university of groningen

faculty of science and engineering

Sentiment Analysis Pipeline

Short Programming Project

Mohammad al Shakoush

s4274865

m.al.shakoush.1@student.rug.nl

Contents

1 Introduction

1.1 Problem description

This project centers on constructing a sentiment analysis pipeline for analyzing sentiment in a dataset mined from GitHub issues and commits. The mining of the dataset was previously accomplished in a bachelor's project [1]. The data collected was related to changes made by developers to Infrastructure-as-Code (IaC) artifacts. The aim of this project is to construct a pipeline that is not just tailored to this specific purpose but can be used for any piece of text or document.

1.2 IaC artifacts

IaC is a DevOps technique that enables machine-readable definition files to be used for the management and provisioning of infrastructure, as opposed to interactive configuration tools or manual configuration of physical hardware [2]. The emergence of Cloud computing has enabled IaC, a practice that involves the programmatic provisioning, configuration, and management of computational resources, to become increasingly commonplace [2].

1.3 Sentiment

Using Sentiment Analysis, one can conclude the "sentiment" of that text [3]. Where a sentiment describes the opinion and subjectivity. By doing sentiment analysis on the developer's commits and issues, an insight can be formed into the developers' decision-making and actions related to changes in the IaC artifacts.

1.4 Pipeline

A pipeline is the segmentation of a (long) process into sub-processes. Where the execution of successive sub-processes can be done in an overlapped mode [2, 4].
Pipelining is the process of breaking down a repeated sequential process into smaller processes. A pipeline can be carried out either in parallel or sequentially, depending on the nature of the sub-processes involved. If the sub-processes do not depend on each other, they can be executed in parallel, resulting in faster completion of the entire process. However, when successive sub-processes require the output of the preceding sub-processes, the pipeline must be executed sequentially. [2, 4].
Due to the nature of pipelining, we get the following advantages when using a pipeline for our project [5]:

- **Modularity**: Pipelining facilitates the segmentation of sentiment analysis into smaller, separate modules with distinct functions, thus rendering the process of development, testing, and modification of each module more attainable without affecting the efficacy of the entire system. So each model can have its own pre-processor and both pre-processing and the extraction of the sentiment could be conducted independently from the entire system.

- **Flexibility**: Pipelining offers flexibility for selecting and combining various models and tools for each module of the pipeline, which facilitates customisation and optimisation of the sentiment analysis process according to specific requirements. Therefore, modules or packages needed for one module do not have to be included in others. Which in turn, optimizes the performance of the pipeline.

- **Efficiency**: Pipelining facilitates parallelism and concurrency by allowing multiple stages of the pipeline to be executed in parallel. This enhances the performance and speed of our project, thereby improving its overall efficiency. Multiple models and their associated pre-processors can be executed concurrently since each will be running in a separate module.
- **Parallelization**: Pipelining can be utilized to parallelize the sentiment analysis process, thereby enhancing the system's overall speed and throughput. By breaking down the sentiment analysis process into multiple modules, each module can be executed in parallel on different computing resources, such as CPUs, GPUs, or cloud instances. This allows for more efficient utilization of computational resources and an overall improved performance. This point is not thoroughly tested in this project since it is outside of its scope. However, it is worth mentioning.
- **Throughput**: Pipelining can enhance the overall throughput of the system by permitting the concurrent processing of multiple inputs. This facilitates the handling of a greater volume of input data at a quicker rate.

1.5 Project's Scope

The purpose of this project is to automate the sentiment analysis of a given text. The data required for this, including the text to be analyzed, was obtained from a prior undergraduate project [1]. This project serves as an extension of that.

This dataset contains text from commits and issues scraped from the version control system GitHub[1], which has been labeled by the project owners. Furthermore, the sentiment of the text has been retrieved using three pre-trained Natural Language Processing (NLP) models: TextBlob[2], VADER[3], and Stanza NLP [4]. As such, this project will adhere to these three models and no training of an NLP model will be attempted.

