mentioned, AI-based applications must yield generally satisfying results. When it comes to DALL-E, the competition has outperformed this service. Applications like MidJourney and Stable Diffusion, even though working on similar models, produce objectively better — more aligned with user desire — pictures. As a result, image-generation-centred communities prefer image-generation applications provided by MidJourney Inc. or Stability AI instead of the ones run by OpenAI.
This thesis project aims to emulate DALL-E's straightforward and intuitive interface. Recognising the appeal of DALL-E's user-friendly design, the project focuses on simplicity, making advanced AI accessible.

Stable Diffusion Web UI

The last example of an application that utilises AI functionalities is the GitHub project called "Stable Diffusion Web UI". Its primary purpose is to generate text-to-image and image-to-image data. As with the DALL-E OpenAI service, this project contains a field allowing the user to input the text, resulting in a rendered image. There is also a possibility of using an existing image as input, which will result in a new image that contains some structure of the source image but with varying features based on training data on which the background AI model was trained. This project has more use cases, which this thesis will not describe further.
As mentioned and shown in picture 2.3, the user interface of Stable Diffusion web UI is more complex than in previous examples; however, the primary user base for this project also differs from the earlier examples. The project is free to download, and it is open-sourced on GitHub. Users of the GitHub platform are more likely to be technologically inclined than the general public. Additionally, the project documentation is extensive, which is a significant help when setting up a GitHub project.
The new project will replicate the thorough "Stable Diffusion web UI" documentation but will avoid its complex interface. The aim is to provide clear and helpful guidance in a simpler, more user-friendly environment.

As illustrated by the above examples, developing AI-based applications involves multifaceted considerations. Primarily, the outcomes must meet the end-user's

expectations and possess a visual appeal. Furthermore, the complexity of the application should match the average user's skill level. Comprehensive and well-structured documentation becomes crucial to mitigate potential user confusion if the application is inherently complex, like the "Stable Diffusion web UI" project. This principle guides the new project, which aims to combine the informative documentation style of "Stable Diffusion web UI" with a more user-friendly interface reminiscent of DALL-E's simplicity and informative interface of MonkeyLearn's service to create a more informative and easier-to-navigate application.

2.3 Selection of tools

Technologies must be introduced as the different applications to thematically similar problems were presented.

Foremost, the whole project was hosted and developed using the Git [10] source control system, as it is one of the most popular tools for this task, and the GitHub hosting platform [11] for easy open-source code distribution. Additionally, the whole project was containerised using Docker [8] to simplify the activation of the service and increase its portability. Dockerization also allowed for simplified deployment of machine-learning endpoints to Google Cloud Service (GCP), which will be further explored later in this chapter. Every part of this application was written in the Visual Studio Code integrated development environment [38].

2.3.1 Backend

The first and primary technology used in the project's backend was the scripting language Python, version 3.11 [25]. It was chosen for numerous reasons: ease of development, readability, and integration with many machine learning libraries that were crucial for the project. The particular version of Python was chosen because of the speed increase and syntax features that were not present in the previous versions. The libraries will be introduced in the order in which they were actively used in the project. FastAPI [33], Uvicorn [36] and Pydantic [23] were used in combination with each other. Their primary purpose was to create an Application Programming Interface (API) to enable HTTP communication between different application functionalities, some of which were written in different coding languages. FastAPI was responsible for the creation of API functionalities; Uvicorn for traffic

handling – it is responsible for running the Asynchronous Server Gateway Interface (ASGI) server; Pydantic was used for type annotation, which helped in writing unit tests, validating incoming requests and debugging errors. The Httpx [15] library's primary use was sending and receiving HTTP requests to and from various endpoints. The "Google Play Scraper" [17] library was added to implement app information and review retrieval based on the URL that was passed from the user. A Redis [26] library was implemented to enable communication between backend functions and the Redis database, which was used as a cache in this project. The implementation of the Redis database will be described in the later part of this chapter.

BERTopic [13] and the Transformers [16] library were chosen to implement machine-learning algorithms. BERTopic was shown [14] to extract keywords from groups of documents to convey their general meaning, whereas DistillBERT has been shown to hold similar accuracy to its larger counterpart – BERT with more minor architecture [31]. Transformers library, however, was chosen because of the large assortment of pretrained transformers-based models, which are often used for sentiment analysis tasks. As the Transformers library is closely integrated with the PyTorch [27] library created by the Meta AI team, it was chosen for post-processing results from the sentiment analysis model.

Numpy [20] was primarily implemented to simulate machine learning models' outputs to test other program functionalities without inferring machine learning models every time. Lastly, the Pytest [24] library was introduced for testing backend functionalities.

As mentioned, the project utilises a Redis non-relational database [29] as a cache. It is done so because of the asynchronous nature of functions that call for inference of machine learning models. As this project utilises machine learning functionalities by calling an endpoint, which runs model inference on the passed data, the response time of said endpoints may vary, depending on the network speed, amount of reviews passed for inference, and the model run on the endpoint. Because of these reasons, asynchronous functionalities were implemented. However, because of the asynchronous responses from different endpoints, there is no way of knowing whether the required data has been returned at run time. Because of this, a cache management functionality had to be implemented. Redis is known for its fast response time and versatility in saving various data structures [2], which works great with simple Python objects like a list.

Running deep learning algorithms requires significant computing power for timely results. Therefore, a cloud computing platform has been integrated into the development of this project. The Google Cloud Platform (GCP) [12] was chosen for its extensive documentation and the broad coverage it receives in third-party articles. Machine learning endpoints are hosted on GCP using the “Container Registry” service for storage and are deployed via the “Cloud RUN” service. A detailed guide for deploying the inference endpoints of both models has been included, greatly simplifying the process. The setup has been streamlined to the essentials, reducing complexity and enhancing efficiency.

2.3.2 Frontend

The project's front end adopted a minimalistic approach, significantly streamlining the development process. The user interface (UI) concept was designed using Figma [9], a tool well-suited for prototyping UI components due to its intuitive interface; this made the UI design phase the most straightforward aspect of frontend development. React [28], an essential library, was pivotal for the UI's reactivity, playing a central role in interface construction, rendering and navigation management. TypeScript [35], a JavaScript variant, was employed for scripting in React, offering optional typing annotations akin to Python for improved code clarity and simplified testing. Given the application's single-page architecture, effective state management was essential. Redux [30] was chosen for its efficacy in managing the global state across various components and networking. Vite [39] was used as the builder of the frontend application.

Chakra UI [3] was utilised instead of the more commonly used Bootstrap CSS library for UI styling and design. This decision was based on its superior integration with React, enhancing the development workflow. The Plotly [22] library was incorporated to visualise the results of machine-learning model inferences through plots. Its structure mirroring Python's equivalent facilitated a more intuitive development process.

3. Purpose and scope of the project

The project's central purpose is to create an application that visualises the sentiment and recurring themes in the Google Play Store's distributed body of application reviews, enabling developers to make informed decisions based on collective experiences and opinions in app reviews.

Web Scraping

The application enables users to input a Google Play Store URL. The backend processes this input by verifying the URL, extracting the app ID, and scraping user reviews based on user-defined criteria, such as star ratings and review count. Efficient error handling is integrated to manage scenarios like incorrect URLs or insufficient review numbers that a given application received.

Natural Language Processing

Following the extraction, reviews are analysed using machine learning techniques. This phase involves cleaning reviews from unwanted data like emojis or unnecessary punctuation marks and breaking down the review text to extract sentiments and themes.

Data Visualisation

The application's frontend needs to present the analysed data interactively and informatively through graphical representations – pie and bar charts. These visual elements must be interactive, allowing users to explore data by hovering for more details or clicking on part of the plot to explore individual topics.

The authors of this thesis created all of the components present in the discussed project. These components were:

- A user interface developed with React and Chakra UI, handling user inputs and data visualisation.
- The backend, built with Python, is responsible for processing requests, web scraping, and managing data passing between the frontend, Redis cache, and AI endpoints.

- Redis Cache Database was used for storing data in memory, scraping from the web, or holding the results from AI inference.
- AI Endpoints comprise the BERTopic topic modelling pipeline and DistilBert machine learning model, performing natural language processing on the reviews.
- Containerisation and Deployment were introduced to streamline the process of installing dependencies and running the project; additionally, it enabled Google Cloud deployment.
- Testing of the application.

4. Methodology

This chapter will describe the application's software architecture and engineering aspects. First, a use case diagram will be presented to show the bird's eye view of the application, how it works and how users may interact with it. Then, as different components are presented – when applicable – additional diagrams will be provided along with screenshots of the elements discussed.

4.1 Use case diagram and functional requirements

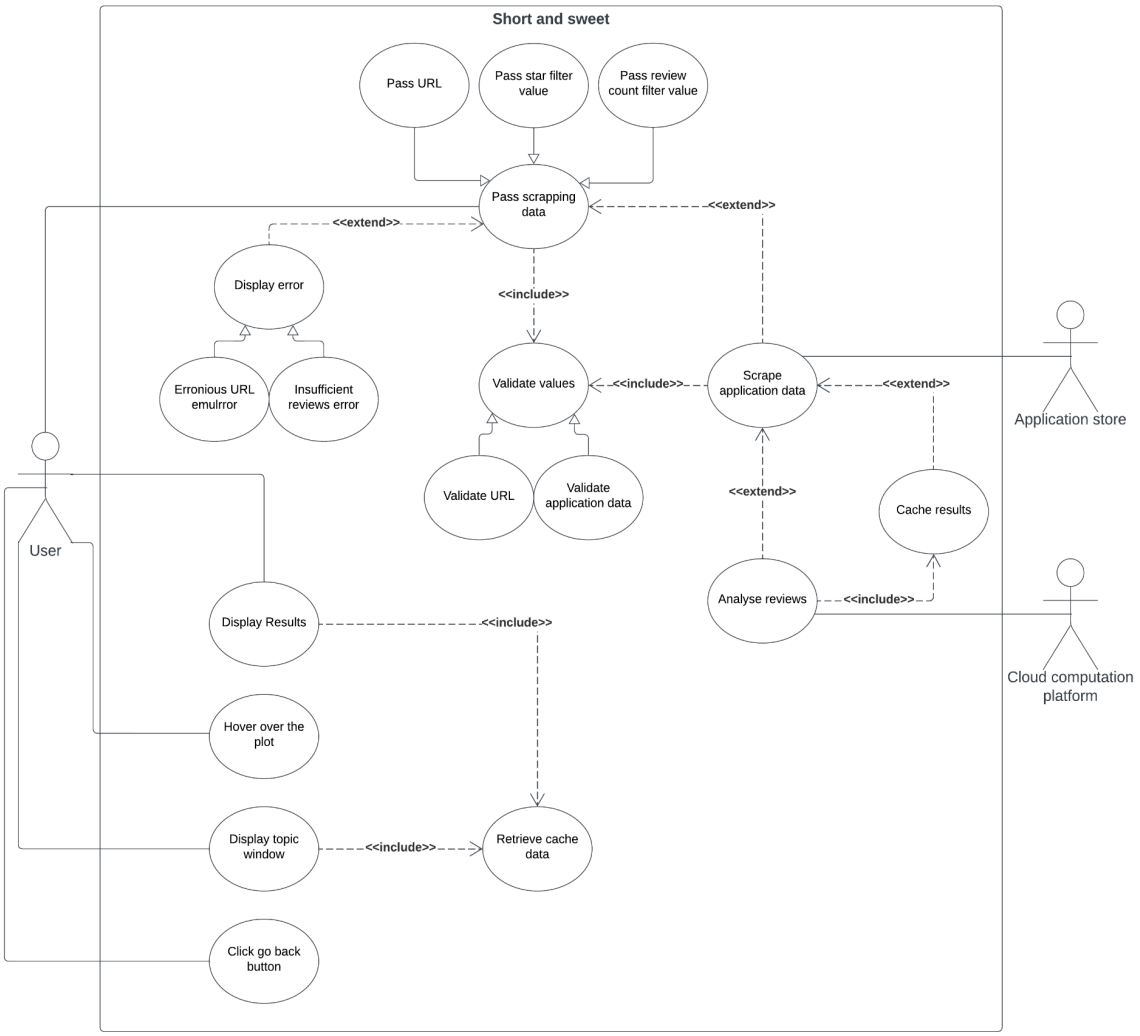

Figure 4.1 - Usecase diagram of the application (source: own elaboration)

Figure 4.1 illustrates multiple user interaction methods with the application. The user can provide necessary data for scraping information and reviews about a specific application. In this thesis, this input is termed "scraping data", comprising the application's URL from the Google Play Store, the desired number of reviews to scrape, and the star ratings for review filtration. This feature is detailed in the accompanying table.

Table 4.1 Pass scraping data functionality

Function name	**Pass Scraping Data**
Description	The user passes the URL, star filter value and review count value passing parameters to the system.
Input data	URL, star filter value, and review count filter value.
Source of input data	Keyboard and mouse.
Result of function	The system receives data for further processing.
Requirements	The application needs to be activated to access the internet.
Purpose	To start the app review analysis process.
Initial condition	Scraping data is sent.
Final condition	The scraping data is successfully inputted and submitted.
Side effects	Incorrect URLs could lead to unnecessary processing wasting system resources.
Reason	To initiate the review scraping process with specified parameters.

Once the system receives reviews from the user, it validates the submitted data. Initially, the system verifies whether the URL directs to the English version of the Google Play Store. If this verification is successful, the system proceeds to scrape reviews from the website.

Table 4.2 Scrape application data functionality

Function name	**Scrape Application Data**
Description	The system extracts relevant data, including the application's name, number of reviews, an icon, and reviews from the specified app's page in the Google Store.
Input data	Application ID extracted from URL passed by the user, star rating, review count.
Source of input data	User input received by the system.
Result of function	Relevant app data is extracted for further analysis.
Requirements	Internet connection
Purpose	To gather essential data from the specified app for review analysis.
Initial condition	Valid application ID and scraping criteria are provided.
Final condition	App data is successfully scraped.
Side effects	The scraping process could be resource-intensive, affecting the performance of other system functionalities.
Reason	To gather essential data for review analysis and app evaluation.

Following the application of review filtering criteria, the system checks if the total number of reviews exceeds 1400. This minimum threshold exists because the BERTopic modelling technique, cannot effectively extract topics from document groups with fewer than 1000 documents.

Table 4.3 Validate values functionality

Function name	**Validate Values**
Description	The system validates the input data for correctness and feasibility.
Input data	URL, star rating, and review count.
Source of input data	User input received by the system.
Result of function	Validation of user's input data.
Requirements	Scraping data must be passed, and internet access must exist.
Purpose	To ensure the integrity and applicability of the user input data.
Initial condition	User input data is received.
Final condition	Input data is confirmed as valid or invalid.
Side effects	Strict validation rules might reject valid data due to format variations, limiting the app's flexibility. Insufficient validation could allow incorrect data to pass, leading to unpredictable errors.
Reason	To ensure data integrity and applicability before processing.

Should any validation step fail to meet a requirement, or if an attempt is made to access the application's result before the process is finalised, the backend will relay an error message to the frontend. This informs the user about a data processing error.

Table 4.4 Display error functionality

Function name	**Display Error**
Description	Displays an error message when a process fails, or data is invalid.
Input data	Error message.
Source of input data	System-generated message based on process failure or validation checks.
Result of function	Error is displayed on the user interface.
Requirements	Passed scraping data must be invalid; the application received insufficient reviews.
Purpose	To inform users about errors in processes or data inputs.
Initial condition	The user passes an invalid parameter, or the application receives insufficient reviews.
Final condition	The error message is displayed to the user.
Side effects	Frequent error messages could negatively impact the user's experience.
Reason	To inform the user about issues requiring their attention or correction.

If the validation process encounters no errors, the analysis phase of the application begins. The retrieved reviews are forwarded to the machine learning endpoints. Here, machine learning solutions BERTopic and DistillBERT, process the data.

Table 4.5 Analyse reviews functionality

Function name	**Analyse Reviews**
Description	The scraped reviews are sent for further machine learning inference analysis. The same function later receives the results of this procedure.
Input data	Scraped app reviews.
Source of input data	Scrapping process output.
Result of function	Reviews are sent to inference endpoints for further analysis. Results are then received.
Requirements	CPU with capabilities equal to or greater than Intel Core i5-8257U 1.40GHz, at least 8 GB of free random access memory, or access to inference endpoint through cloud computation platform; internet connection.
Purpose	To process the scraped reviews through machine learning algorithms.
Initial condition	Reviews have been successfully scraped.
Final condition	The system receives the results of inference.
Side effects	High volumes of data sent for inference could overload the ML system, leading to slow processing times or failures.
Reason	To derive insights from reviews through machine learning techniques.

As the requests sent to the machine learning endpoints are asynchronous, it is crucial to implement a caching mechanism. This approach manages concurrency by preventing redundant computations of the same data across multiple asynchronous tasks. Additionally, it allows for retrieving cached data from various code sections without passing them as arguments.

Table 4.6 Cache results functionality

Function name	**Cache Results**
Description	Store the results of different operations in memory.
Input data	Outputs from various operations.
Source of input data	Calling functions.
Result of function	Other values are stored in contemporary memory.
Requirements	At least 1 GB of free random access memory.
Purpose	To keep the analysis results temporarily.
Initial condition	The cache function is called.
Final condition	Results are stored in the cache.
Side effects	Memory space required for saving data in the cache might be wrongly allocated, resulting in an error when a retrieve call is called.
Reason	To optimise the cache and retrieval of different operations.

Table 4.7 Retrieve cache data functionality

Function Name	**Retrieve Cache Data**
Description	The system fetches the stored results from the cache for display or further processing.
Input data	Request for cached results.
Source of input data	User or system-initiated retrieval process.
Result of function	The cached data is retrieved for use.
Requirements	The called-for data must be present in the cache.
Purpose	To store the analysis results temporarily for quick retrieval.
Initial Condition	Need for data that has been previously analysed and stored.
Final condition	Data is successfully retrieved from the cache.
Side effects	Frequent cache accesses could lead to performance issues.
Reason	To efficiently access previously processed results.

Once any endpoint returns the results of the machine learning inference, they can be displayed to the user. Specifically, the BERTopic results can be presented as a bar plot, while the DistillBERT results can be showcased using a pie chart.

Table 4.8 Display results functionality

Function Name	**Display Results**
Description	Present the processed analysis results to the user in an easily interpretable format. The system will display these results using bar and pie charts, visually representing the data for quick understanding and insights.
Input data	Analysed data ready to be visualised
Source of input data	Data retrieved from the backend processing was recovered from the cache, analysed and prepared for presentation.
Result of function	The user is presented with bar and pie plots visually representing the analysis results.
Requirements	Data required for plotting needs to be present in the cache.
Purpose	to effectively communicate complex data analysis results to the user in a visually intuitive manner
Initial Condition	The analysis is complete, and the data is ready to be visualised.
Final condition	The plots are successfully rendered on the user interface, and users can view and interact with them.
Side effects	Obtained analysed data might not have enough values, thus preventing the plot from being rendered.
Reason	To make the analysis results more accessible and understandable to the user.

Given the application's emphasis on minimalism, it is essential to maintain a clutter-free interface. An interactive feature is implemented to align with this principle: hovering over a plot. This approach minimises on-screen information, revealing more details only when the user hovers over a specific part of the plot, thereby reducing clutter and enhancing user experience.

Table 4.9 Hover over the plot functionality

Function Name	**Hover over the plot**
Description	A user interacts with the plot elements (like pie charts or bar graphs) to view more details or specific data points.
Input data	User action (hovering with the mouse over plot elements).
Source of input data	User action (hovering over plot elements).
Result of function	Display detailed information about the specific part of the plot.
Requirements	Plots need to be rendered.
Purpose	To present the analysis results in a user-friendly format.
Initial Condition	The user views the results displayed in graphical format.
Final condition	Detailed data related to the hovered part of the plot is revealed.
Side effects	A missing value error could be raised if the backend does not provide the data displayed on the hover action.
Reason	To enhance user interaction and provide more in-depth information about the displayed results.

Likewise, if the user wishes to access more detailed information about the classification of reviews, they can simply click on the bar plot. This action will show additional information on generated topics.

Table 4.10 Display topic window functionality

Function Name	**Display Topic Window**
Description	The user clicks or selects parts of the bar plot to view specific data or insights. Then, a window pops up to provide detailed information about one particular topic triggered by user interaction with the plot.
Input data	User action clicking on the parts of the bar plot.
Source of input data	Mouse input on specific parts of the bar plot.
Result of function	Detailed information about the selected topic is displayed in a pop-up window.
Requirements	Internet connection: every previous analysis step needs to be finished.
Purpose	To enable users to engage interactively with the bar plot for deeper insights.
Initial Condition	The user is presented with a bar plot as part of the results and clicks on the bar in the bar plot.
Final condition	Specific interaction with the bar plot is completed, yielding detailed information on the selected bar – topic.
Side effects	If the topic keywords are too long, they might unpredictably change the structure of the layout.
Reason	To provide a deeper and more interactive exploration of the data represented in the bar plot.

Table 4.11 Go back button functionality

Function Name	**Go back button**
Description	A ‘go back’ button allows the user to go back to the starting page of the application. It’s formed to resemble the arrowhead in many browsers and serves the same purpose.
Input data	User action clicking on the ‘go back’ button.
Source of input data	Mouse input on a ‘go back’ button
Result of function	Restarting the applications view.
Requirements	The application validates user input.
Purpose	To allow the user to reuse the application without restarting the landing page.
Initial Condition	Scrapping data is validated, and the scrapping process begins.
Final condition	The page view is reset to the initial state.
Side effects	If the inference endpoints already run on the scrapped reviews and the “go back” button is pressed, the inference results will not be presented, even though they will be computed.
Reason	To allow the user seamless use of the application.

4.2 Class and object diagrams

This sub-chapter describes functionalities that employ object-oriented programming principles.

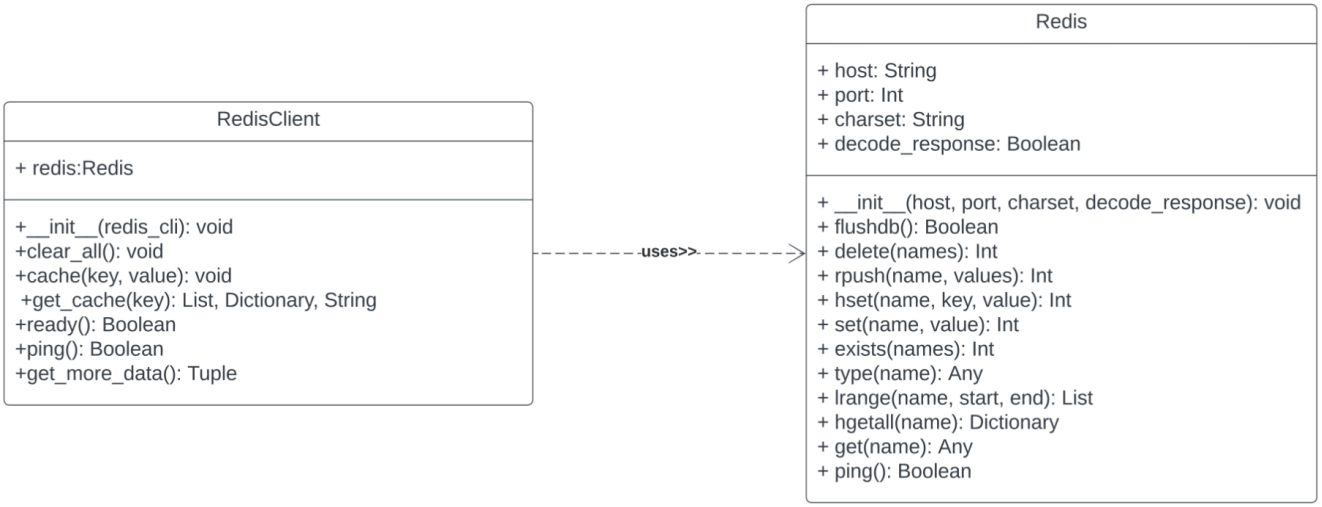

Figure 4.2 Class diagram of RedisClient and Redis classes (source: own elaboration)

The caching features mentioned in tables 4.6 and 4.7 use RedisClient. This class makes it easier to manage the cache, allowing various types of data to be stored in the Redis cache more simply. For example, instead of using different methods for saving lists, dictionaries, or strings, RedisClient offers a unified way to handle these tasks.

```
redis.rpush(key, *value)
redis.hset(key, k, v)
redis.set(key, value)
```

The RedisClient simplifies cache management by providing a single method to handle three different data types, making the process more efficient,

```
redis_client.cache(key, value)
```

but still stores them in a redis cache manner inside of the method:

```
        if value is None or not value:
            pass
        elif isinstance(value, dict):
            for k, v in value.items():
                self.redis.hset(key, k, v)
        elif isinstance(value, str):
            self.redis.set(key, value)
        elif isinstance(value, list):
            self.redis.rpush(key, *value)
```

As mentioned earlier, RedisClient utilises an object from the Redis library. This is the basis for the interaction between these two objects.