[1] https://github.com/
[2] https://textblob.readthedocs.io/en/dev/
[3] https://github.com/cjhutto/vaderSentiment
[4] https://stanfordnlp.github.io/stanza/

2 Requirements

The following requirements have been collected :

Functional requirements

1. The pipeline must be dockerized for both deployment and development environments.
2. The pipeline must be deployable with a docker composition.
3. Adding a model to the pipeline must be as easy as adding its implementation.
4. The pipeline must accept requests via a web server running.
5. Fetching results of a specific model must have a separate endpoint.
6. Fetching results of a specific model must strictly accept unlabeled text.
7. Fetching results of multiple models must have a separate endpoint.
8. Comparing two or more models must have its own endpoint.
9. Comparing two or more models must accept labeled text.
10. Comparison of models must take the labeled text as a reference.
11. Comparison of models must include whether the model predicted the reference sentiment.
12. Comparison of models must include the confusion matrix.
13. A model must strictly process one sentence at a time.
14. A model's preprocessor must accept a list of text.
15. A model's preprocessor must send one sentence at a time to its model.
16. A model's preprocessor must gather all responses from the model and return it to the caller.
17. Developers must be able to change the hosting address of the pipeline web server.
18. Developers must be able to change the port of the pipeline's web server.
19. The web server must contain swagger documentation of the API endpoints.

Non-functional requirements

1. The pipeline must be horizontally scalable.
2. Every node must be separately horizontally scalable.
3. The pipeline must consistent of decoupled nodes.
4. Every model's preprocessor must be separately horizontally scalable.
5. The pipeline must handle a heavy workload.

6. Documentation must describe how to start with development.
7. Documentation must describe how to deploy the pipeline.
8. Documentation must describe how to add simple models.

3 Background

In order to start off with this short programming project, a small set of papers was used as the initial seed. This section will give a brief description of each paper and how it aided this project's progress. Unfortunately, due to the nature of this project, limited literature is available online and most of the reviewed literature was provided by the supervisors.

Previous bachelors project [1]

The dataset used in this project was gathered specifically for this purpose and was essential to its successful completion. Requests were sent to the pipeline using the dataset, and the sentiment of these requests had already been labeled, thus eliminating the need for additional manual labeling.

The bachelor's project enabled a better understanding of the usage and purpose of the pipeline. Through this project, my knowledge of terms, models, and technologies used to gather the data set, as well as the process of managing a complex task, was deepened.

Mining Cloud Cost Awareness — Is it possible?

This paper aimed to investigate if software developers are cognizant of the costs associated with deploying and delivering software in the cloud and explore the kind of information that can be extracted from GitHub projects.

Opinion Mining for Software Development: A Systematic Literature Review [6]

This paper provided a systematic review of the various applications of sentiment analysis in different software engineering activities. In doing so, it advanced the progress of this project by illustrating how researchers leverage the potential of sentiment analysis to gain insights into large-scale public opinion.
This paper has helped to elucidate the fact that sentiment analysis is not solely restricted to detecting sentiment polarity, but can be used to uncover other interesting and meaningful characteristics like :

- Emotion Detection
- Politeness Detection
- Trust Estimation

4 Methods

Initially, it was intended to utilize the code base of the preceding bachelor's project as the basis for this short programming project. However, upon attempting to work with the said code base, it became evident that due to certain oversights which had occurred during the previous bachelor's project, it was necessary to begin anew.
Reasons for this decision include :

- Using essential files on Google Drive that are not accessible to me
- Using Jupiter notebooks
- Loading the wrong data sets

After consultation with my supervisors, it was decided that it would be most beneficial to begin anew rather than attempt to bring the existing code base up to a functioning level. This approach allows for the work to begin without having to first address any prior issues.

5 Technology Stack

In this section, we will mention the technology stack we used and the reason behind the decision.