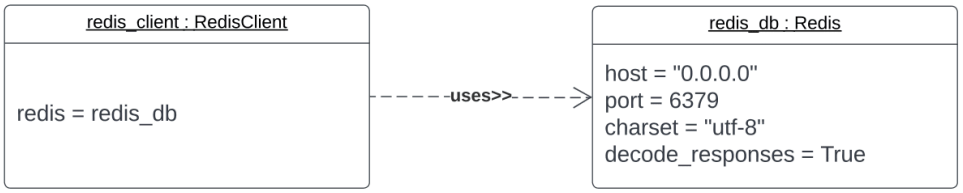

Figure 4.3 Object diagram of redis_client and redis_db objects (source: own elaboration)

Above, it is highlighted that the Redis object utilises four attributes. The first two, 'host' and 'port', establish a connection to the Redis database, while the remaining two manage value encoding and decoding. Upon initiating the 'redis_db' redis object, it is then passed as an argument to the RedisClient class's initialisation method. In this case, the decision not to use inheritance was strategic and aimed at distinguishing errors originating from the custom class versus those from the external library class object.

Additionally, all backend requests utilise data classes from Pydantic's BaseModel. Pydantic, a third-party Python library, is essential for data validation. Developers can access its validation tools by creating classes that inherit from BaseModel. This means every attribute in such a class is automatically validated. While Python does not inherently offer type declaration, Pydantic fills this gap. An illustrative example of a Pydantic class would be 'InferenceRequest'.

```
class InferenceRequest(BaseModel):
   reviews: List[str] = Form(...)
```

The class inherits from an abstract class BaseModel, and accepts a single attribute: 'reviews'. This attribute must be a list of strings to be valid.
Incoming or outgoing HTTP request data is validated to maintain consistency and reliability thanks to the Pydantic library. For example, a machine learning endpoint uses the previously mentioned InferenceRequest class to validate incoming requests. If a request's body differs from what is defined in the class, it will be rejected, ensuring data integrity and alignment with expected formats.

```
@router.post('/inference')
async def inference(request: InferenceRequest) -> ClassificationResponse:
```

More classes inherit from Pydantic's BaseModel, for example backend.

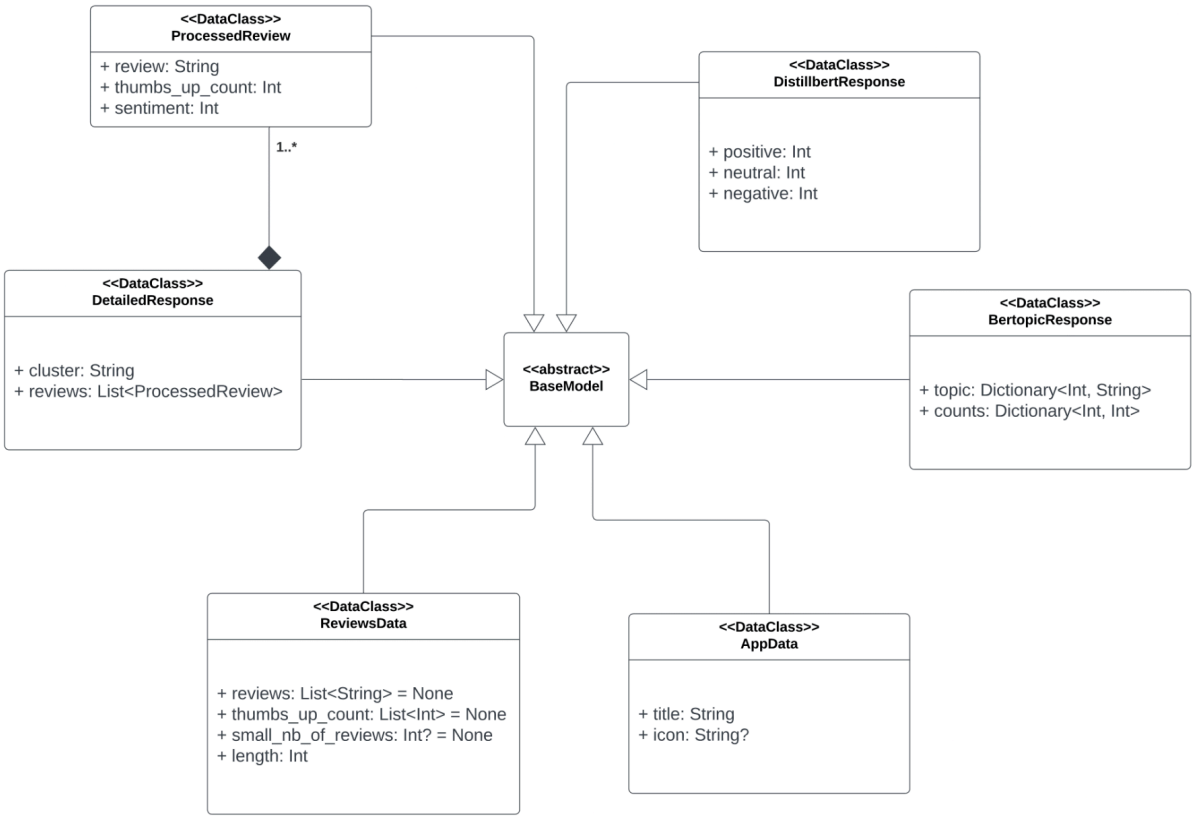

Figure 4.4 Class diagram of the data classes used in the backend (source: own elaboration)

The first Pydantic data classes initiated in the project are ReviewsData and AppData. ReviewsData serves multiple purposes: it validates whether the application has received a sufficient number of reviews and also temporarily stores these reviews before they are cached in Redis. On the other hand, AppData is utilised by the backend endpoint /get_data. This class facilitates the return of the application's name and icon based on the URL provided by the user.

Figure 4.5 Object diagram of the ReviewsData and AppData objects (source: own elaboration)

When AppData is transferred to the frontend and displayed to the user, it triggers another request to the machine learning endpoints.

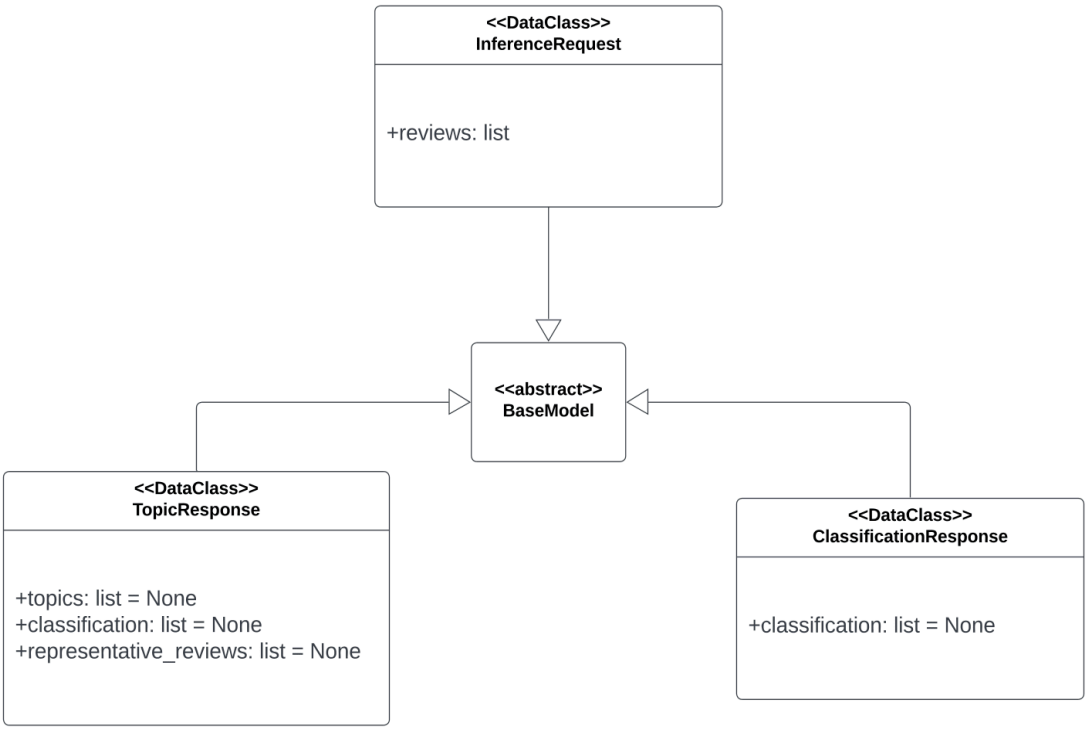

Figure 4.6 Class diagram of classes used in the machine learning endpoints (source: own elaboration)

Figure 4.6 illustrates the previously mentioned InferenceRequest, TopicResponse, and ClassificationResponse classes. The latter two are returned by the BERTopic and DistillBERT endpoints, respectively, and they may take on the following values.

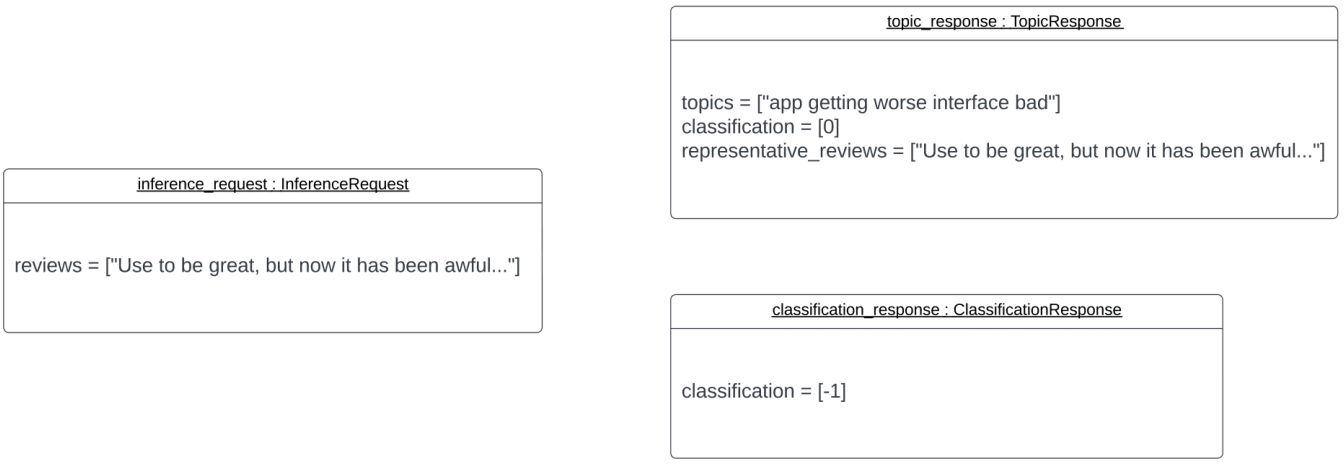

Figure 4.7 Object diagram of the objects utilised in the machine learning endpoints (source: own elaboration)

Upon receiving the inference results from the machine learning endpoints, the backend processes them into a format suitable for visualisation on the frontend. The classes that validate the return values from the backend to the frontend are depicted in Figure 4.8 and may take the following forms.

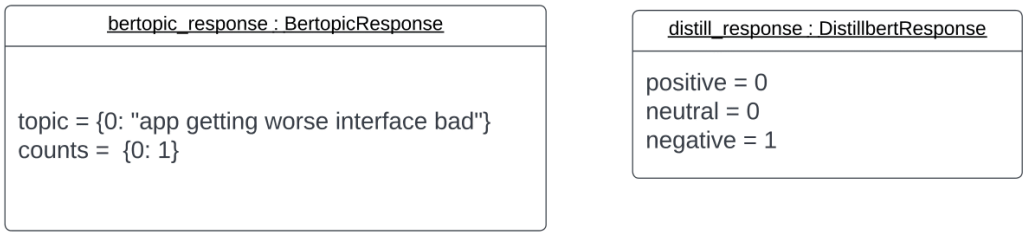

Figure 4.8 Object diagram of the objects passed to the frontend (source: own elaboration)

The DistillResponse class provides data for the pie plot, while the BertopicResponse class is used for the bar plot. If the user interacts with the bar plot by clicking on one of the bars, the request for more specific information will be sent.

```
            onClick={(e) => {
              // @ts-ignore
              const newReviewType = e.points[0].label as string;
              setCurrentReviewType("");
              setTimeout(() => setCurrentReviewType(newReviewType), 0);
            }}
```

```
const {
  refetch: fetchMoreData,
  data: reviewsData,
  isError: isFetchMoreDataError,
  error: fetchMoreDataError,
} = useGetMoreDataQuery(topicId, {
  skip: !shouldFetchMoreData && topicId === "",
});
```

The backend's response format is defined in the DetailedResponse class, which internally uses another class named ProcessedReview to offer more comprehensive details on the topic selected through the bar plot.

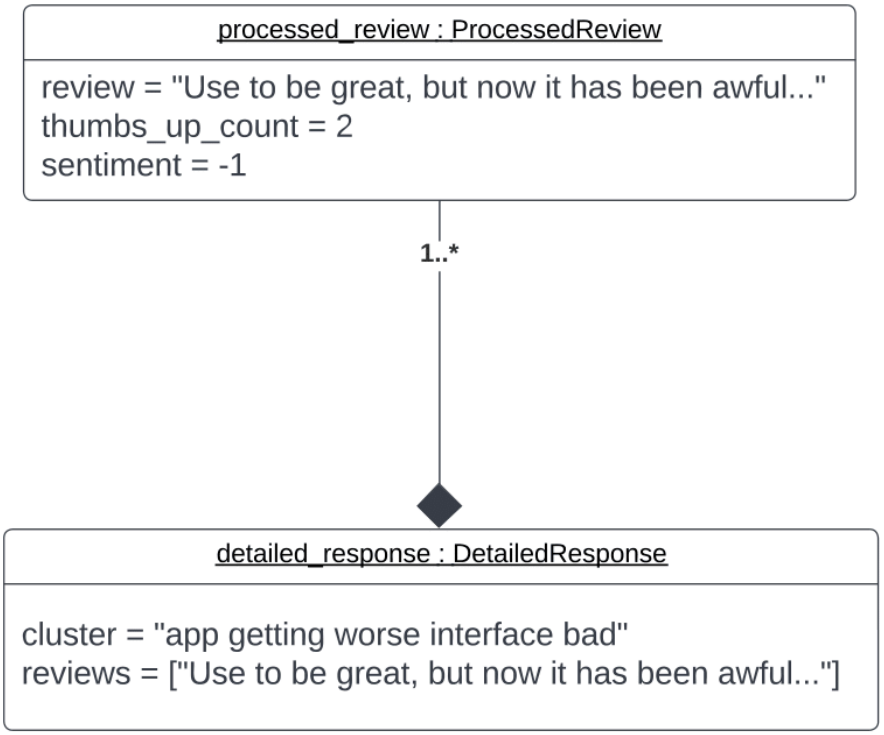

Figure 4.9 Class diagram of the ProcessedReview and DetailedResponse classes (source: own elaboration)

As depicted in Figure 4.9, there is a composition relationship between the DetailedResponse class and the ProcessedReview class.

4.3 Deployment diagram

Regarding the application deployment, two options are available. Users can run the application locally with machine learning endpoints on their local CPU or deploy them to a cloud computing platform. This thesis provides instructions for deploying them on the Google Cloud Platform.

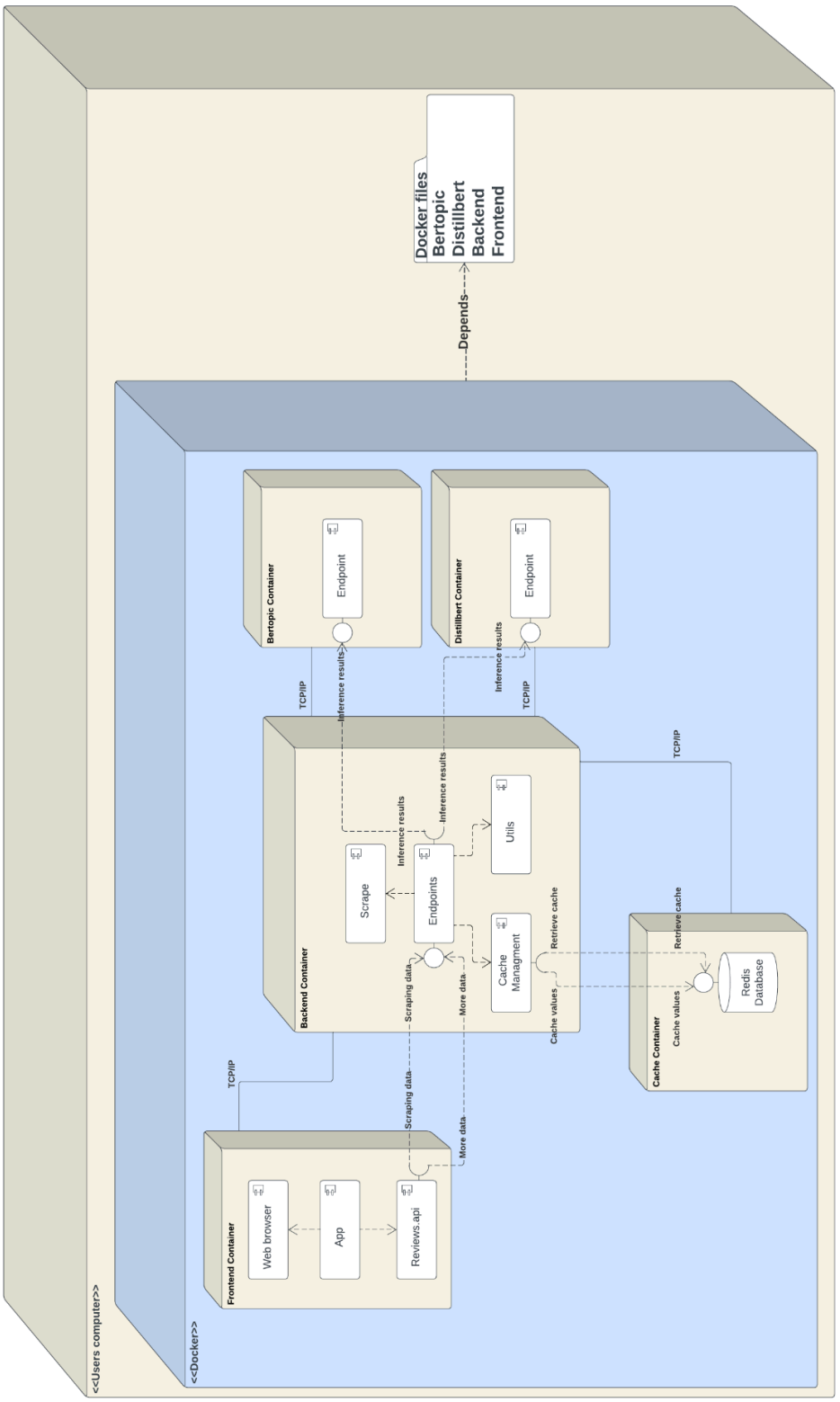

Figure 4.10 Deployment diagram of machine learning endpoints deployed locally (source: own elaboration)

As mentioned, the entire application runs inside a Docker environment. Each application component operates within a separate container, these containers communicate with each other via HTTP requests. The frontend segment requires a web browser to display the interface. User inputs are received by the frontend and sent to the backend through fetch queries defined in the Reviews.api module. The Endpoints Python module handles incoming requests. Depending on the type of request, data is either cached in the Redis database, reviews are scraped by the application in the Scrape module, or other review-related operations are performed from the utils module. The reviews are sent to the BERTopic and DistillBERT inference endpoints if the frontend requests inference from machine learning models. These AI endpoints perform inference on the provided data, and the results are received by the backend, converted into a format suitable for plotting, and returned to the frontend in order to present the results.

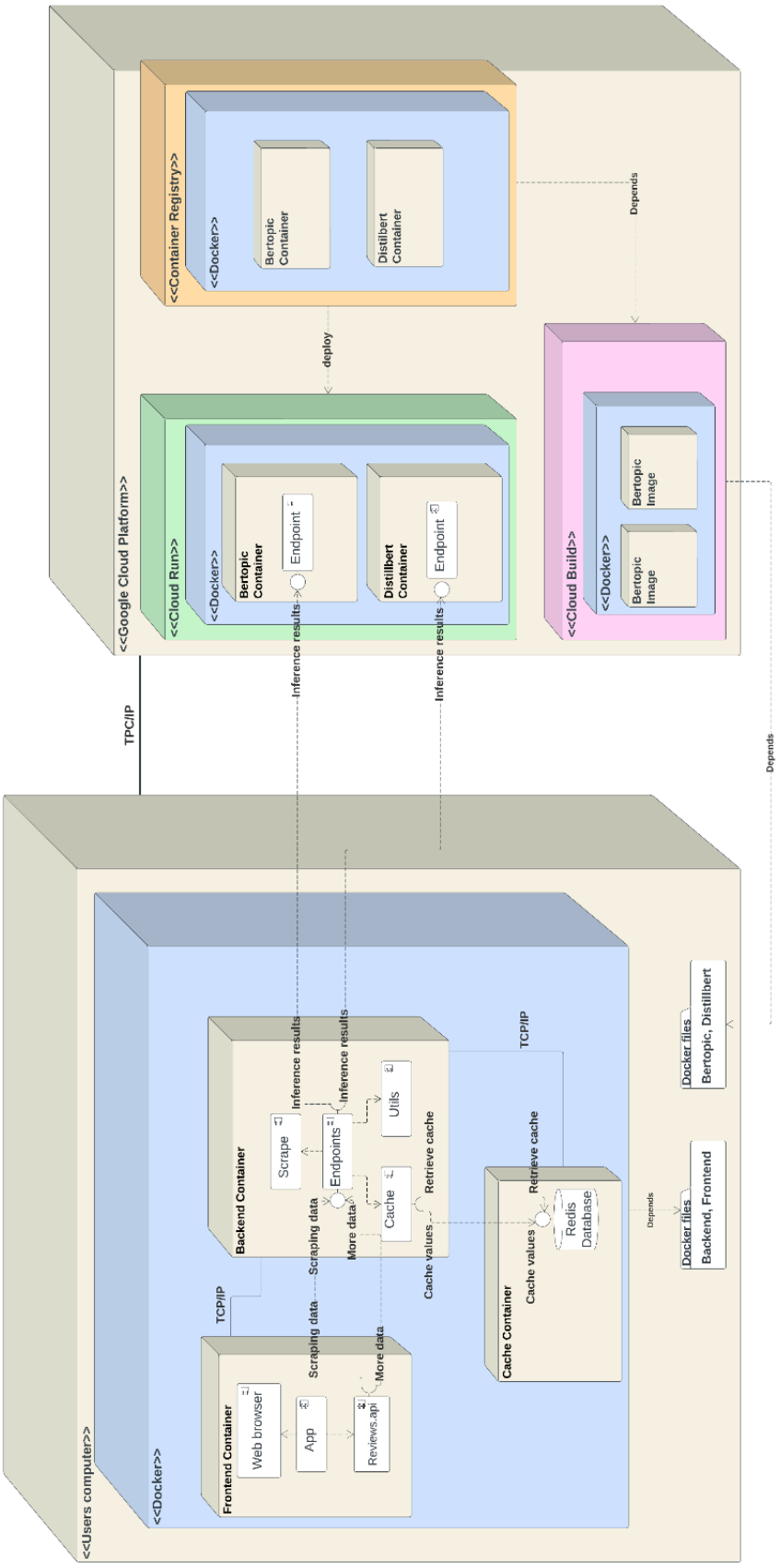

Figure 4.11 Deployment diagram of machine learning endpoints deployed on the Google Cloud (source: own elaboration)

However, if a user lacks the computing and memory resources required to run two machine learning models locally, they can deploy them on Google Cloud. In the case of cloud computing, the data flow remains the same as in the previous deployment version. However, the setup of the application differs. To deploy containerised endpoints to Google Cloud, the user must first build the image of the machine learning endpoints in the Google Cloud Platform called Cloud Build by providing a command that points to the Dockerfile responsible for creating the machine learning endpoints. After they are created, a Container Registry stores them. Later in order to deploy these containers a Cloud Run service is employed, which runs the machine learning models in the same way as local deployment. The user must then update the addresses of the machine learning endpoints with the ones provided by the Google Cloud Run service. This ensures that every time a model is inferred for a response, it runs on Google Cloud.