5.1 Ray

Ray Serve [5]is a scalable and easy-to-use framework for building and deploying machine learning models as HTTP microservices. It allows developers to easily deploy and manage machine learning models in a production environment. Ray Serve provides many features the most relevant to our project is automatic scaling. The purpose of Ray is to make it simpler to construct and manage large-scale, high-traffic machine learning applications, which is precisely what is required. Ray Serve is built on top of the Ray distributed computing system, which provides a fast and efficient execution engine for machine learning workloads. Therefore, Ray Serve is an excellent choice for this project since we are building a scalable microservice of a pipeline in a production environment.

5.2 Fast API

FastAPI [6]is a modern, high-performance web framework for building APIs with Python. It facilitates the creation of efficient, scalable, and maintainable web services. FastAPI uses the latest and greatest features of Python, including type hints, async and await syntax, which makes it simple and straightforward to build high-performance APIs. Additionally, it provides robust support for industry-standard tools and libraries, such as OpenAPI and JSON Schema, as well as comprehensive documentation and built-in validation capabilities. Finally, it perfectly integrates with Ray. Therefore, we chose FastAPI as our backend-framework.

5.3 Docker

Docker [7]is a platform that allows developers to package and run applications and services in a containerized environment. Containers are lightweight, standalone executables that contain everything needed to run an application, including code, libraries, dependencies, and configuration files. Docker provides a standardized way to create, distribute, and manage containers across different environments, making it easier to deploy and scale applications without worrying about differences in underlying infrastructure. With Docker, we can build, test, and deploy applications quickly and reliably, and we can share our code base more effectively by sharing container images and configurations. Which aligns perfectly with the requirements of this project.

5.4 Locust

Locust [8]is a popular open-source tool for load testing and performance testing of web applications. Locust allows developers to simulate concurrent user activity to measure the performance and scalability of their web applications. We can create test scenarios that simulated multiple concurrent users making requests to the sentiment analysis API endpoint.

[8]https://docs.ray.io/en/latest/serve/index.html
[8]https://fastapi.tiangolo.com/
[8]https://www.docker.com/
[8]https://locust.io/

6 Design and implementation

This section will provide an overview of the system design. The finalized version of the system can be accessed on GitHub [9].

6.1 General overview

The system consists of a main component that can be divided into the following parts:

- Ray cluster / Pipeline Deployment
- Main entry point (API)
- Model deployment(s)