5. Project implementation and testing procedures

5.1 Implementation

The entire project was designed with a focus on distributed services, where each significant component

- frontend,
- backend,
- machine learning endpoints,
- the Redis database,

were developed to function independently. Each component can be deployed separately on its server, provided they have internet access to communicate. While deployment instructions are explicitly provided for the machine learning endpoints, the modular design allows for flexible deployment of various project parts as needed.
Docker played an essential role in orchestrating this distributed systems approach. It offered several advantages for deploying applications with containerised components.
Visual Studio Code IDE was chosen for its versatility and plugin support. Its plugin system allowed seamless transitions between programming languages and solutions during development, making it particularly useful for experimenting with different technologies.

5.1.1 Machine learning endpoints

The application primarily focuses on machine learning, particularly BERTopic and DistillBERT. These models were selected for topic generation, document classification, representative review selection, and sentiment classification. Both solutions are based on the BERT language model, a transformer-based language model.

The choice of these models was influenced by their well-known ability in the machine learning community to map natural text information to vectorised form, known as text embedding. This capability allows for text classification based on the direction of the corresponding vector. Depending on the embedding model used, this action can produce sentiment classification or allow for identification of themes within textual

data. BERTopic was chosen for its capacity to perform unsupervised clustering of textual data. On the other hand, DistillBERT was selected for its faster inference time for the sentiment classification task, compared to the standard BERT model, while maintaining a similar accuracy level. API endpoints enabling the inference of these models were created using the FastAPI library.

```
@router.post('/inference', response_model=TopicResponse)
async def inference(request: InferenceRequest):
   try:
       document_classification = model.fit_transform(request.reviews)[0]
       topics = []
       representative_reviews = []
       for _, topic, _, name, _, rep_docs in model.get_topic_info().itertuples():
           if topic == -1:
               continue
           topics.append(" ".join(list(set(name.split('_')[1:]))))
           representative_reviews.append(request.reviews.index(rep_docs[0]))
       response = TopicResponse(
           topics=topics,
           classification=document_classification,
           representative_reviews=representative_reviews)
       return response
```

After the creation of both endpoints, the next step was containerisation. Both models rely on many external libraries, making tokenisation of these inference endpoints essential.

```
COPY requirements.txt .

RUN apt-get update && \
   apt-get -y install gcc && \
   pip install --upgrade pip && \
   pip install -r requirements.txt --default-timeout=100

COPY ./app .
```

Following the containerisation process, the next step was deployment. The dockerised endpoints were deployed using the Cloud Build, Container Registry, Cloud Run services offered by GCP.

5.1.2 Backend

Similarly to the inference endpoints, the project's backend was primarily developed using the FastAPI library. It involved creating endpoints that utilised various data management, text preprocessing, and scraping functions. One significant challenge encountered during development was handling data retrieval from the inference endpoints asynchronously. The initial project plan aimed to build the backend as a single endpoint that would scrape data, send it to the inference endpoints, and simultaneously communicate the progress of each process. However, this approach did not allow for the immediate return of results from the endpoints as soon as the backend received them. To address this, a caching mechanism was introduced. It enabled data management independent of the processes related to endpoint management. As a result, separate endpoints were created to perform different tasks, such as data retrieval, passing reviews for inference, and transmitting additional inference information. Similar to the machine learning endpoints, the backend was dockerised. All endpoints, data management classes, and text preprocessing functions were bundled into a single Docker image. A Redis database was also run as a separate container to facilitate data caching.

5.1.3 Frontend

The Figma tool was used to design the minimalistic user interface.

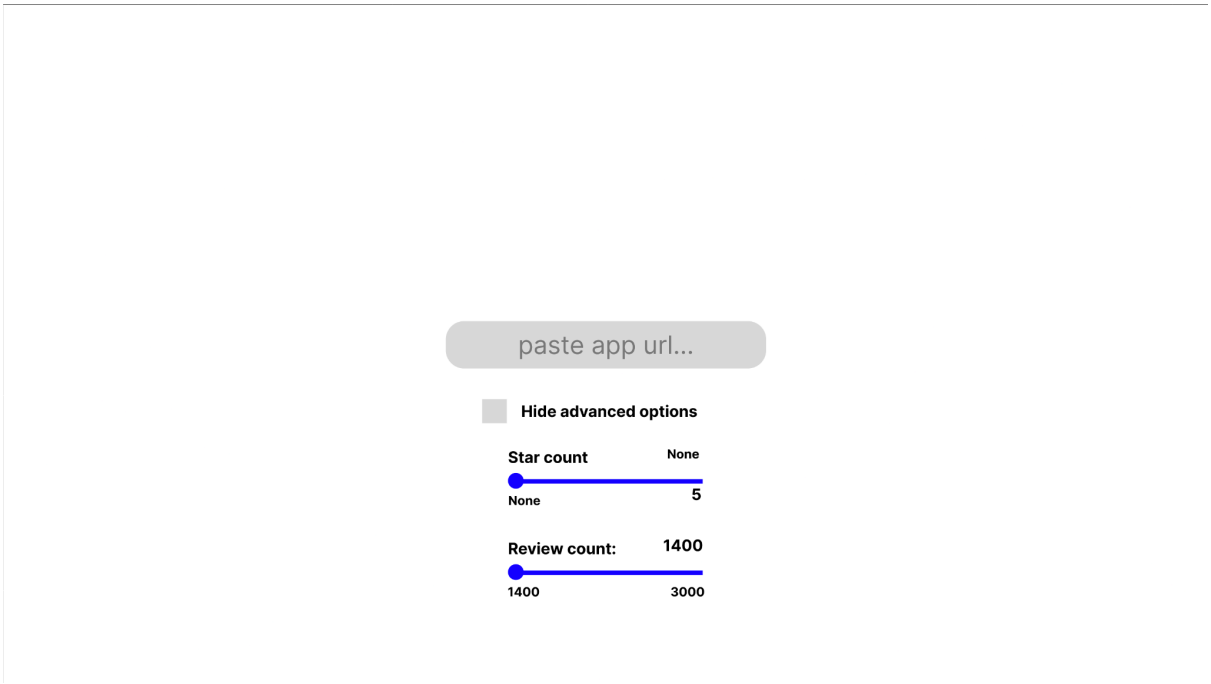

Figure 5.1 Design of the landing page in Figma (source: own elaboration)

However, despite the minimalist design philosophy, creating the interface proved challenging. The initial versions of the interface were built using HTML, CSS, and vanilla JavaScript, resulting in convoluted code, mainly due to the single-page design chosen for the application.

```
function pasteUrl() {
 var urlInput = document.getElementById('urlInput').value;
 if (verifyUrl(urlInput)) {
     encodedAppId = encodeURIComponent(extractId(urlInput));
     fetch(`${apiUrl}?app_id=${encodedAppId}`)
         .then(async (response) => {
         if (!response.ok) {
             const data = await response.json();
             switch (response.status) {
                 case 403:
                     displayErrorMessage(data.detail);
                     break;
                 case 404:
                     displayErrorMessage('Not found, check the URL and endpoints');
                     break;
                 default:
                     console.log(response);
                     displayErrorMessage(data.detail);
```

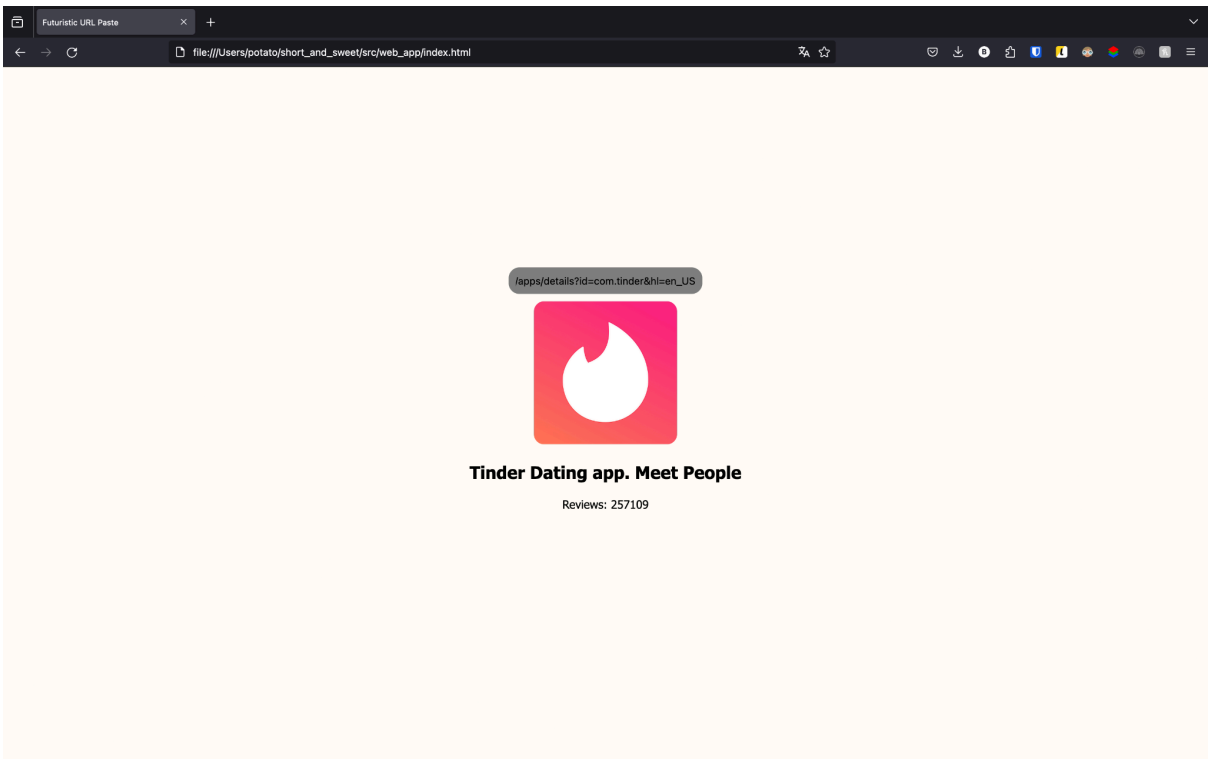

Figure 5.2 First prototype of the landing page written in vanilla JavaScript (source: own elaboration)

Figure 5.2 illustrates the outcome of the user interface created using the traditional stack. After JavaScript, HTML, and CSS, the go-to choice was the Streamlit Python library, designed to create user interfaces for AI-based applications. However, Streamlit's plotting solutions did not support additional "on-click" actions. After multiple iterations, a different stack was chosen. Chakra UI, a React library, and TypeScript (instead of JavaScript) significantly simplified UI development. The development process became more intuitive, especially when using type annotations with TypeScript, comparing Python's typing hints. Thanks to the Chakra UI, a Form visible in figure 7.2 didn’t require extensive CSS classes but a simple JSX component.

```
.futuristic-input {
 padding: 10px;
 border: none;
 border-radius: 15px;
 background-color: #818181;
 color: #000000;
 font-size: 14px;
 width: 250px;
 outline: none;
}
```

```
  <FormControl
    display={"flex"}
    flexDirection={"column"}
    justifyContent={"center"}
    alignItems={"center"}
    gap={"10px"}
  >
```

During the development of UI components, a method for sending requests and queries to the backend needed to be implemented. ReduxJS was chosen for this purpose. Once the user interface development was completed, the application was containerised using Docker, following the same approach as in the project's earlier phases.

5.2 Testing

At the end of the development of each submodule, a testing phase was run. The nature of the tests varied depending on the specific component being tested.

Machine Learning Endpoints

Due to their lengthy computation times, the machine learning endpoints were not tested using unit tests. Instead, a preprocessing pipeline was developed in the project's backend. This pipeline was based on the input data requirements and the known limitations of the machine learning models' embedding capabilities, ensuring smooth operation. Furthermore, the correctness of the inference endpoints was confirmed through every model inference during the project's testing phase.

Backend

In the backend testing, most were unit tests, focusing on individual elements like functions isolated from other backend systems.

```
def test_get_more_data_with_empty_cache(redis_client):
   for dependency in REDIS['dependencies']:
       redis_client.cache(dependency, [])
   assert all(data == [] for data in redis_client.get_more_data())
```

For instance, the 'test_get_more_data_with_empty_cache' function evaluates how 'RedisClient' manages empty values, possibly arising from an unsuccessful scraping process, resulting in an empty review list. This function ensures that 'None' or blank values are not stored in the cache.

```
def validate_url(url: str) -> bool:
   return bool(url_pattern.match(url))
```

Another aspect of the backend's functionality that was tested was URL verification. This function checks if a URL fits the RegEx pattern defined in one of the backend config files. Before the tests were performed, a simple check determined if the URL contained the 'hl=en' segment, verifying it was from the English version of the Google Play Store's website.

```
+&hl=en(&.+)*$
*hl=en[^&]*)(&[^&]+)*$
```

However, this method only searched for the 'hl=en' pattern at the string's end, not allowing variations like 'en_US'. Consequently, URLs like 'https://play.google.com/store/apps/details?id=com.google.android.apps.maps&hl=en_US' were incorrectly rejected due to the '_US' following the 'hl=en' pattern, even though 'en_US' indicates English-language reviews. The source code for these testing functions can be found in the project's 'tests' directory.