All of the components are showcased in Figure 1

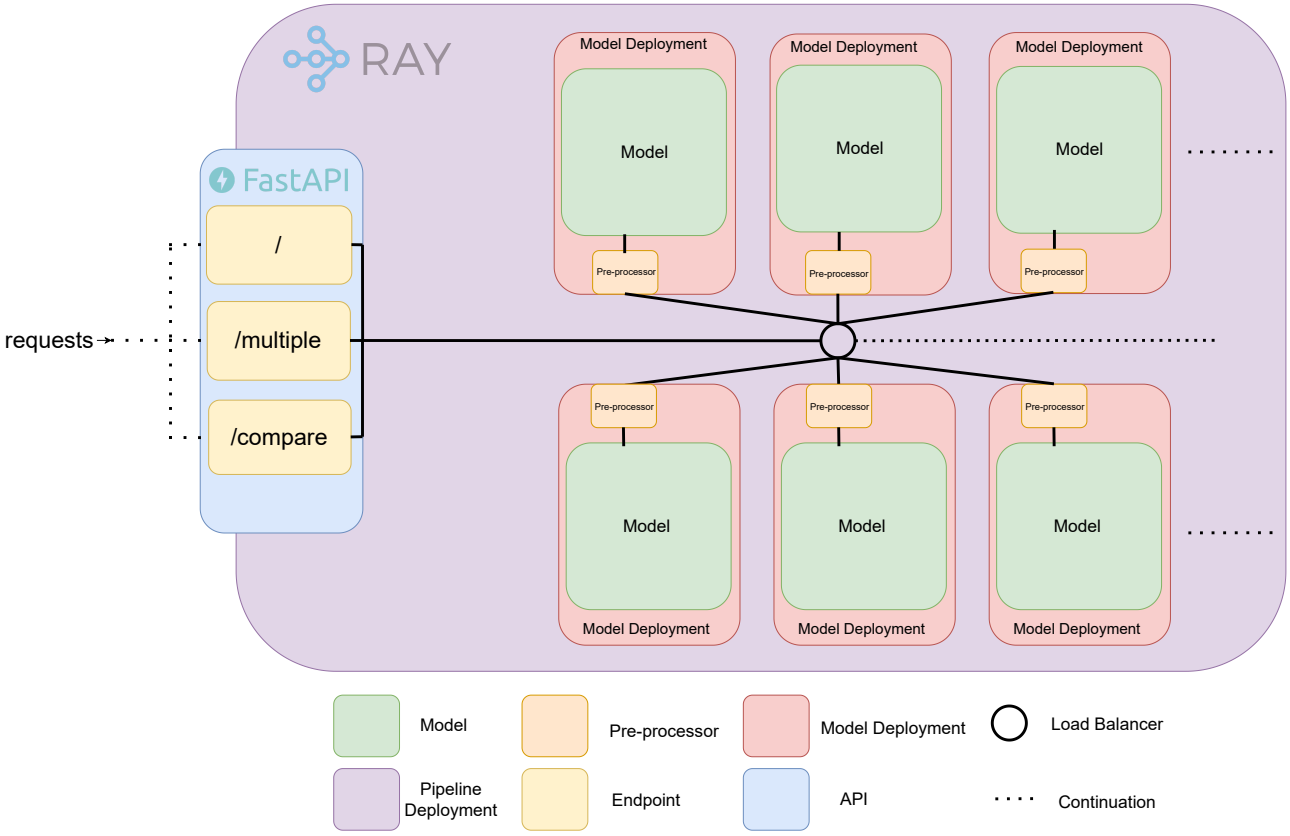

Figure 1: Ray cluster overview

6.2 Ray cluster / Pipeline Deployment

A Ray cluster is a group of machines that are configured to work together to execute distributed computations using the Ray framework. A Ray cluster typically consists of a head node, which serves

[9] https://github.com/Stylo2k/SentimentAnalysis

as the control plane for the cluster, and one or more worker nodes, which execute the computation tasks assigned to them by the head node.
The head node manages the resources and scheduling policies of the cluster, and coordinates the distribution of tasks to the worker nodes. The worker nodes are responsible for executing the tasks assigned to them, and communicating with the head node to receive new tasks and report their progress.
A Ray cluster can be deployed on a variety of infrastructure, including physical machines, virtual machines, and cloud instances. Ray also supports dynamic resource allocation, which means that worker nodes can be added or removed from the cluster on demand to scale the cluster up or down based on the workload.

6.3 Main entry point (API)

The API described has three endpoints, which are identified by their respective URLs:

- `GET /` This endpoint returns a list of all available models that can be used for classification.
- `POST /` This endpoint is used for text classification. The API takes in a list of text and the name of a classifier and returns a classification for each text in the list. The input data is structured as follows:

```
{
    "classifier": "text_blob",
    "text": [
        "string1",
        "string2"
    ]
}
```

- `POST /multiple` : This endpoint is used to classify a list of text with multiple classifiers. The input data is structured as follows:

```
{
    "classifiers": [
        "text_blob", "vader"
    ],
    "text": [
        "string1",
        "string2"
    ]
}
```

- `POST /compare` : This endpoint is used to compare the classification of a list of text by different classifiers. The input data is structured as follows:

```
{
    "classifiers": [
        "text_blob", "vader", "stanza"
    ],
    "text": [
```

```
        {
            "sentence": "string",
            "sentiment": "neutral"
        }
    ]
}
```

For the purpose of avoiding an overly cluttered document, the response bodies have been omitted. However, each endpoint provides the given text with its expected sentiment, while the comparison endpoint additionally provides a confusion matrix.
Every request received is routed accurately, and the load balancer ensures that requests are distributed equally across all available model deployments. In the event of an overload, Ray will automatically increase the number of replicas of affected model deployments in order to compensate. This way we ensure our system is always scaled up to handle as many requests as possible [10].