Frontend

The frontend part of the application was tested thoroughly, mainly through manual tests. A key bug found during testing related to URL handling. If the backend rejected a URL because of incorrect formatting, it sent an error message to the frontend, which was then shown to the user.

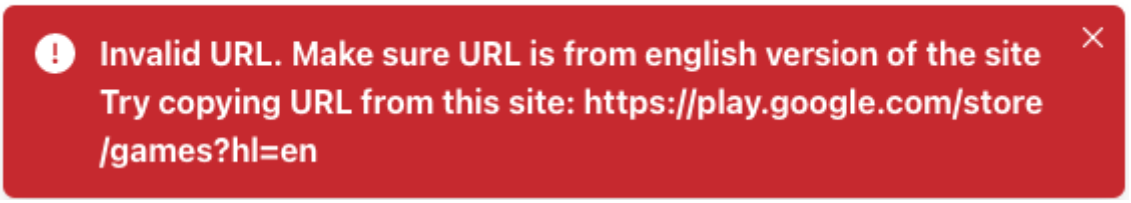

Figure 5.3 Error popup shown after the URL is rejected by the backend's validation process (source: own elaboration)

When a user pasted a URL a second time, the data was mistakenly sent to the backend immediately, bypassing the need for the user to click the 'Send' button. This issue occurred because the state 'shouldSendRequest', which prevents an immediate backend query, was only set correctly when the 'Send' button was first clicked.

```
const { isLoading, isSuccess, error, isError } = useGeStAppDataQuery(
  { url, stars, reviews },
  {
    skip: !shouldSendRequest,
  }
);
```

```
const handleKeyDown = async () => {
  setShouldSendRequest(true);
};
```

If the query did not yield a 200 result code, this state was not reset to 'false', which would have prevented an immediate backend query.

```
useEffect(() => {
  if (isSuccess) {
```

```
    navigate(AppRoutePaths.RETRIEVING_REVIEWS);
  setShouldSendRequest(false);
  }
```

Modifying the useEffect hook, as shown below, resolved the issue, preventing an immediate query after a new URL was pasted following an unsuccessful attempt.

```
useEffect(() => {
  if (isSuccess) {
    navigate(AppRoutePaths.RETRIEVING_REVIEWS);
    setShouldSendRequest(false);
  }

  if (isError) {
    toast({
      //@ts-expect-error
      title: error?.data?.detail || "",
      status: "error",
      duration: 7000,
      isClosable: true,
    });

    setShouldSendRequest(false);
  }
```

6. Installation of the project

Since the project's target audience is the developer community, detailed setup instructions for well-known tools like Git, Google Cloud SDK, or Docker are not included in this chapter. The processes for initialisation and usage of this project are detailed in the README.md files in the project's source code.
To download the project's source code, the user must pull it from a GitHub repository with the following command in the command line.

```
git clone git@github.com:bjpietrzak/short_and_sweet.git
```

Which will clone the repository to the directory in which they activated the command. As a result of the previous operation, a new directory will appear called ‘short_and_sweet’. The pulled directory has the following form.

```
.
├── LICENSE
├── README.md
├── src
└── tests
```

The ‘src’ directory contains the docker-compose file which launches the whole application.
The user has two possible ways to run the application. The first demands the least effort from the user, and the second – as mentioned earlier – will require the user to deploy machine learning endpoints on the Google Cloud Platform.

6.1 Simple activation

To activate the application, one has to enter the earlier mentioned ‘src’ directory and type the following command.

```
docker-compose up --build
```

The command will activate the application's initialisation process. First, every application component will be built from the respective Dockerfiles, which can take half an hour.

6.2 Advanced activation

To deploy machine learning endpoints to the Google Cloud Platform, the user has to create an account. If the account is freshly made, a free trial offer allows for the free usage of the services provided on the platform. Once the user is logged in, they need to create a project. After completing this task, a project ID will be presented to them, and they should note it for further use. By navigating to the provided link, they may follow the instructions on the web page [19] in order to create a project. Each machine learning container will require one Cloud Run deployed endpoint; however, each Google Cloud project should only contain one Cloud Run endpoint, which means the user has to repeat the process of creating the project for a second time.
After the creation of two projects, users should activate Cloud Build [4], Container Registry [6], and Cloud Run [5] services. Once the services are active, users may return to the source code directory and initialise the Google Cloud SDK. After SDK is turned on, the user may move to the next step of deploying docker containers to the Cloud Build service. Since the following steps require definitions of service server localisation, some parameters will be predefined; the user may change them in any shape or form.
To deploy one endpoint, the user must enter one of the directories ‘src/ai/<bertopic or distilbert>/. The following process must be performed separately for both directories.

```
gcloud builds submit --tag gcr.io/<project_id>/inference
```

The project-id segment must be replaced by the project ID acquired on activating the Google Cloud Project. This command will build and containerise a docker image in the Google Cloud. Then, the user must run the following command to deploy the endpoint on the Cloud Run Service.

```
gcloud run deploy --image=gcr.io/<project-id>/inference:latest \
--execution-environment=gen2 \
--region=europe-central2 \
--project=<project-id>
```

If either of these steps results in an error, users can refer to the README.md documentation for additional instructions to troubleshoot the issues.
Once the endpoints are successfully deployed, URLs for sending requests will be provided via the command line interface. The user must then update the 'src/backend/app/configs/endpoints.json' file in the source code. They should replace the predefined 'bertopic' and 'distilbert' values with the URLs pointing to the deployed services.

```
  "bertopic": "http://model:8000/inference/bertopic",
  "distilbert": "http://model:8000/inference/distilbert",
```

Finally, after updating these values, users must 'comment out' the machine learning services in the docker-compose file, as mentioned in the previous section. This step is necessary to prevent the activation of machine learning endpoints locally.

```
# bertopic:
#   build: ./ai/bertopic
#   image: bertopic:latest
#   ports:
#     - 8080:8080
#   environment:
#     - PORT=8080

# distilbert:
#   build: ./ai/distilbert
#   image: distilbert:latest
#   ports:
#     - 8081:8081
#   environment:
#     - PORT=8081
```

Then, the user can use the command 'docker-compose up build' command to install all the dependencies necessary to activate the project.

7. Use of the application

Once the application is launched, it can be accessed by the URL address: ‘http://localhost:3000/’. A first-page interface will be visible.

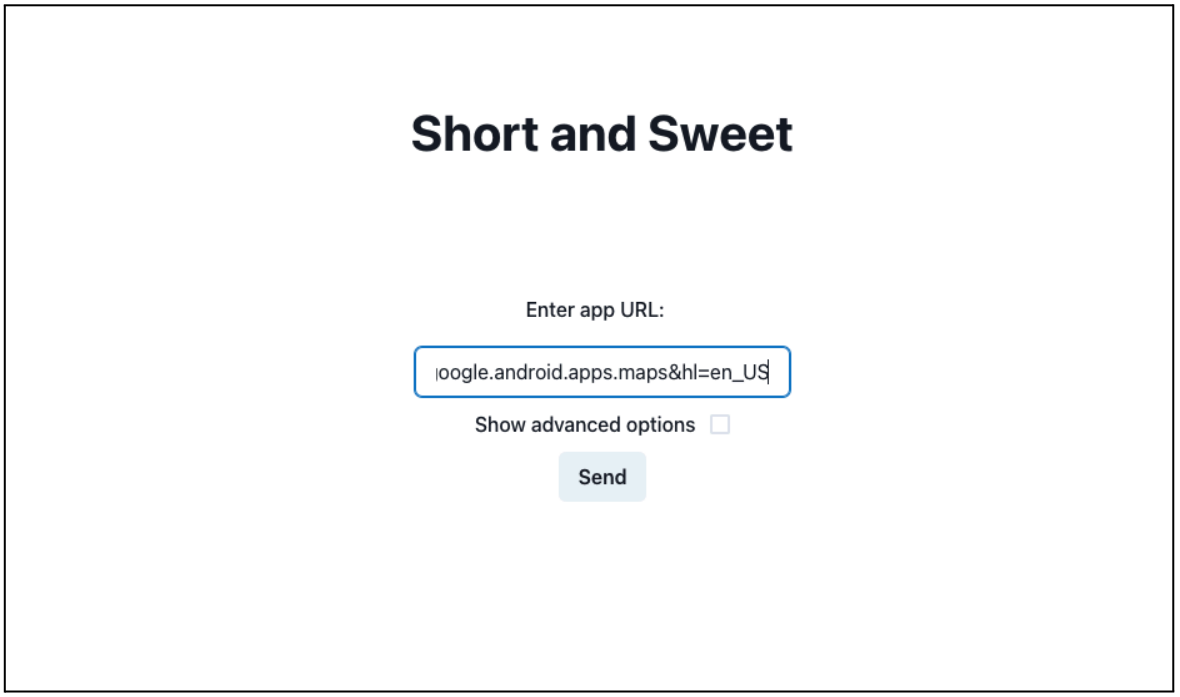

Figure 7.1 Landing page of the application (source: own elaboration)

Users may enter the URL of a picked application in the form field. They may also click ‘Show advanced options’ to view a list of sliders that allow them to filter the review scrapping process further.

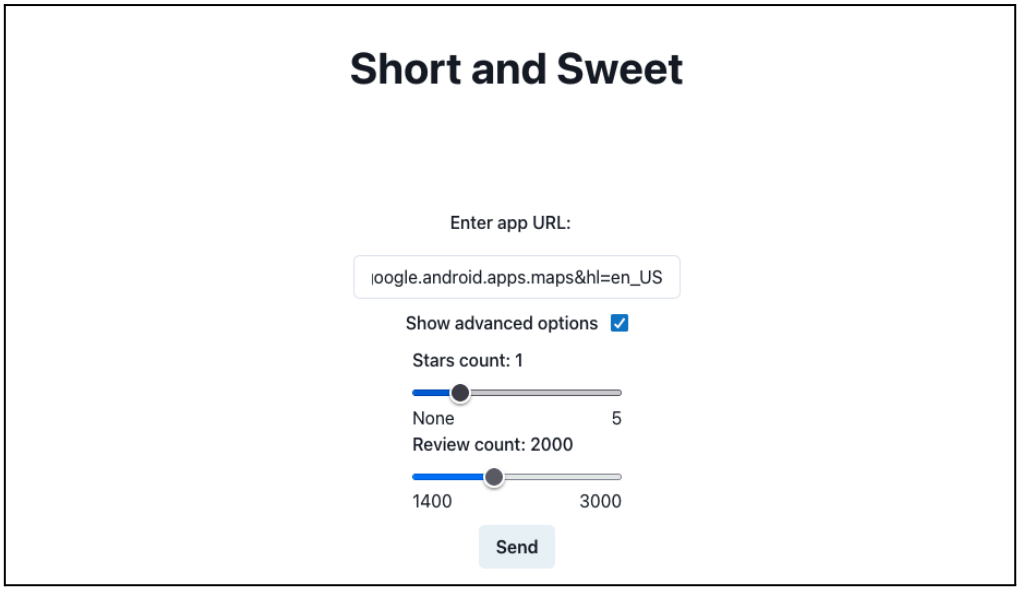

Figure 7.2 Advanced options tab (source: own elaboration)

If the application did not receive the required number of reviews or the URL does not point towards the English version of the Google Play Store website, the error will be displayed after passing the input through the ‘Send’ button.

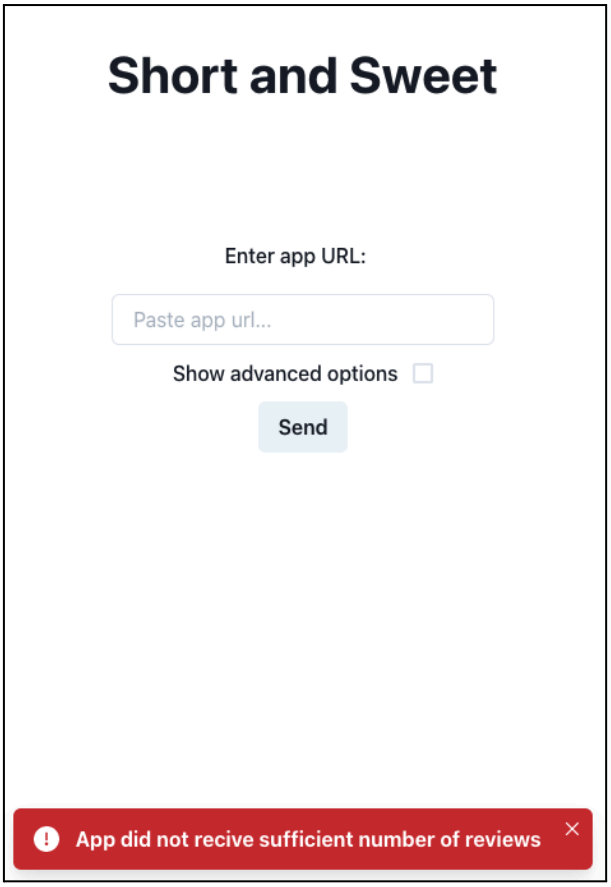

Figure 7.3 Not enough reviews error popup (source: own elaboration)

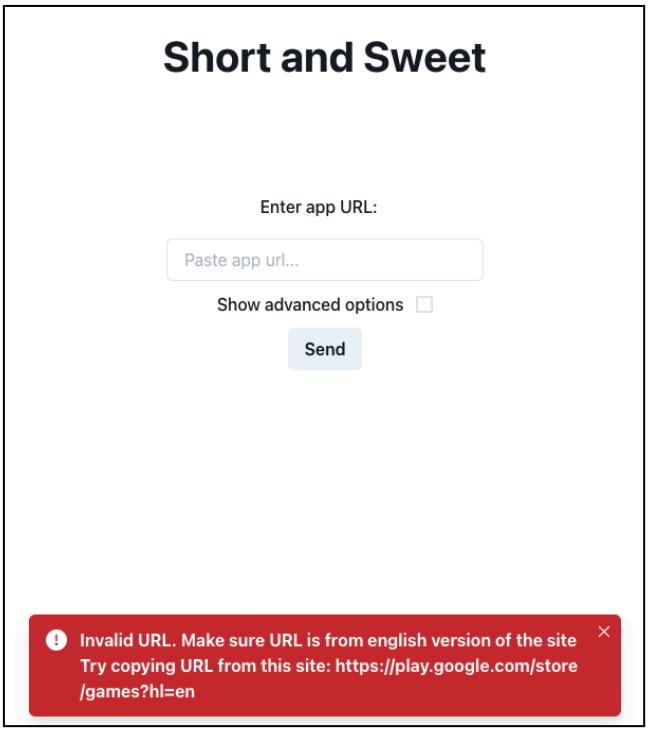

Figure 7.4 Invalid URL error popup (source: own elaboration)

If, however, the input is correct and the application receives enough reviews, a scrapping process will begin.

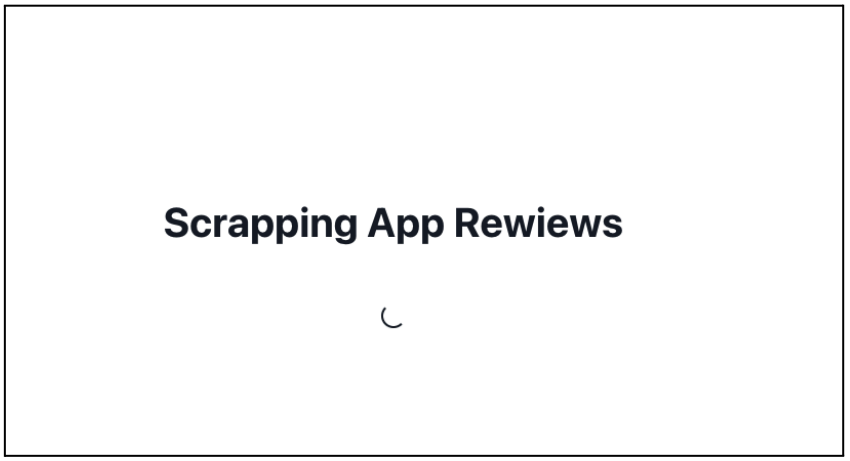

Figure 7.5 Scrapping app reviews awaiting screen (source: own elaboration)

When the application data is retrieved, the application icon will be displayed along with its title. At the same time, the machine learning inference process will begin.

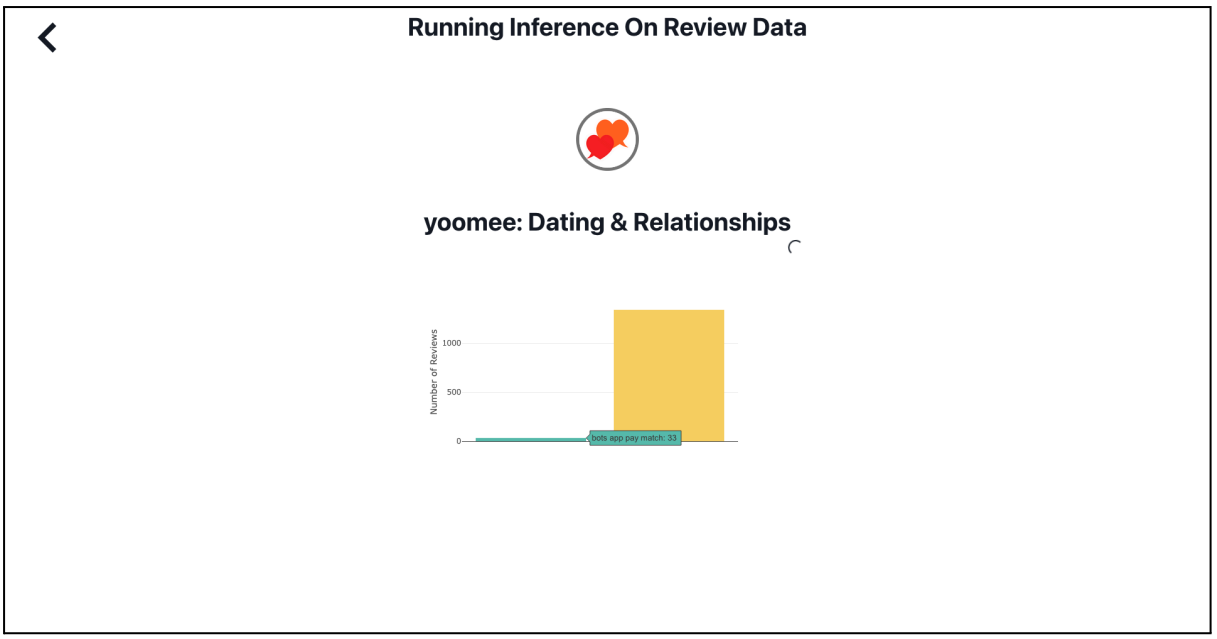

Figure 7.6 Awaiting for the results screen (source: own elaboration)

When the results from one of the endpoints are retrieved, they will be automatically displayed as a plot without awaiting the result of the second model. When both endpoints return their results, users may interact with the plots by hovering or clicking on them.

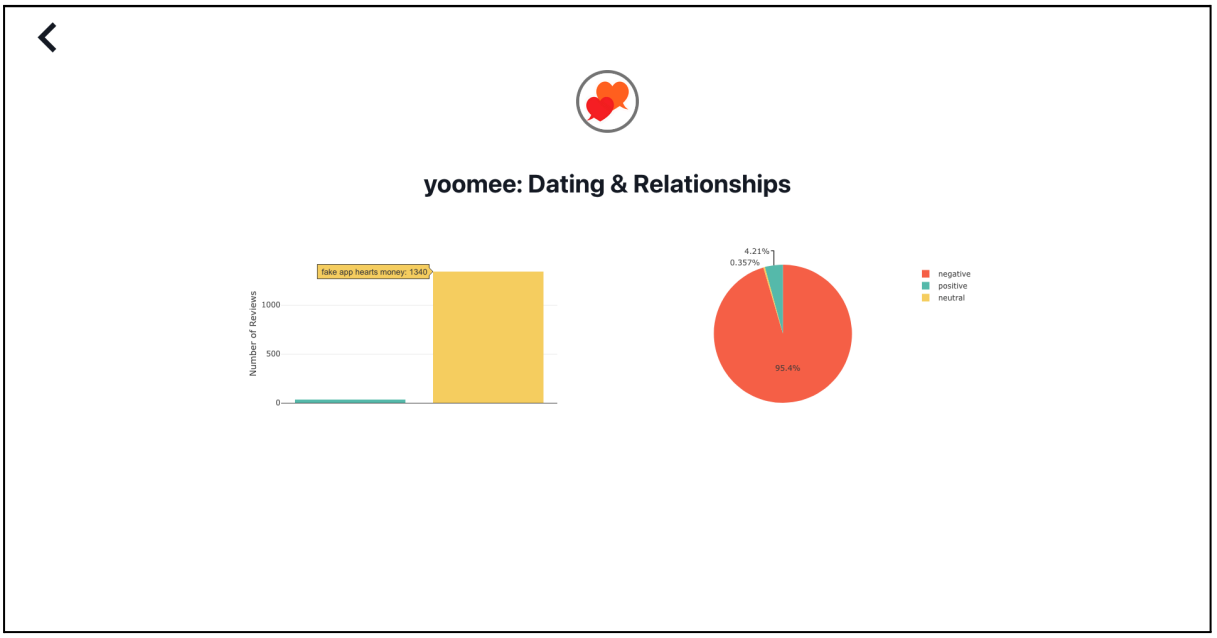

Figure 7.7 Finalised inference screen (source: own elaboration)

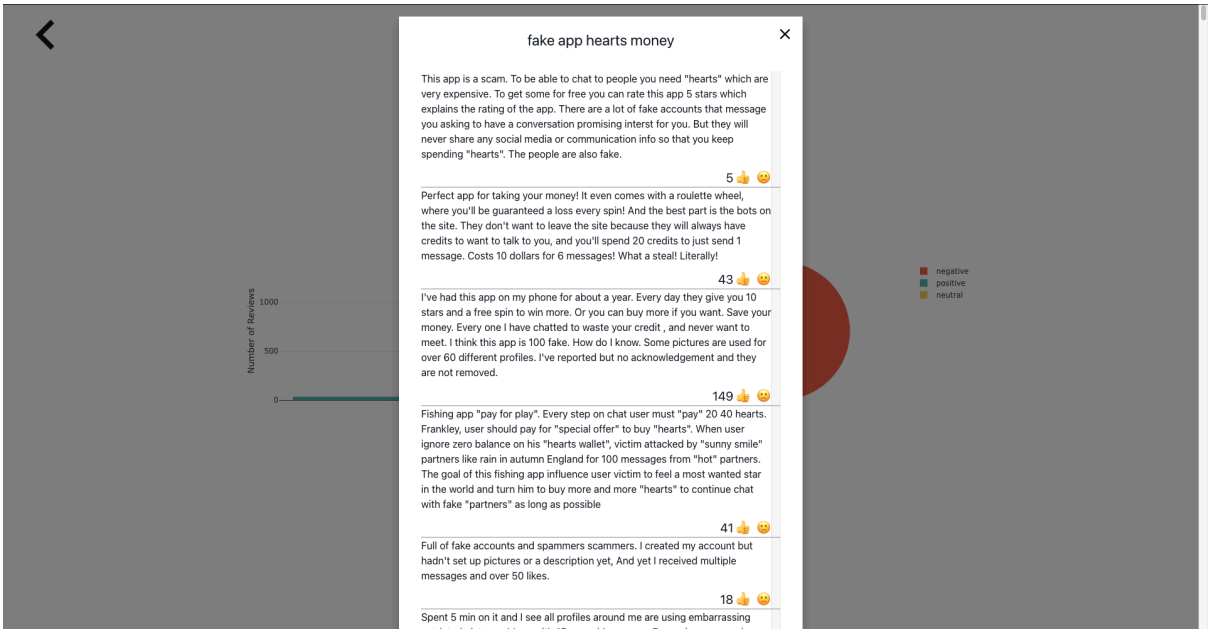

Figure 7.8 Detailed view (source: own elaboration)

When hovering over the bar plot, a topic will be displayed, with the count value at the side, to provide information on how many reviews have been classified to the given topic. Users can view the topic detail window if they click on the bar. It will show topic keywords extracted from the reviews. In the case of Figure 7.8, it is 'fake app hearts money'. Then, below the topic, the review that represents the topic the most will be presented below. The rest of the classified reviews will be displayed underneath the representative review. The thumbs-up count and emoticons representing the sentiment of each review can be found at the bottom of each review.