6.4 Model deployment(s)

What is depicted in Figure 1 as a model deployment is in essence a Ray deployment that consists of two separate, yet closely connected, Ray deployments.
In a Ray deployment, the head node acts as the central control plane, managing the resources and scheduling policies of the cluster, and distributing tasks to the worker nodes. The worker nodes are responsible for executing the assigned tasks and reporting their progress to the head node, which in turn provides new tasks. The deployment also includes the necessary software libraries and packages required by the application.
The two deployments are :

- Pre-processor
- Model

Decoupling the pre-processor from the model was a deliberate decision, as having them in separate Ray deployments provides several advantages over having them in the same deployment. Specifically, this arrangement allows for:

- **Scalability**: The separation of the preprocessor and the model facilitates independent scaling of each component, thereby improving the scalability and performance of the system. This allows for the optimization of each component according to its specific requirements, as the preprocessor can be scaled up to accommodate an increase in input data size without affecting the resources allocated to the model.
- **Modularity**: The decoupling of the preprocessor and the model facilitates the independent modification and updating of each component. As such, modifications to the preprocessor can be undertaken without affecting the model, and vice versa, allowing for ease of maintenance and evolution of the system over time.

[10] The maximum number of replicas which may be accommodated is contingent upon the resources available; beyond a certain point, further scaling is precluded due to the capacity of the host machine being reached.

- **Reduced Resource Usage**: The separation of the preprocessor and the model into different Ray deployments can result in a decrease of overall resource usage in the system. This is due to the fact that the preprocessor typically requires fewer computing resources than the model, making a joint deployment of both result in an overprovisioning of resources, thus leading to an increase in costs and a decrease in efficiency.
- **Flexibility**: The separation of the preprocessor and the model enables them to be deployed on distinct hardware or cloud environments, depending on the particular needs of each component. For example, the preprocessor may necessitate specialized hardware or software libraries that are not present on the same hardware as the model. By separating the components, they can be hosted on the most suitable hardware or cloud environment for their specific requirements. This point is not of great effect for this project, but is nonetheless an important advantage that is worth mentioning.

7 Lessons learned

Containerization with Docker: I had prior knowledge of containerization through Docker, however, it was not as in-depth as what was required for this project. As a result, I gained experience in configuring both production and development Docker images, as well as in utilizing Docker Compose.

Scalability & Distribution: The utilization of Ray, a distributed computing framework, provided an effective means of scaling the system. All with the help of locust. This necessitated the familiarization of the features of Ray, such as parallelism and distributed computing, to enable the optimization of performance and scalability. To be successful in this, the configuration of Ray's resources, the comprehension of the task execution model, and the tracking and solving of distributed tasks had to be initiated. As none of the courses I had taken up to this point had prepared me for this, it was the most demanding endeavor.

Monitoring and performance tuning: Deploying the pipeline involved monitoring its performance and making necessary adjustments to optimize its efficiency. I learned the importance of monitoring tools and techniques, analyzing performance metrics, and fine-tuning the pipeline based on real-time feedback to ensure smooth operation and optimal performance.

Finally, some of the obvious but important nonetheless. Constructing a pipeline, reading documentation, planning, and seeking help when stuck.

8 Results

Our testing results using locust. All three figures align.

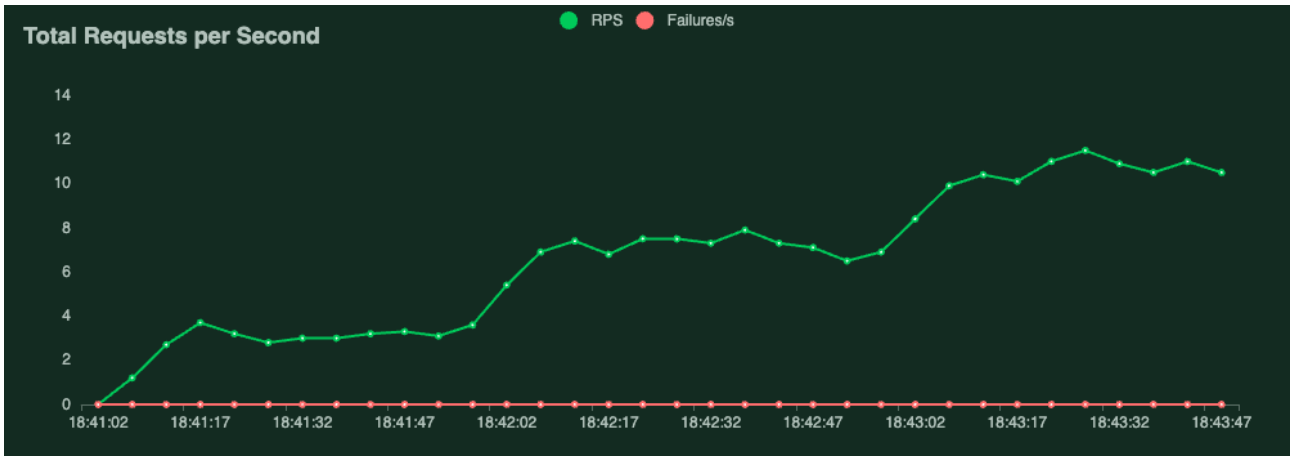

Figure 2: Requests per seconds to failure

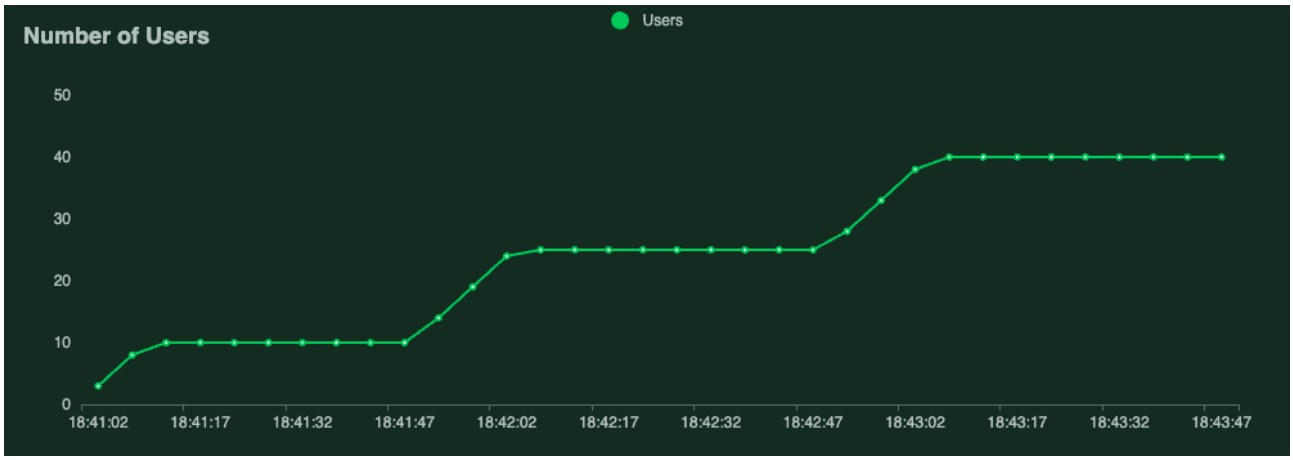

Figure 3: Number of users

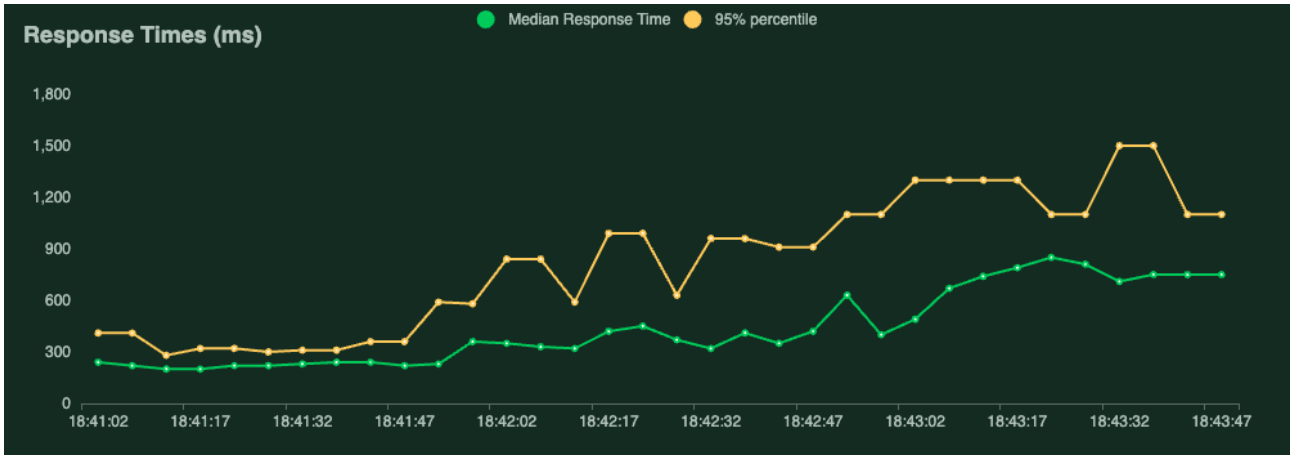

Figure 4: Response time

9 Conclusion

The development of a sentiment analysis pipeline using Ray, FastAPI, and Docker has proven to be a successful undertaking. This project has successfully provided a scalable and efficient sentiment analysis solution that can process large quantities of text data in real-time. The pipeline consists of Ray for distributed computing, FastAPI for web APIs, and Docker for containerization. The resultant solution has demonstrated its worth through its ability to quickly and accurately process large volumes of text data.

9.1 Improvements

A worthwhile endeavor is to research a technique to achieve a simpler way of adding a new model to the pipeline. Currently, the implementation of the model itself will have to be added, but there may be a way to just reference the model and the code will generate the implementation itself using some kind of external API. Additionally, Some issues may arise, one of which is the serialization of the model. From the limited research we have done, this has to be done for every model that is not serializable using pickle [11].

9.2 Future Work