Users may switch between the topics, view their details, or go back to the main first page of the application, clearing all of the rendered plots by clicking the go back black arrow in the upper left corner.

8. Summary

During the development of this project, much knowledge had to be acquired. Because of this, many skills have evolved. A Google Cloud Platform has been discovered and learned thanks to the slowness of the inference processes in machine learning endpoints. Because of the limitations of the Streamlit library for frontend functionalities, a new tech stack has been acquired.

Additionally, after the project's development, additional improvements emerged during the application analysis. However, because of the time limits, they could not be implemented. The list of possible improvements will be listed in this subchapter.

8.1 Possible Improvements

During the development of machine learning endpoints, a daunting realisation came that the only way to run the inference in Docker is on the CPUs of the host machine. This is because Docker images usually do not support accelerated computations with GPUs. However, it is possible to implement the GPU computation when running machine learning endpoints in Docker, thanks to Nvidia's base images [21]. Unfortunately, these images require many more preinstallation steps on the part of the Dockerfile to make them function correctly.

An additional increase in computation speeds could be obtained through Nvidia's Triton Server [34]. A Triton server allows for hosting machine learning models in one robust and GPU-accelerated environment. Thanks to this solution, if a user has acquired a GPU capable of CUDA acceleration, it provides a much more efficient way of running models. It is also possible to deploy the Triton Server on a Google Cloud Platform service called Vertex AI [37]. Unfortunately, as it will be a common theme in this subchapter, a time requirement and tech overhead were way too demanding to implement into the project.

Another way the project could be improved would be an additional websocket utility provided by FastAPI. Instead of creating unidirectional endpoints, a websocket could communicate with the client and machine learning endpoints. As a result, analysis progress could be better-visualised thanks to the more accessible communication between the nodes thanks to the websocket implementation.

As the development of the frontend part of the application was approaching its final stages, a framework that could solve most of the React libraries was discovered –

SveltKit [32]. For instance, instead of using Redux, React Server and additional dependencies with the React app, a SveltKit has all the features with no pre-installation steps required. It has been used in the project mentioned in the Market Analysis Chapter of this thesis [1]. The source code of the frontend part of this project seems more intuitive than Reacts equivalent operations. It will be further explored in upcoming personal projects.

9. Bibliography

[1] Automtic1111, Stable diffusion web UI https://github.com/AUTOMATIC1111/stable-diffusion-webui, (access: 15.01.2024)

[2] Bhagtani L., *Redis as Cache: How it Works and Why to Use it*, https://codedestine.com/redis-cache-how-works-why-use/, (access: 15.01.2024)

[3] Chakra UI, https://chakra-ui.com/, (access: 15.01.2024)

[4] Cloud build, https://console.cloud.google.com/cloud-build/, (access: 15.01.2024)

[5] Cloud run, https://console.cloud.google.com/run/, (access: 15.01.2024)

[6] Container registry, https://console.cloud.google.com/gcr/, (access: 15.01.2024)

[7] Dall-e 2, https://openai.com/dall-e-2 (access: 15.01.2024)

[8] Docker, https://www.docker.com/, (access: 15.01.2024)

[9] Figma, https://www.figma.com/, (access: 15.01.2024)

[10] Git, https://git-scm.com/ (access: 15.01.2024)

[11] Github, https://github.com/ (access: 15.01.2024)

[12] Google Cloud Platform, https://cloud.google.com/, (access: 15.01.2024)

[13] Grootendorst M., *BERTopic*, https://maartengr.github.io/BERTopic/index.html/, (access: 15.01.2024)

[14] Grootendorst M., *BERTopic: Neural topic modeling with a class-based TF-IDF procedure*, ArXiv, 2022, pp. 5-6

[15] Httpx, https://www.python-httpx.org/, (access: 15.01.2024)

[16] Hugging Face, Transformers, https://github.com/huggingface/transformers/, (access: 15.01.2024)

[17] JoMinguy, https://github.com/JoMingyu/google-play-scraper, (access: 15.01.2024)

[18] Monkeylearn, Text Analitics, https://monkeylearn.com/ (access: 15.01.2024)

[19] New project page, https://console.cloud.google.com/projectcreate/, (access: 15.01.2024)

[20] Numpy, https://numpy.org/, (access: 15.01.2024)

[21] Nvidia/cuda container, https://hub.docker.com/r/nvidia/cuda/, (access: 15.01.2024)

[22] Plotly, https://plotly.com/javascript/, (access: 15.01.2024)

[23] Pydantic, https://docs.pydantic.dev/, (access: 15.01.2024)

[24] Pytest, https://docs.pytest.org/en/7.4.x/, (access: 15.01.2024)

[25] Python 3.11 https://www.python.org/downloads/release/python-3110/, (access: 15.01.2024)

[26] Python Redis library, https://redis.io/docs/connect/clients/python/, (access: 15.01.2024)

[27] Pytorch, https://pytorch.org/, (access: 15.01.2024)

[28] React, https://react.dev/, (access: 15.01.2024)

[29] Redis, https://redis.io/, (access: 15.01.2024)

[30] Redux, https://redux.js.org/, (access: 15.01.2024)

[31] Sanh V., Debut L., Chaumond J., Wolf T. *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*, ArXiv, 2019, pp. 3

[32] Sveltkit, https://kit.svelte.dev/, (access: 15.01.2024)

[33] Tiangolo, FastAPI, https://fastapi.tiangolo.com/, (access: 15.01.2024)

[34] Triton inference server, https://developer.nvidia.com/triton-inference-server/, (access: 15.01.2024)

[35] Typescript, https://www.typescriptlang.org/, (access: 15.01.2024)

[36] Uvicorn, https://www.uvicorn.org/, (access: 15.01.2024)

[37] Vertex ai, https://cloud.google.com/vertex-ai/, (access: 15.01.2024)

[38] Visual studio code, https://code.visualstudio.com/, (access: 15.01.2024)

[39] Vite, https://vitejs.dev/, (access: 15.01.2024)

10. Index of figures

11. Index of tables

Abstract

The current internet is filled with unstructured information. Reviews, blog posts and emails require our attention to extract the information. However, the more channels pass their data towards us, the more challenging it becomes to keep up with them. For example, in application reviews, developers who want to understand their users need to monitor their application reviews on websites like Google Play Store to understand users' sentiments and main topics about their product. This project thesis aims to summarise application reviews with the help of AI. Short and sweet, an application that allows one to understand general sentiments and themes running through thousands of application reviews presented in a minimalistic and aesthetic way with pie and bar charts. The project's backend was built with Python scripting language, depending on Machine Learning model DistillBERT and topic modelling technique BERTopic. The application's front end was written in the React library with ChakraUI components. The whole application was also containerised for seamless installation and cloud deployment.

Keywords: machine learning, cloud development, artificial intelligence, web development.

Streszczenie

Obecny Internet jest wypełniony nieustrukturyzowanymi informacjami. Recenzje, posty na blogach i e-maile wymagają naszej uwagi, aby wyodrębnić ukryte w nich informacje. Jednak im więcej kanałów przekazuje nam swoje dane, tym trudniej jest za nimi nadążyć. Na przykład w przypadku recenzji aplikacji; deweloperzy, którzy chcą zrozumieć swoich użytkowników, muszą monitorować recenzje w witrynach takich jak Sklep Google Play, aby zrozumieć nastroje użytkowników i główne tematy dotyczące ich produktu. Niniejsza praca dyplomowa ma na celu podsumowanie recenzji aplikacji za pomocą sztucznej inteligencji. “Do rzeczy” to aplikacja, która pozwala zrozumieć ogólne nastroje i tematy przewijające się w tysiącach recenzji danej aplikacji, przedstawionych w minimalistyczny i estetyczny sposób za pomocą wykresów kołowych i słupkowych. Backend projektu został zbudowany przy użyciu języka skryptowego Python, współpracującego z modelami uczenia maszynowego, takimi jak BERTopic i DistillBERT. Frontend aplikacji został napisany w bibliotece React z komponentami ChakraUI. Cała aplikacja została również skonteneryzowana w celu bezproblemowej instalacji i wdrożenia w chmurze.

Słowa kluczowe: uczenie maszynowe, rozwiązania chmurowe, sztuczna inteligencja, tworzenie stron internetowych.

COLLEGIUM DA VINCI

ARKUSZ PODZIAŁU PRAC WYKONYWANYCH PRZEZ STUDENTÓW W RAMACH REALIZACJI WIELOAUTORSKICH (ZESPOŁOWYCH) PRAC INŻYNIERSKICH NA WYDZIALE NAUK STOSOWANYCH COLLEGIUM DA VINCI W POZNANIU

Temat pracy: Short And Sweet: Comprehensive Approach for Summing up Application's Reviews using NLP

	imię	nazwisko
WSPÓŁAUTOR 1:	Bartosz	Pietrzak
WSPÓŁAUTOR 2:	..... imię	..... nazwisko
WSPÓŁAUTOR 3:	..... imię	..... nazwisko
WSPÓŁAUTOR 4:	..... imię	..... nazwisko
WSPÓŁAUTOR 5:	..... imię	..... nazwisko

	tytuł/stopień	imię	nazwisko
PROMOTOR:	dr	Tomasz	Tyksiński

ul. Gen. Tadeusza Kutrzeby 10
61-719 Poznań
61 271 10 10 / info@cdv.pl

1. Całościowy udział w pracy określony procentowo przez studenta weryfikowany przez promotora w procesie dyplomowania:

	PROCENTOWY UDZIAŁ WSPÓŁAUTORA W PRACY	SUMA W %
WSPÓŁAUTOR 1	100 %	100%
WSPÓŁAUTOR 2		
WSPÓŁAUTOR 3		
WSPÓŁAUTOR 4		
WSPÓŁAUTOR 5		

2. Procentowy udział współautorów pracy w realizacji zadań wynikających z wymogów stawianych inżynierskim pracom dyplomowym*:

LP	OPIS ZADANIA	PROCENTOWY UDZIAŁ WSPÓŁAUTORA W WYKONANIU ZADANIA					SUMA W %
		WSPÓŁAUTOR 1	WSPÓŁAUTOR 2	WSPÓŁAUTOR 3	WSPÓŁAUTOR 4	WSPÓŁAUTOR 5	
1	Opracowanie wstępu pracy	X					100%
2	Określenie aktualnego stanu wiedzy	X					100%
3	Określenie celu, zakresu projektu i podział zadań w projekcie	X					100%
4	Opracowanie metodyki pracy	X					100%
5	Wytworzenie (kodowanie) front-end	X					100%
6	Wytworzenie (kodowanie) back-end	X					100%
7	Przygotowanie dokumentacji zgodnej z wymaganiami inżynierii oprogramowania	X					100%
8	Przygotowanie opisu projektu	X					100%
9	Procedura testowania oprogramowania	X					100%
10	Przygotowanie opisu sposobu wdrożenia/instalacji/eksploatacji	X					100%
11	Opracowanie podsumowania, wniosków	X					100%
12	Edycja i redakcja pracy (zgodnie z obowiązującym dokumentem – Wymogi Formalne - Informatyka	X					100%
13	Inne**						100%

* Proszę wypełniać tylko zadania realizowane w ramach pracy.

ul. Gen. Tadeusza Kutrzeby 10
61-719 Poznań
61 271 10 10 / info@cdv.pl

** W wierszu „inne" proszę wpisać zadanie, które nie zostało uwzględnione powyżej.

Załącznik stanowi podstawę oceny pracy indywidualnego wkładu każdego studenta w realizacji pracy dyplomowej i stanowi integralną część pracy dyplomowej jako ostatnia nienumerowana strona pracy.

Załącznik podpisują autorzy pracy i jest on zatwierdzony przez promotora.

Realizacja poszczególnych zadań przez autorów zależna jest od charakteru pracy.

WSPÓŁAUTOR 1: Bartosz Piotrowski
.....
podpis

WSPÓŁAUTOR 2:
.....
podpis

WSPÓŁAUTOR 3:
.....
podpis

WSPÓŁAUTOR 4:
.....
podpis

WSPÓŁAUTOR 5:
.....
Podpis

PROMOTOR:
.....
podpis

Poznań, 19.01.2024
miejscowość data

ul. Gen. Tadeusza Kutrzeby 10
61-719 Poznań
61 271 10 10 / info@cdv.pl