This system will be utilized in a future thesis that investigates the prediction of cost-management actions related to Infrastructure as Code (IaC) artifacts changes. Furthermore, this system can be of use to anyone wishing to create a scalable pipeline and web server as a request handler for their sentiment analysis model(s).

[11] https://docs.python.org/3/library/pickle.html

Bibliography

[1] M. Berardi, M.-T. Penca, and R.-D. Boza, "Mining and analysis of cost-related decisions in cloud infrastructures," 2022. https://fse.studenttheses.ub.rug.nl/27946/.

[2] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, "Adoption, support, and challenges of infrastructure-as-code: Insights from industry," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 580–589, 2019.

[3] Y. Mejova, "Sentiment analysis: An overview," 2009.

[4] B. Polson, "Pipeline design patterns," in *ACM SIGGRAPH 2015 Courses*, SIGGRAPH '15, (New York, NY, USA), Association for Computing Machinery, 2015.

[5] C. V. Ramamoorthy and H. F. Li, "Pipeline architecture," *ACM Comput. Surv.*, vol. 9, p. 61–102, mar 1977.

[6] B. Lin, N. Cassee, A. Serebrenik, G. Bavota, N. Novielli, and M. Lanza, "Opinion mining for software development: A systematic literature review," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, mar 2022.